Security for Containers and Kubernetes

Learn how to implement robust security measures in containerized environments

Luigi Aversa

www.bpbonline.com

https://t.me/learningnets

Copyright © 2023 BPB Online

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 BPB Online 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.

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

First published: 2023

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55518-439

www.bpbonline.com

https://t.me/learningnets

Dedicated to

My beloved wife:

Natalya

&

My Parents ***Rita*** *and* ***Andrea***

https://t.me/learningnets

About the Author

Luigi Aversa has been working in the tech industry for more than 20 years, playing central roles in numerous projects as a technical leader and security engineer, delivering projects using Linux technologies, and combining DevOps skills with security acumen. Currently, he is a Staff Information Security Engineer at Grail. In the meantime, he successfully got many security certifications in the cyber security and security compliance fields. Furthermore, the author writes technical articles on information security, cyber security and related topics.

https://t.me/learningnets

About the Reviewer

Werner Dijkerman is a freelance cloud, Kubernetes (certified), and DevOps engineer. He's currently focused on, and working with, cloud-native solutions and tools including AWS, Ansible, Kubernetes, and Terraform. He is also focused on Infrastructure as Code and monitoring the correct "thing" with tools such as Zabbix, Prometheus, and the ELK Stack, with a passion for automating everything and avoiding doing anything that resembles manual work. He is an active reader, comics, non-fictional and IT related books, where he is a Technical reviewer for various books about DevOps, CI/CD and Kubernetes.

https://t.me/learningnets

Acknowledgement

I want to express my deepest gratitude to my family for their unwavering support throughout this book's writing, especially my wife Natalya. I also want to thank my parents Rita and Andrea for their encouragement and love throughout all my life.

I am also grateful to BPB Publications for their guidance and expertise in bringing this book to fruition. It was a long journey of revising this book, with valuable participation and collaboration of reviewers, technical experts, and editors.

I would also like to acknowledge the valuable contributions of my colleagues and co-worker during many years working in the tech industry and more lately in the information security field, who have taught me so much and provided valuable feedback on my work.

Finally, I would like to thank all the readers who have taken an interest in my book and for their support in making it a reality. Your encouragement has been invaluable.

https://t.me/learningnets

Preface

Building a secure containerized environment is a complex process that requires a comprehensive understanding of the container stack and the hardware and software infrastructure on which it is built upon. The recent raise of Kubernetes as a container orchestrator solution expands the complexity and the security challenges of the container stack either on premise or in the cloud. Securing both systems requires a good understanding of the threat landscape, reference to the associated risks and knowledge of the powerful tools that have become increasingly popular in the cybersecurity field.

This book is designed to provide a comprehensive guide to understanding security best-practices for containers and Kubernetes. It covers a wide range of topics, including the concepts behind the virtualization of the container stack, advanced topics such as securing container automation and orchestration, and how to secure the Kubernetes cluster for building secure microservices.

Throughout the book, you will learn about the key concepts and techniques needed to secure a container platform from the ground up, including hardware and operating system. You will also learn about best practices and design patterns to apply security best-practices to application and microservices and you will be provided with numerous practical coding examples to help you understand and reproduce the concepts.

This book is intended for security practitioners, DevOps engineers, security engineers, cloud engineers, platform engineers, and cloud architects who play a pivotal role in containerization and Kubernetes deployment. This book is also intended for experienced professionals who want to expand their knowledge with some peculiar security best-practices or techniques in building robust and secure container and Kubernetes stacks.

With this book, you will gain the knowledge and skills to visualize the entire container stack and the security gaps that should be addressed to reduce the attack surface and increase the security posture of your container and orchestrator platforms, but also to obtain hands-on strategies for measuring, analyzing and evaluate the impact of threats and vulnerabilities. I hope you will find this book informative and helpful.

Chapter 1: Containers and Kubernetes Risk Analysis – provides a high level overview of the risks associated with the implementation of the container platform

https://t.me/learningnets

and the Kubernetes orchestrator, including the risks associated with the underlying hardware and software infrastructure. It also provides a brief overview of the risk associated with container images and container registries as essential components of the container stack.

Chapter 2: Hardware and Host OS Security – presents a detailed overview of the main in-hardware security features that have direct impact on the virtualization technology and therefore on the container stack. It also presents a detailed overview of the main operating system security features that can be leveraged by one or more layer of the container stack including Kubernetes.

Chapter 3: Container Stack Security – provides an overview of the container systems available today to be integrated with Kubernetes, including Docker, and the security best-practices needed to reduce the attack surface and strengthening the network communication in a containerized environment. This chapter presents also a full secure connection section to enable Docker to use TLS communication that can be fully implemented to secure CI/CD systems.

Chapter 4: Securing Container Images and Registries – allows the reader to learn fundamental concepts related to container images and container registries. Image hardening and file configuration are the baseline upon which enabling a sufficient security posture, while vulnerability scanning helps to determine the image life cycle. This chapter also illustrates security strategies to store and retrieve container image either in private or public registries, and how and when applying security methodologies such as SAST or DAST.

Chapter 5: Application Container Security – describes in detail the security aspects of the microservices model by recalling application security frameworks such as OWASP CSVS and NIST SP-800.190. This chapter analyzes security testing models such as SAST, DAST, IAST and RASP and when applying these throughout the phases of the software development life cycle in a containerized environment.

Chapter 6: Secure Container Monitoring – shows an in-depth analysis of how to secure the container stack at any layer of the container infrastructure. It describes logic and methodologies behind container workloads, alerting and topology visualization; it also illustrates how to ingest metrics into the chosen SIEM, and how to detect a drift in a system behaviour by leveraging machine learning models.

Chapter 7: Kubernetes Hardening – provides a technical overview of the Kubernetes architecture and how to improve the security of the cluster by applying hardening to the control plane and data plane layers. This chapter highlights the importance of securing the network communication within the cluster with

https://t.me/learningnets

a detailed walkthrough of the security techniques adopted to secure the cluster runtime interface and the PODs. It also provides an overview of the common treats the Kubernetes cluster is exposed to, including the POD escaping technique.

Chapter 8: Kubernetes Orchestration Security – is dedicated to introducing the readers to the fundamental of the orchestration by discussing the complexity of the Kubernetes cluster and how to apply security best-practices in high available scaling environments. This chapter also asserts the importance of securing fundamental components of the cluster such as the internal API system and the admission controller, while highlighting the potential threat the orchestrator is exposed to, and what are the recommended countermeasure.

Chapter 9: Kubernetes Governance – explains how to achieve a robust governance security layer able to cover the main weakness the Kubernetes cluster by leveraging policy engines that can propagate network and security policies. The chapter also describes the admission controller threat model, how to secure resource management and what are the limitation or missing security features affecting the cluster today.

Chapter 10: Kubernetes Cloud Security – is dedicated to describing the security features of the most popular Kubernetes cloud service solutions such as AWS EKS, Azure AKS, and Google GKE. This chapter discusses the shared responsibility models typical of the public cloud providers, and the new 4C security model recently created by the Cloud Native Computing Foundation. This chapter also describes the security peculiarities of less popular Kubernetes services such as OpenShift, Rancher and Tanzu.

Chapter 11: Helm Chart Security – explains best-practices to secure the most popular Kubernetes package manger, how to introduce integrity into external packages handling by the mean of cryptography, and how to establish a circle of trust implementing public keys verification into infrastructure as code tools like Terraform. This chapter also explains how to scan helm charts for vulnerabilities and how to address supply chain security threats by adopting the SBOM model.

Chapter 12: Service Mesh Security – describes in detail the microservices architecture and security benefits that the service mesh brings by running in parallel with the workloads. This chapter explains the container network interface, why the choice to use the Envoy proxy in the service mesh architecture and how to secure it, and delves into the security aspects of the mutual TLS system. This chapter also describes Istio security features, and the zero-trust networking model.

https://t.me/learningnets

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/1i48l0x

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Security-for-Containers-and-Kubernetes**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At **www.bpbonline.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 BPB books and eBooks.

https://t.me/learningnets

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 **business@bpbonline.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 **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights 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.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

https://t.me/learningnets

Table of Contents

1. Containers and Kubernetes Risk Analysis 1

Introduction 1

Structure 1

Objectives 2

Host OS risks 3

Attack surface 4

System-level virtualization 5

Component vulnerabilities 6

Authentication 6

File system integrity 7

Image risks 7

Image vulnerabilities 8

Image misconfiguration 8

Embedded secrets 9

Embedded malware 10

Untrusted images 11

Registry risks 12

Non-secure connections 13

Stale images 14

Authentication and authorization 15

Container risks 15

Container runtime 15

Network traffic 16

The Application layer 17

Rogue containers 18

Orchestrator risks 18

Admin access 20

Unauthorized access 20

Network Segregation 21

https://t.me/learningnets

Workload levels 22
Worker node trust 23
Conclusion 23

2. Hardware and Host OS Security 25
Introduction 25
Structure 26
Objectives 26
Hardware security 27
Secure boot 29
Virtualization-based security 30
TPM 31
Trusted execution environment 33
DICE 34
Host OS Hardening 35
Linux namespaces 37
Control groups 39
Capabilities 41
Security Enhanced Linux 43
AppArmor 45
Seccomp 48
Conclusion 50

3. Container Stack Security 51
Introduction 51
Structure 52
Objectives 52
Container security 53
Containerd 57
CRI-O 57
Docker 59
Least privilege 59
Resource limitation 61

https://t.me/learningnets

Container isolation *64*
Namespaces *65*
AppArmor *66*
Seccomp *67*
Network security 68
Mirantis container runtime *71*
An interesting exclusion *72*
Secure connection 73
Server certificate *77*
Client certificate *79*
Enable dockerd TLS *81*
Secure CI/CD *86*
Update life cycle 94
Conclusion 95

4. Securing Container Images and Registries 97
Introduction 97
Structure 97
Objectives 98
Container image hardening 98
Building file configuration 100
Multi-stage builds *104*
Minimal and distroless images 106
Scanning and verifying images 112
Private and public registries 117
Registry authentication *120*
Role-Based Access Control 124
Auditability 125
Image control 127
Vulnerability management 130
Conclusion 136

https://t.me/learningnets

5. Application Container Security 137
Introduction 137
Structure 139
Objectives 139
Application Container Security 139
Threat intelligence 150
CI/CD Security integration 152
Shift left 153
Remediation 154
Managing privileges 155
Penetration testing 157
Third-party components 160
Conclusion 162

6. Secure Container Monitoring 163
Introduction 163
Structure 164
Objectives 165
Container activity 165
Docker engine monitoring 166
Containers monitoring 168
Host monitoring 171
Application monitoring 173
Workload observability 178
Anomaly detection 182
Externalise logs 187
Alerting 189
Topology visualization 190
Conclusion 192

7. Kubernetes Hardening 193
Introduction 193
Structure 195

https://t.me/learningnets

Objectives 195
Architecture 195
Control plane hardening 197
Worker node hardening 212
Securing network communication 213
Securing container runtime interface 215
POD security 219
POD escaping 224
Hardening tools 226
Updating life cycle 230
Conclusion 231

8. Kubernetes Orchestration Security 233
Introduction 233
Structure 234
Objectives 235
Authentication and authorization 235
API bypass risks 241
RBAC vs ABAC 245
Admission controller 247
Securing secrets 251
Cluster isolation 253
Audit logging 255
POD escaping privilege escalation 260
Assess and verify 261
Conclusion 265

9. Kubernetes Governance 267
Introduction 267
Structure 268
Objectives 268
Policy engines 268
Admission controller threat model 282

https://t.me/learningnets

Network policies 286
Resources management 290
Security policies 295
Limits and limitations 302
Conclusion 305

10. Kubernetes Cloud Security 307
Introduction 307
Structure 308
Objectives 308
Cloud native security model 308
Amazon elastic Kubernetes service 312
Azure Kubernetes Service 322
Google Kubernetes Engine 327
Red Hat OpenShift 334
Rancher 337
Tanzu 339
Conclusion 341

11. Helm Chart Security 343
Introduction 343
Structure 344
Objectives 344
Helm 344
Tiller 346
Integrity 349
IaC trust 351
Chart scanner 356
Dependencies 358
Conclusion 362

12. Service Mesh Security 363
Introduction 363

https://t.me/learningnets

Structure 365
Objectives 365
Overview 365
Architecture 367
Container Network Interface 369
Envoy security 372
Secret discovery service 378
Mutual TLS 381
Istio security 384
Zero-Trust networking 388
Conclusion 389

Index 391-402

https://t.me/learningnets

Chapter 1
Containers and Kubernetes Risk Analysis

Introduction

At the time of writing the most popular version control system, **GitHub** hosts nearly 143,000 repositories related to containers with over 23 million commits, while over 106,000 repositories are related to Kubernetes with over 3 million commits. The Kubernetes repository itself hosts nearly 110,000 commits. Those impressive numbers are clearly the sign of an exponential growth that highlights how the microservice age has evolved over the last few years. More than that, it is the sign of how the need to adopt containerized solutions and how to manage them has become prominent across the spectrum of the software development life cycle.

As containers and the use of Kubernetes grow, so does the need to secure the systems. The most common cause of incident is the "known threat": misconfiguration. Almost 70% companies reported a misconfiguration in their containerized environment, making "ignoring the basics" the most common type of vulnerability.

Structure

In this chapter, we will discuss the following topics:

- Host OS Risks
    - Attack Surface

https://t.me/learningnets

    - System-Level Virtualization
    - Component Vulnerabilities
    - Authentication
    - File System Integrity
- Image Risks
    - Image Vulnerabilities
    - Image Misconfiguration
    - Embedded Secrets
    - Embedded Malware
    - Untrusted Images
- Registry risks
    - Non-secure connections
    - Stales images
    - Authentication and Authorization
- Container Risks
    - Container Runtime
    - Network Traffic
    - The Application Layer
    - Rogue Containers
- Orchestrator Risks
    - Admin Access
    - Unauthorized Access
    - Network Segregation
    - Workload Levels
    - Worker Node Trust

Objectives

This chapter aims to provide a brief but significant overview of the main risks associated with the implementation of containerized solutions, including the technical components often forgotten, especially in agile environments where the DevOps methodology is applied.

https://t.me/learningnets

Host OS risks

First and foremost, what is it a host, and why it is an important part of risk analyzing? A host OS is the software that interacts with the underlying hardware, and it represents the first layer of security we should look at from the software standpoint. In *Chapter 2, Hardware and Host OS Security*, we will also look at the hardware layer and its intrinsic bond with the operating system. Container and orchestrator technologies have surfaced along with the adoption of DevOps practices that attempt to improve the integration between building and running applications; as a result, the Host OS or operating system is something that is often overlooked due to the shift in focus. Many readers are already familiar with the difference between the deployment of applications within containers and virtual machines, but it is helpful recalling the difference visually in *Figure 1.1, Virtual Machines and Containers Structure*, facilitating the understanding of the risk this section aim to address. Refer to the following figure:

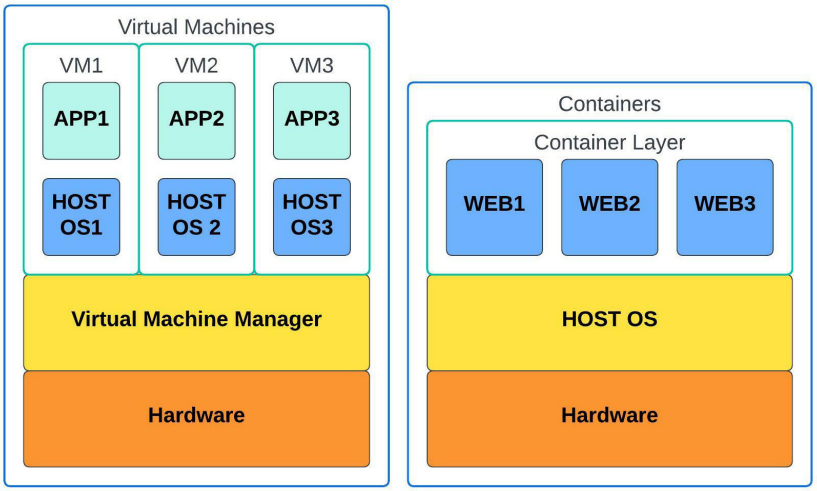

***Figure 1.1**: Virtual Machines and Containers Structure*

Figure 1.1, Virtual Machines and Containers Structure shows that regardless of the deployment methodology, the operating system is a crucial component of that deployment, except for some dedicated Cloud services like AWS ECS or Azure container instances where the burden of maintaining the underneath OS layer shifts back to the Cloud provider.

Both approaches allow multiple applications to share the same hardware infrastructure, but while the virtual machines use a hypervisor that provides hardware-abstraction via a virtual machine manager, the containers approach allows multiple applications to "share" the same operating system. From the security perspective, the hypervisor is also responsible for providing hardware-level isolation across virtual machines,

https://t.me/learningnets

while the container service is responsible for enabling hardware-level resources for running containers.

The thoughts about Cloud Managed Services would include a wider argumentation that is not the objective of this chapter, so it is deferred to *Chapter 10, Kubernetes Cloud Security*, for a deeper analysis.

Attack surface

An operating system has an attack surface as much as any other platform or system. The extension of the attack surface is strictly connected to the type of operating system and to the technical philosophy behind it. A Linux desktop distro would potentially have a wider attack surface than a Linux server minimal distro, and a Windows 11 system would potentially have a wider attack surface than a Windows Nano server system.

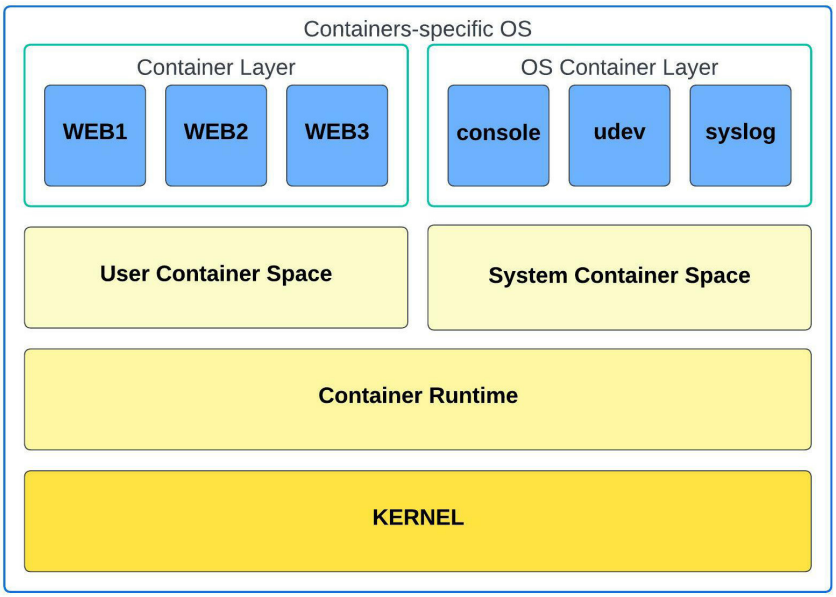

***Figure 1.2**: Container-specific OS*

There are essentially two types of Host OSes: ***General-purpose OSes***, such as Ubuntu, openSUSE Leap, and RedHat Enterprise Linux; and ***Container-specific OSes***, such as CoreOS Container Linux (now Fedora CoreOS), openSUSE Leap Micro, and RancherOS. The former category is the Host OS as we know it, typically used in any known application environment, while the latter has been specifically designed to have a minimalistic approach to run containers. In some cases, such as **openSUSE MicroOS**, **RancherOS** or **Clear Linux**, the Host OS itself is a containerized abstraction of the operating system, capable of providing atomic updates via rolling release

https://t.me/learningnets

distribution, where any single service, including system service as *udev* or *syslog*, is running as a container.

Adopting container-specific OSes could be initially challenging, but they provide immediate relief from the security standpoint, as shown in *Figure 1.2, Container-specific OS*, their attack surface is minimal, and they are container-optimized; that means they often provide a read-only filesystem, a basic set of services enabled on boot, and basic hardening best practices. Container-specific OSs are prone to reduce and mitigate the typical risks associated with general-purpose OSes distros, where a costly hardening process should be implemented to achieve an equal security posture.

System-level virtualization

This feature has been also described as "shared kernel" capability, leaving the door open to misinterpretation. Containers are not running the kernel on their own, and they are not sharing the kernel with the underlying operating system, not in the way in which the word "sharing" would be intended anyway. On the contrary, the container daemon is intercepting all the system calls that require kernel executions, but it is borrowing resources rather than effectively running them.

This technology uses the unique capability of *nix systems to share their kernels with other processes, achieved via a feature called **change root**. The **chroot** feature was initially thought to provide security isolation to processes running on a system without limiting the availability of the resources from the system itself; then, it evolved in what is known as container-based virtualization today. Readers with less system administrator background can think of this like an enhanced Python **virtualenv** where the purpose is not only to create isolated Python environments with full control on versions and Python modules but also with the capability of running anything else allowed by Linux. It stands out that being a container capable of running system calls via kernel execution, it represents a threat to the security of the system. There are a few basic but effective mitigation techniques applicable to this use case:

- Keep the kernel updated
- Use only SSH authentication
- Disable SSH password authentication in favor of SSH Keys
- Remove the root user
- Implement the Kernel Lockdown feature

Of the given list, the least known feature is likely to be the Kernel Lockdown feature. As per Linux main page description at **https://man7.org/linux/man-pages/man7/kernel_lockdown.7.html**.

https://t.me/learningnets

Note: The Kernel Lockdown feature is designed to prevent both direct and indirect access to a running kernel image, attempting to protect against unauthorized modification of the kernel image and to prevent access to security and cryptographic data located in the kernel memory while still permitting driver modules to be loaded.

Component vulnerabilities

The Linux Operating System primarily has three components: the Kernel, the System Library and the System Utility; this is illustrated in *Table 1.1 – Kernel Components*:

Component	Description
Kernel	The Kernel is the core part of Linux. It consists of various modules, and it interacts directly with the underlying hardware.
System libraries	System libraries are non-volatile resources used by applications or system utilities to access Kernel's features.
System Utilities	System utilities are software tools responsible for executing individual user-level tasks.

***Table 1.1**: Kernel Components*

Like any other software, these components may present vulnerabilities. And due to the criticality of their functions and the proximity with low-level code execution, they can greatly impact the integrity of the system on which they are running. All the components should be kept updated, not just the kernel.

This is particularly important for the container runtime components, as newer releases often add security protections beyond simply correcting vulnerabilities. The immutability guaranteed by the *Container-specific OS* with no data stored persistently and no application-level dependencies enhances a stateless operating mode, significantly increasing the host's security posture.

Authentication

The operating system is exposed to risk anytime users log in to the system to directly manage anything that is pertinent with the business objectives. In a post-COVID world, where working from home is normal, connecting from unsecure networks is, unfortunately, very common.

Even if most container deployments rely on CI/CD pipelines and orchestrators to distribute the load across hosts, logging on to the systems is still a very common (not recommended) practice, especially for troubleshooting purposes.

https://t.me/learningnets

Login sessions should be monitored and audited when needed, **sudo** limited to a known number of identified individuals. Interactive user login should be minimized, and most often forbidden, unless security concerns need to be addressed.

File system integrity

Container misconfigurations can expose host volumes to risk. A container can only access the files stored as part of the container image, therefore information should be considered "non-persistent data", in alignment with the ephemeral nature of the container philosophy. There is no real necessity to share files between Host OS and containers; it is a bad practice. Containers should run with the minimal set of file system permissions required.

Image risks

A container image is a static file containing executable code that can be used to create a running container. Images are efficient because they allow users to include all the elements required for an application into one package. Each image consists of a series of layers that can be combined via **UnionFS** into a single layer. There are essentially three types of layers:

- The base image layer
- The image layer
- The container layer

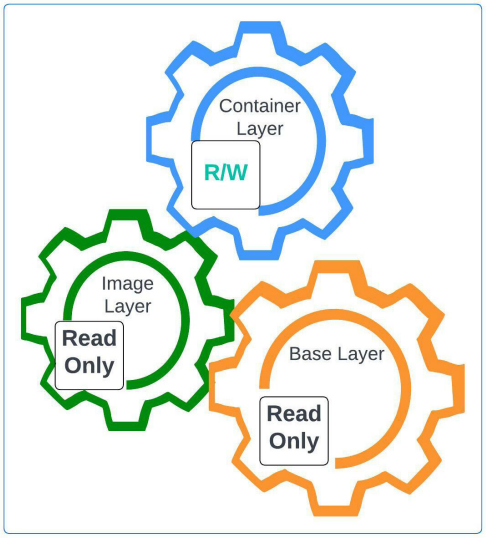

***Figure 1.3**: Image Layers*

https://t.me/learningnets

Of the mentioned three, only the container layer is writeable; the other two are read-only, as shown in *Figure 1.3 – Image Layers*.

Image vulnerabilities

Images are essentially static files containing executable code used to run a specific application. It is a good practice to use the most recent packages and keep the image as updated as possible, but we need to keep in mind that the image must have an assigned life cycle, as the software contained within the image becomes outdated over time and may contain vulnerabilities. The challenge with images is that the updates must be made upstream, and it involves triggering a redeployment.

Obtaining visibility into the application framework other than only the base layer of the image is essential, and it provides a reference policy framework to enforce quality control on the image creation process.

Image misconfiguration

To address configuration defects and fix configuration files containing misconfigured code, the framework illustrated in *Figure 1.4, Image Misconfiguration Framework* can be adopted:

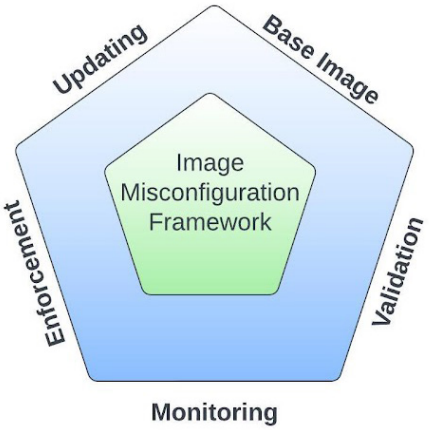

***Figure 1.4**: Image Misconfiguration Framework*

Preferring minimal base images like **Alpine Linux** or **Windows Nano Server** rather than fully installed operating system distributions is the first security requirement that should be satisfied. When there is no need for general system libraries, graphic user interfaces or unused services keeping the image tidy and clean limits the attack surface. Introduce a validation mechanics of the configuration settings to identify any drift in the configuration that could cause harm. Monitor the base image modelling

https://t.me/learningnets

framework to identify possible threats and enforce quality control of the image by introducing a "blessing" procedure. Only images with a minimum set of standards should be allowed to be created, and those standards should include policies like including the "run as" for non-root users and disabling SSH. Use the immutable feature of container systems to execute rolling updates.

Embedded secrets

The key word of this section is "*embedded*". It is common practice when building an image, with configuration file like Dockerfile for example, especially in testing environments, to provide all the information needed right from the get-go, including credentials. All the parameters needed to make everything working are *embedded* into the code, as the image itself is not really what we are working on; rather, it is what the image contains we care about.

From the risk assessment standpoint, a secret is any confidential data that would put information at risk if exposed. Secrets should be stored outside containers, and any other piece of software for that matter, and should be *consumed* on a need basis and rotated at given intervals. Refer to the following figure:

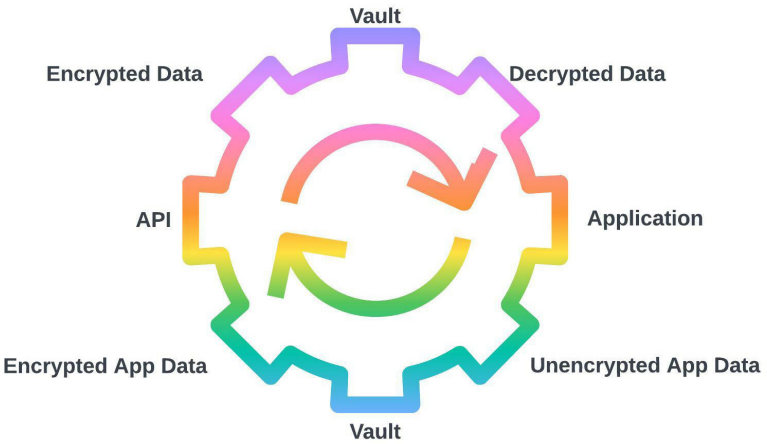

***Figure 1.5**: Key Management*

Container solutions either on premises or in the Cloud can provide key management systems. *Figure 1.5, Key Management*, shows a typical key management workflow in a microservice environment, where the application requests the key from the vault system, which is decrypted as part of the key management life cycle. The decryption and encryption mechanisms are provided through API calls managed securely by the vault system.

For instance, Docker Swarm has its own key management system, and AWS KMS is likely the most known Cloud Key Management service. However, there are similar

https://t.me/learningnets

solutions worth noting, such as Hashicorp Vault or Azure Key Vault. An interesting tool that can help identify secrets in the code is **SecretScanner**, and it is available at **https://community.deepfence.io/docs/secretscanner.**

Note: Deepfence SecretScanner can find unprotected secrets in container images or file systems. SecretScanner is a standalone tool that retrieves and searches container and host filesystems, matching the contents against a database of approximately 140 secret types.

Embedded malware

Malicious code could be unintentionally or intentionally packaged as any other software or component of the image, and it would have the same capabilities and privileges as any other component posing a serious risk to the system and infrastructure.

Palo Alto Networks **Unit 42** researchers have identified several different versions of Docker images containing ***XMRig*** used to mine Monero cryptocurrency. The threat actor used a Python script called *dao.py*, which was baked inside the Image and updated to Docker Hub. The Image was then downloaded 2 million times and was able to feed a crypto wallet for over $36 million.

Container images should be scanned regularly for known vulnerabilities; tools like Quay, Clair or Anchore can run static image analyzing even on a layer-by-layer basis, but those tools have a large footprint.

When considering shift-to-the-left in your DevSecOps pipeline, **Static Application Security Testing** (**SAST**); and **Software Composition Analysis** (**SCA**) are the methodologies that come to mind. Refer to the following figure:

Figure 1.6: *Secure Software Development Life Cycle*

Adopting a zero-trust security model is implementable, thanks to tools like MetaDefender Jenkins plugin available at **https://plugins.jenkins.io/metadefender.** *Figure 1.6*, **Secure Software Development Life Cycle** (**SSDLC**), illustrates how to implement scanning tools inside the **Software Development Life Cycle** (**SDLC**), adding the facto a security layer to the CI/CD pipeline.

https://t.me/learningnets

MetaDefender checks Jenkins builds for malware before releasing the build, and it has the great feature to include over 30 **Antivirus** (**AV**) engines and a **Proactive Data Loss Prevention** (**DLP**) system, resulting in great security efficiency for the CI/CD pipeline. MetaDefender is also available for TeamCity, Kubernetes via Terraform and Helm Chart or for AWS CloudFormation.

Untrusted images

Untrusted images are identified as non-official images or images downloadable from third-party repositories. The difficult part is to create a mechanism to identify "trusted images":

- First of all, avoid the *latest* tag when an image is pulled; always be declarative in choosing the version of the image needed.
- Use an approved image, also called *blessed*, by an expert of the security team.
- Establish a circle of trust by inspecting the base image.

For example, if you run **docker inspect** on the ubunut:18.04 image, you get the following:

```
1. "RootFS": {
2.     "Type": "layers",
3.     "Layers": [
4.         "sha256:49c23cd3c582026251e2ee4adde9217329f67aef-
   230298174123b92a7a005395"
5.     }
```

If you build a new image using the ubuntu:18.04 as base image using the following Dockerfile:

```
1. FROM ubuntu:18.04
2. RUN apt-get update
3. ADD ciao.txt /home/my-user/ciao.txt
4. WORKDIR /home/my-user
```

Inspecting the new image will highlight that both images share the same first layer that belongs to the initial ubuntu:18.04 base image:

```
1. "RootFS": {
2.     "Type": "layers",
3.     "Layers": [
```

https://t.me/learningnets

```
4.          "sha256:49c23cd3c582026251e2ee4adde9217329f67aef-
   230298174123b92a7a005395",
5.          "sha256:52f389ea437ebf444d1c9754d0184b57edb-
   45c912345ee86951d9f6afd26035e"
6.      }
```

Another interesting tool for exploring and inspecting container images is ***dive***. It is a command line tool with some interesting basic feature:

- While inspecting image contents broken down by layer, the contents of that layer combined with all previous layers is shown.
- Files that have changed, been modified, added, or removed are indicated in the file tree. This can be adjusted to show changes for a specific layer or aggregated changes up to that layer.
- Image efficiency estimation: the basic layer info and an experimental metric to identify how much wasted space the image contains.
- Build and analysis cycles: building a Docker image and performing an immediate analysis with one command: **`dive build -t some-tag`**.

Registry risks

A container registry is a repository used to store and access container images**.** Container registries can support container-based application development, often as part of DevOps processes. It is the natural evolution of what system administrators have known for years to be simple the "repos" to use in conjunction with tools like *rpm*, *zipper*, and *apt-get*.

Despite the name, a registry is *just another server system*, with one or more services exposed to a port listening for connections, and therefore exposed to risks either by storing compromised images or by granting access to an entity missing the appropriate level of permissions. There are essentially two types of container registries:

- Public registries are used by individuals or small teams that want to quickly get up and running. However, this can bring more complex security issues like patching and access control.
- Private registries provide security and privacy implementation into enterprise container image storage, either hosted remotely or on-premises.

Most cloud providers offer private image registry services:

- AWS ECR Elastic Container Registry

https://t.me/learningnets

- Microsoft ACR Azure Container Registry
- GCR Google Container Registry

Non-secure connections

Registries should allow connections only over secure channels in order to perform pushes and pulls between trusted endpoints via encryption in transit mechanisms. Public registries should already have such features in place, where HTTPS and TLS are the standard nowadays, but essentially, the registry acts like any other publicly exposed system or website in this case.

Things become quite interesting with private registries where technically, unless the private registry is deployed via a software as service model, it is necessary to enforce security. There are enterprise versions, also known as self-hosted systems like JFrog Container Registry, Docker Registry, Nexus or GitHub Container Registry, where the exposed service is often running on HTTP only, with no certificates.

Unfortunately, self-hosted services like Jfrog Artifactory running on AWS EC2 Instances with security groups allowing connections on port 80 or teams enabling the *insure-registries* feature to avoid the burden of setting up a secure connection are not uncommon scenarios. A good way around this is to use self-signed certificates in a few simple steps:

1. Generate your own certificate:

```
1.  $ openssl req \
2. -newkey rsa:4096 -nodes -sha256 -keyout your-dir/domain.key \
3.  -addext "subjectAltName = DNS:your-registry.domain.com" \
4.  -x509 -days 365 -out your-dir/domain.crt
```

2. On Linux, copy the **domain.crt** file to **`/etc/docker/certs.d/myregistry-domain.com:5000/ca.crt`** on every Docker host.
3. On Windows, right-click on the **`domain.crt`** file and choose to install certificate. When prompted, select **local machine** as the store location and **place all certificates in the following store**.
4. Click on **Browser** and select **Trusted Root Certificate Authorities**.
5. Click **Finish** and restart Docker.
6. Restart the registry, directing it to use the TLS certificate.

```
1. $ docker run -d \
2.  --restart=always \
```

https://t.me/learningnets

```
3.  --name registry \
4.  -v "$(pwd)"/certs:/certs \
5.  -e REGISTRY_HTTP_ADDR=0.0.0.0:443 \
6.  -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt \
7.  -e REGISTRY_HTTP_TLS_KEY=/certs/domain.key \
8.  -p 443:443 \
9.  registry:2
```

Please note this is just an example to address the risk of non-secure communication when using container registries, a full example will be provided in *Chapter 4, Securing Container Images and Registries*.

Stale images

There is a wrong tendency in preserving images for long time, sometimes because the image is the alpha or beta version of today's RC1, RC2 or stable version, or maybe because a bug is later introduced in a following version, which was not affecting the older release. Then, the application evolves, and sooner or later, the previous versions of the same image are forgotten, or maybe the business has not defined clear policies about the image life cycle.

The result is that older images become outdated very soon, making software, components or libraries vulnerable. Those images are not posing a threat for the simple reason of being stored in the registry only, but they do increase the likelihood of accidental deployment of risky images.

There are two approaches to mitigate the issue:

- Vulnerable images should be pruned at regular intervals, according to the Software Development Life Cycle cadence and the size of the team working on the development. If the team deploys a release once a week, it is reasonable to prune images every quarter, while a different logic can be applied to different use cases.
- Use the tags to identify the correct deployment strategy, ingesting in your CI/CD pipelines declarative naming convention using immutable versions. Avoid using the *latest* tag, and always declare the version needed, so the commit will highlight moving from version:1.7 to version:1.8 for example, helping keep the registry tidy and clean.

https://t.me/learningnets

Authentication and authorization

Account federation allow users to use a single account to login onto different platforms without the need to re-authenticate their identity, tools like OKTA are very common in modern enterprises, allowing a centralized login system to span over several technical solutions. All the write access to the registry should be regularly audited.

Differentiate between who can pull and who can push to the registry; do not assume that permissions are equally granted. Also, use a segregation approach, the Team A can push to the repository A but not to repository B and vice versa.

Obtain control on the push logic, implementing the CI process to allow images to be signed by authorized individuals; in a DevSecOps model, those should be member of the Security Operations department, so images are pushed to the registry only if they meet eligible criteria like passing vulnerability scans.

Container risks

In a previous paragraph (*Host OS Risks*), we learned the difference between Virtual Machines and Containers. We can recall here that containerization works as virtualization system at the operating system layer, essentially enabling hardware or resource abstraction via the container system manager. The **Open Web Application Security Project** (**OWASP**) has established the **Container Security Verification Standard** (**CSVS**), and also created a quick cheat sheet, which comes handy for a quick read and verification of the basic container security rules:

https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html.

Container runtime

The container runtime is the element of a container platform that manages the life cycle of a container. It is essentially the so-called Linux daemon service that creates, starts, stops and destroys containers, and also manages the storage and networking for a container.

Figure 1.7, Container Runtime, illustrates the process chain needed for a containerized platform to work properly, starting with the Docker Engine that encapsulates several other child processes to allow containers pre and post processing. The **`containerd`** system is the parent process of many **`shim`** child processes. If **`containerd`** fails, all

https://t.me/learningnets

the child processes will automatically fail as well. This is what we call a "single point of failure":

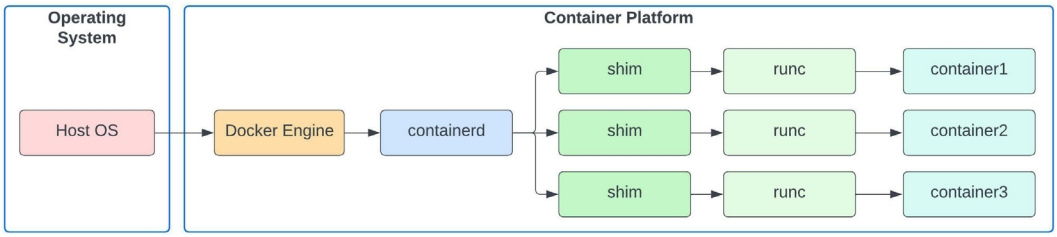

***Figure 1.7**: Container Runtime*

Due to its nature, the runtime is the point of connection between the containerized platform and the operating system (left panel of *Figure 1.7*; a compromised container instance would potentially allow a threat actor to pursue lateral movements, eventually gaining access to other container instances, or even the underlying operating system. This threat vector is called *container escape*. There are two main reasons why this could happen:

- Insecure configurations
- Runtime software vulnerabilities

The CIS Docker Benchmark provides a vast range of details and recommended settings, but operationalizing those is challenging. The opposite approach would suggest enabling technologies like SELinux or AppArmor to enhance control and isolation for containers running on Linux, while a good monitoring solution like Sysdig FALCO would detect unexpected behavior and intrusion detection in real time. This is also where good governance on the orchestrator side becomes valuable, for instance, blocking the orchestrator through a security policy to deploy anything to a vulnerable runtime, as we will discuss in *Chapter 9, Kubernetes Governance*.

Network traffic

In a containerized platform, the running container is the innermost component; it is, indeed, the result of the container process. Therefore, in order to communicate externally, it would generate egress traffic, which is notoriously difficult to manage.

Containers require a network layer, which is the default bridge network. The better approach is to use a custom bridge network, ensuring that containers cannot communicate with each other.

https://t.me/learningnets

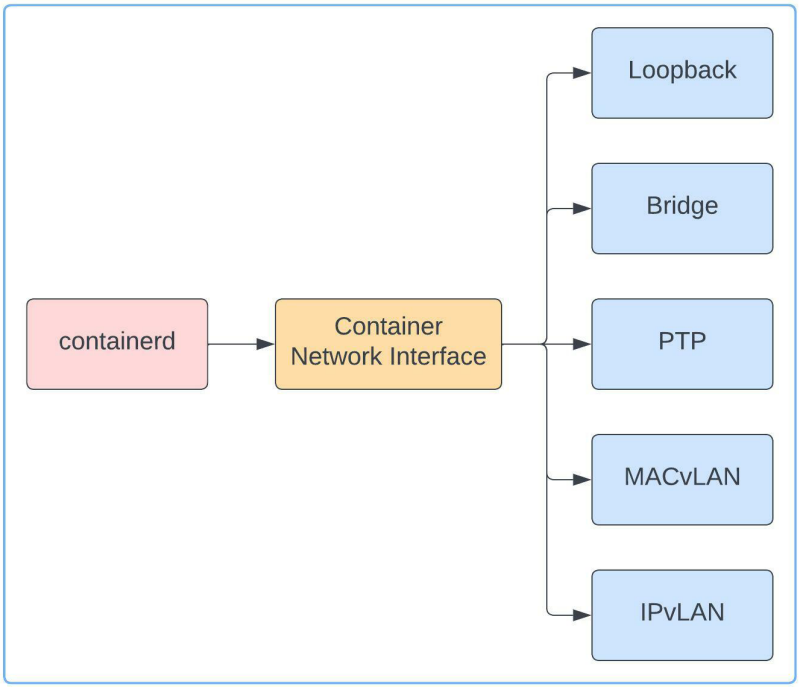

***Figure 1.8**: Container Network Traffic*

Figure 1.8, Container Network Traffic, shows another layer of the container stack: the network layer. Due to the use of a bridged network, normal network devices are usually blind to the container network traffic. It is important to assess the container networking surface, understanding inbound ports and process-port bindings, but it also important to have a proper network monitoring system in place to detect traffic flows between containers and other network entities or between themselves.

The Application layer

An often-underestimated issue is the consideration of the application that the container is running. This is not a problem with the container itself, clearly, but it is a typical flaw of container environments. While this extends the scope of the security argument indefinitely, it is good to understand that the container environment is not the only aspect of the security landscape we need to look at. Readers interested in how to secure applications can refer to the OWASP TOP 10; it is a very good place to start.

A web application could be vulnerable to cross-site scripting and could be used as an attack vector to compromise the container. It is needed to detect abnormal behavior in applications in order to take corrective action and prevent incidents. The Mitre provides a comprehensive list of attack tactics and techniques, which are useful

https://t.me/learningnets

when it comes to applying countermeasures and to analyzing the application's activity; visit **https://attack.mitre.org** for a comprehensive overview. The focus in relation to the current argument is on the following detections:

- Forbidden system calls
- Forbidden process execution
- Changes to configurations files or executables
- Write attempts to forbidden locations
- Network traffic to unexpected network destinations

Applications should be "contained" in a separate filesystem, keeping the root filesystem in read-only mode, to provide isolation between the container itself and the application.

Rogue containers

Rogue containers are unplanned or unexpected containers deployed in a container platform. This is quite common in staging or testing environments. Separate environments for development, testing, and production are highly recommended, with specific controls to provide **Role Based Access Control** (**RBAC**). Institute a triage process to act as incident response to any malicious container deployed:

- Information gathering
- Forensic analysis
- Lesson learned

Container creation should be associated with individual user identities and logged to provide auditing of activities when needed.

Orchestrator risks

In computing, orchestration is the capability of a system to automate configuration, deployment, and management of computer systems and software. Containers can provide microservice-based applications, which is a deployment unit and self-contained executable environment. Containerized microservices are much easier to orchestrate because they include storage, networking and security in a single operative instance. Therefore, container orchestration is the capability of a system to automate deployment, life cycle and networking of containers. Google introduced the open-source Kubernetes platform in 2015, largely based on their internal orchestrator project called Borg. Since the beginning, Kubernetes has been the most popular container orchestrator. Kubernetes runs workloads by placing containers

https://t.me/learningnets

into Pods running on Nodes. A node may be a virtual or physical machine, depending on the cluster. Each node is managed by the control plane and contains the services necessary to run Pods.

There are two areas of concern for securing Kubernetes:

- Securing the cluster components that are configurable
- Securing the applications that run in the cluster

Securing the applications is not in scope for the current analysis, but it will be subject to a deep review in *Chapter 5, Application Container Security*. The following diagram provides an overview of the Kubernetes Orchestrator:

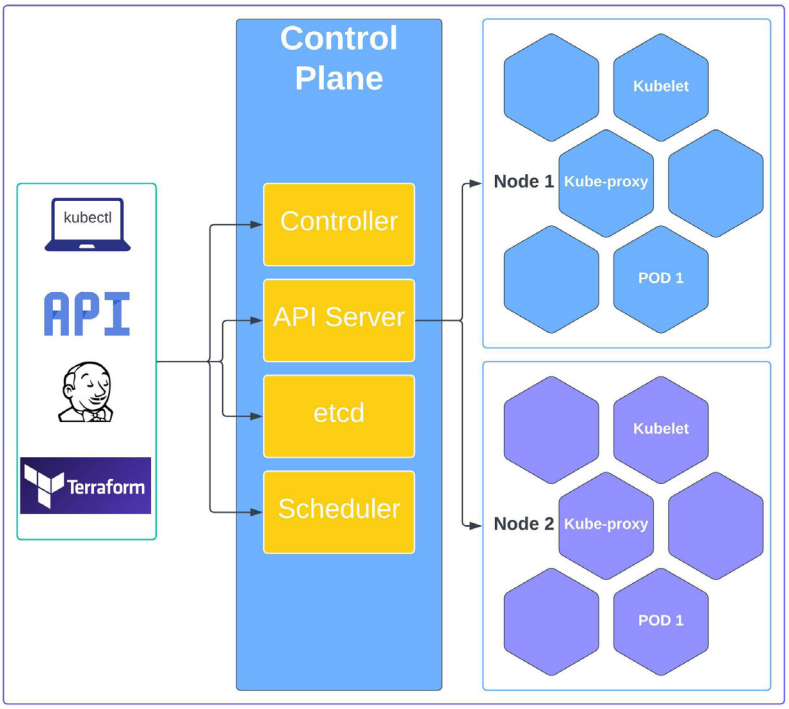

***Figure 1.9**: Kubernetes*

Figure 1.9, Kubernetes, provides a high-level overview of the main components of the Kubernetes Orchestrator.

Despite Kubernetes being the most popular open-source container orchestrator, it is worth nothing that it is not the only one available; a few alternatives are given as follows:

- Docker Swarm is an open-source orchestrator built on the Docker runtime engine.

https://t.me/learningnets

- Apache Mesos is a UC Berkeley project. It is the go-to option for companies like Airbnb, Apple and Twitter.
- Red Hat OpenShift is a platform-as-a-service (PaaS) built on Kubernetes.
- HashiCorp Nomad is a platform focused on scheduling and management processes.
- Rancher is an open-source platform that provides a manageable, observable, and scalable solution for managing multiple Kubernetes clusters.

The focus of this section is to try to abstract the security concerns common to the most popular container orchestrator solutions on the market today, trying to identify common patterns that would potentially affect any individual orchestrator. We will then shift the focus solely on Kubernetes in *Chapter 7, Kubernetes Hardening*, *Chapter 8, Kubernetes Orchestration Security,* and *Chapter 9, Kubernetes Governance*.

Admin access

Many container orchestrators were designed with the assumption that users would be administrators. This is a typical pattern in complex system where the permissions are left wide open to facilitate the development. However, a single orchestrator usually runs several applications, each belonging to a different team. It is necessary scoping out and tailoring specific permissions for each team using the following:

- The least privileged model
- Minimizing distribution of privileged tokens
- Implementing a role-based access control model
- Forbid access to secrets

Specifying a role and a corresponding RoleBinding for the account ensures that only legitimated API resources are accessed when needed.

Unauthorized access

Lack of visibility into account access governance within the cluster is a key risk. Compromise of system credentials or secrets maintained within the container orchestrator may result in generation of false identities. Container orchestrators often include their own authentication directory service; this can lead to bad account management practices.

Container may run on any given node, so data required by the application must be available to the container regardless of which host it has been deployed on. Data must be encrypted at rest to prevent unauthorized access. Access to cluster-wide

https://t.me/learningnets

accounts should be tightly controlled, and it should implement single sign-on to the existing directory systems.

Network Segregation

Traffic between nodes can be compared to traffic between containers; the difference here is the higher degree of complexity ingested via the consideration that multiple PODs on Node A could potentially talk or communicate with multiple PODs on Node B. Encrypting the "internal network traffic" increases the level of difficulty in gaining visibility, leaving any monitoring tool blind to what the orchestrator is doing internally, as we will see in *Chapter 12, Service Mesh Security*.

Container orchestrators should be configured to separate network traffic by applying network segregation. This concept is very similar to the application of the VLANs. The criteria or logic applied really depends on the specific use case, but to provide some context, containers should ideally be grouped by functionality.

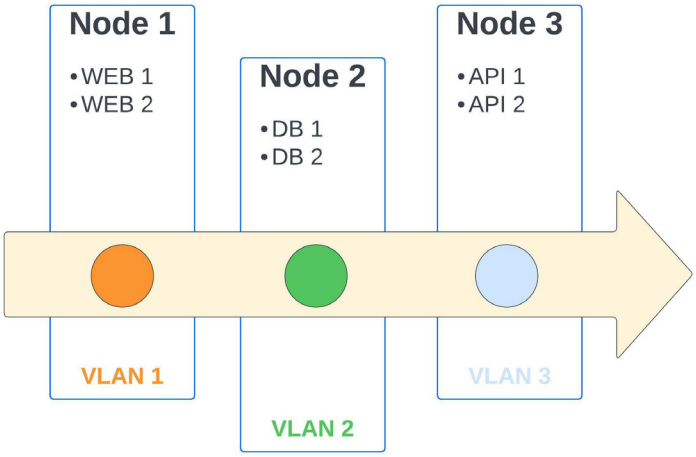

***Figure 1.10**: Network Segmentation Logic*

All the WEB PODs have no reason to share the same network segment of the API PODs; similarly, all the MySQL PODs have no reason to sit on the same network segment of the WEB PODs. This approach is called *per-app segmentation*. The per-app segmentation model is shown in *Figure 1.10, Network Segmentation Logic*.

Another approach highlights the localization of the applications in relation with their allocated network location. For instance, public-facing apps should sit in a virtual network that is separated by applications classified as "internal", which do not need to be exposed and are serving only internal processes.

https://t.me/learningnets

Another approach is to define virtual networks based on the sensitivity level, depending on what kind of data the applications are able to access or on the stage of the software development life cycle the applications are on. Therefore, in the last case, all the staging applications should sit in one network that is separated from all the production applications.

From the **Software Development Life Cycle** (**SDLC**) perspective, it is not unusual to have deployment in code with tools like Terraform, Azure Resource Manager or CloudFormation; so, it is useful to have a tool like CheckOV that can scan flat files. It is available at **https://www.checkov.io** and can provide static code analysis for scanning **Infrastructure as Code** (**IaC**) for misconfigurations.

Readers may argue that having all the production applications running in one virtual network is not recommended, and that is when a combination of one or more of the logic approaches explained earlier is applied.

Workload levels

Container orchestrators are primarily driving the scale and density of workloads. That is their main focus. Usually, the orchestrator is blind to the functionality attributed to a container; the only thing that the container understands is to place a new resource request on a node that has the most available resources at any given moment, which contravene the logic expressed in the previous section.

Sensitivity level is the process of labelling container according to the functionality or purposed destination they have to offer. Orchestrator should be configured to take advantage of the sensitivity level right from the beginning. The general model is to define rules that block high sensitivity workloads from being deployed on the same host as those running lower sensitivity workloads. Segmenting containers by purpose, sensitivity, and threat posture provides additional defense-in-depth.

To increase the security posture, an interesting tool comes to help: **Kube-Scan**. Kube-Scan is a Kubernetes risk assessment tool created by OctarineSec with the purpose of scanning the workloads that are currently running on the cluster, showing the risk score and the risk details in a user-friendly web UI. The risk score lies between 0 and 10. The tools is still free available under the **Massachusetts Institute of Technology** (**MIT**) License on GitHub, but the company was later acquired by VMWare as part of the VMware Carbon Black Cloud team, a cybersecurity company focused on developing cloud-native endpoint security applications.

A best practice is to group containers by relative sensitivity and ensure that a given host only runs containers of a given sensitivity level. This can be achieved by using multiple physical servers or by deploying more than one container orchestrator, each of them created for serving a specific sensitivity level. An attacker compromising a single host would have limited capabilities to perform reconnaissance on other

https://t.me/learningnets

containers of a similar sensitivity level, achieving network isolation and containment at the same time.

Worker node trust

The trust relationship for nodes in an orchestrated environment is utterly important and represents another risk factor that should be taken into consideration. Nodes should be securely introduced to the cluster, have a persistent identity throughout their life cycle, and should also be able to provide their connectivity states. **Mkit** is an interesting tool that can audit the cluster and the node configuration, and it is available at **https://mkit.darkbit.io**.

An orchestrator should be resilient to the compromise of individual nodes without affecting the overall security of the cluster. A compromised node should be isolated and removed from the cluster seamlessly. Container orchestrators should provide mutually authenticated network connections between cluster members and end-to-end encryption of intra-cluster traffic.

Conclusion

In this chapter, we explored the basic concepts of the risks and vulnerabilities related to containers and Kubernetes. We also analyzed concepts related to the underlying platforms and systems that support container deployment from the security standpoint. This chapter provided an overview of the security concerns related to microservices and the management platform that withstand the deployment processes in a DevSecOps environment.

In the next chapter, we will learn about the security concerns related to the Hardware and Host OS. Although this seems an approach going back in time, readers would be surprised to learn that some of these concerns are directly related to containers security and orchestrators and cloud solutions.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

https://t.me/learningnets

https://t.me/learningnets

Chapter 2

Hardware and Host OS Security

Introduction

The **National Institute of Standard and Technology** (**NIST**) has defined an interesting concept: "**The Container Technology Stack**". This stack aims to define the functional layers involved in the deployment of a container platform. Refer to the following figure:

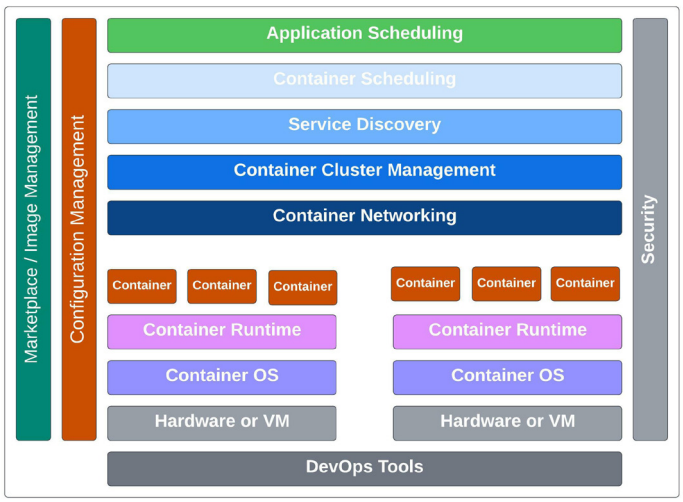

***Figure 2.1**: NIST Container Technology Stack*

https://t.me/learningnets

As illustrated in *Figure 2.1, the Security Layer* (right side of the picture) spreads across the technological layers of the Container Technology Stack, including the Physical Host and Container OS; indeed, no technological layer is excluded from the Security Layer.

This kind of approach helps reconcile the reason for which security is applicable not only to the Host OS Layer but also to the Hardware Layer, even when considering Microservices technologies like the Container Stack. The readers may argue that this concept is difficult to apply to Cloud environments, but in consideration of the **Shared Responsibility Model** (refer to *Chapter 10, Kubernetes Cloud Security*), the Public Cloud Provider is responsible for the entire security related to the Hardware stack and the Virtualization Environment of their Infrastructure.

For example, AWS is known to be using XEN Hypervisor as virtualization layer of their infrastructure, so all the security patches related to that environment and the physical layer underneath must be applied in accordance with security best practices.

Structure

In this chapter, we will discuss the following topics:

- Hardware security
    - Secure Boot
    - Virtualization-Based Security
    - TPM
    - Trusted Execution Environment
    - DICE
- Host Hardening
    - Linux Namespace
    - Control Groups
    - Capabilities
    - SELinux
    - AppArmor
    - Seccomp

Objectives

This chapter aims to provide a complete analysis of the most common hardware and operating system security features, and of the implementation of best practices to elevate the security posture of the technological stack from the ground up.

https://t.me/learningnets

Hardware security

The Firmware Supply Chain has significantly evolved over the past 2 years. Nowadays, it is highly unlikely that a chip manufacturer like Intel or AMD develops all the components presents in its products. Intel has recently started going in the opposite direction with its Intel Foundry Service with the Landmark investment in Ohio and spanning across Europe. Nevertheless, many of the components of the big chip makers are still outsourced, including the firmware line.

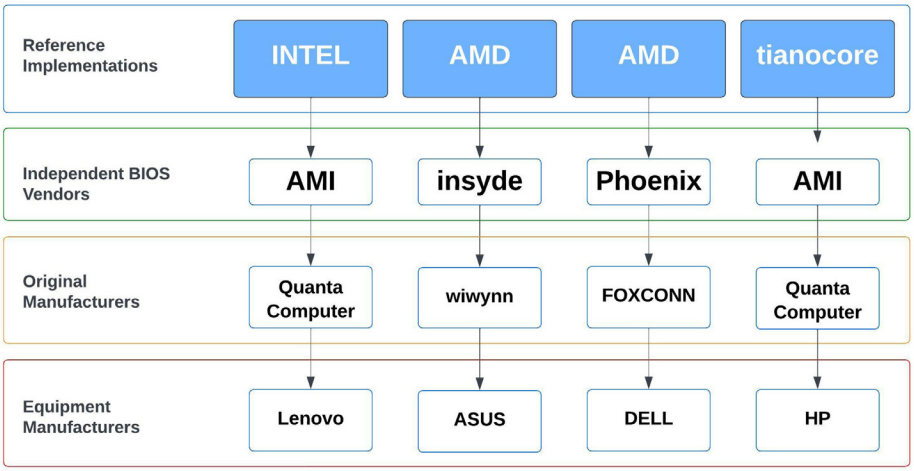

***Figure 2.2**: Firmware Supply Chain*

As more parties are involved in the process, refer to *Figure 2.2, Firmware Supply Chain*, it becomes increasingly difficult to control the firmware development, and therefore, the security related to each component.

The impact of a vulnerability differs based on the business implementation. The BIOS DoS vulnerability **DSA-2021-103**, available at **https://www.dell.com/support/kbdoc/en-us/000187958/dsa-2021-103-dell-emc-poweredge-server-security-update-for-bios-vulnerabilities**, affecting some of the Dell PowerEdge Server firmware, can lead to significant disruption, but it is considered a medium severity.

Nevertheless, if the device is part of an IoT, such as a camera system or a medical device, the disruption can have a much more significant impact. For instance, this was the root cause of the Ukraine Power Grid attack in December 2015, where hackers were able to compromise serial-to-ethernet converter devices to gain access to the Ukrainian's Power Grid systems.

Firmware's main purpose is to interface hardware resources with the Operating System at the first layer, and supplying information about each resources, like

https://t.me/learningnets

CPU, RAM, Hard Drive and so on and so forth. Each of those resources has its own firmware embedded, contributing to expand the Hardware Attack Surface. *Figure 2.3, Boot Sequence*, illustrates the various steps needed to reach a "system up and running" state from the **Power On** phase.

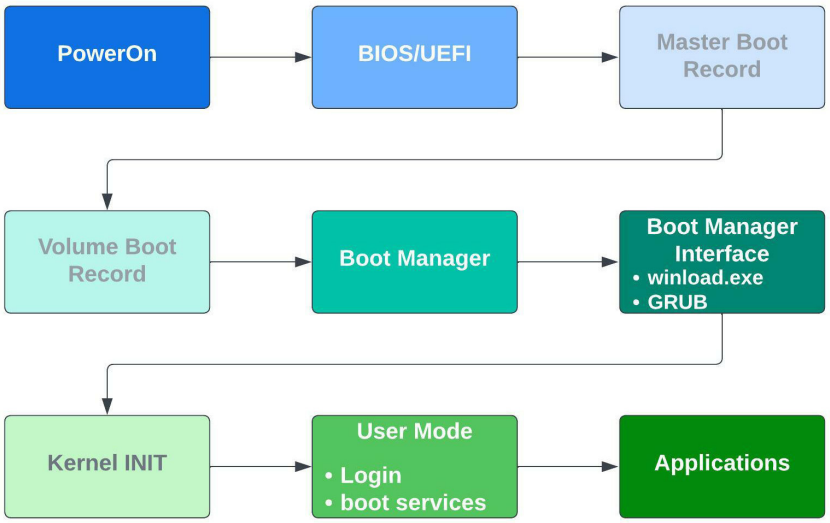

***Figure 2.3**: Boot Sequence*

The most famous type of firmware is the Basic Input Output System, also known as **BIOS**, successively replaced by the **UEFI** solution, which stands for Unified Extensible Firmware Interface. At the beginning of the 2022, **Binarly** discovered several vulnerabilities affecting the **InsydeH2O** UEFI system, one of the most famous firmware hardware brands adopted by firms like Intel, Dell, Lenovo, Microsoft, and HP.

InsydeH2O is an UEFI software developed by the Chinese Insyde Software, a company that is specialized in UEFI system firmware and engineering support services, primarily for **Original Equipment Manufacturers** (**OEM**) and **Original Device Manufacturers** (**ODM**) computer and part device manufacturers. These vulnerabilities are particularly dangerous because UEFI/BIOS-based attacks can bypass firmware-based security mechanisms. Among those attacks, it is worth recalling the "SMM allout" (System Management Mode) or privilege escalation, SMM memory corruption, and DXE (Driver eXecution Environment) memory corruption.

Binarly developed the **FwHunt** application, a rule format system to detect vulnerable code patterns within the UFEI system. The security risk posed by those vulnerabilities is significant because they can be used by attackers to bypass hardware-based security features like secure boot, **virtualization-based security** (**VBS**), and **trusted**

https://t.me/learningnets

platform modules (**TPM**). The National Security Agency has published an extensive Technical Report that has valuable information with regard to UEFI Boot Security. The document is available here:

https://media.defense.gov/2020/Sep/15/2002497594/-1/-1/0/CTR-UEFI-Secure-Boot-Customization-UOO168873-20.PDF

Secure boot

Secure Boot is one the secure features of the Unified Extensible Firmware Interface version 2.3.1 (also known as Errata C). When Secure Boot is enabled and configured, it helps in increasing the security posture of a computer. Secure Boot detects boot loaders alternations, key operating system files, and non-authorized Read Only Memory coding by validating their digital signatures, creating a Trust Boot Architecture. Readers familiar with solutions like private and public key pairs will find the following presentation published on uefi.org interesting: **https://uefi.org/sites/default/files/resources/1_-_UEFI_Summit_Deploying_Secure_Boot_July_2012_0.pdf**

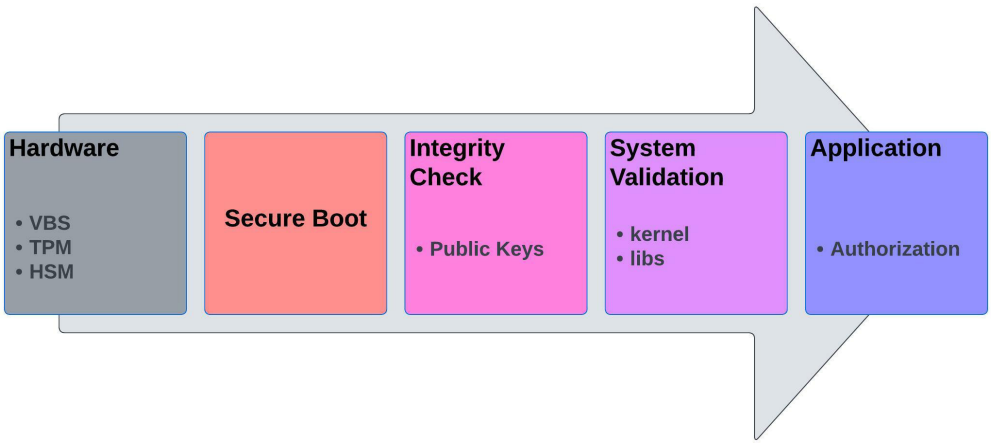

***Figure 2.4**: Chain of Trust*

Figure 2.4, Chain of Trust, illustrates what is known as the Chain of Trust. In essence, when Secure Boot is enabled, the boot process is secured by preventing the loading of boot managers that are not signed with a digital signature. The public key generated by the Secure Boot system, also known as the "platform key", is saved directly in the firmware. The Secure Boot system then enters the "user mode", where only operating system boot loaders signed with the platform key are allowed. Linux distributions and Microsoft have developed different implementations for Secure Boot. Microsoft has, of course, a significant footprint on the Original Equipment

https://t.me/learningnets

Manufacturers market (OEM), and it has requested manufacturers to generate the platform key at the firmware level (effectively creating the first application of the Windows Hardware Quality Labs), but Linux, with less ties to the hardware world, had to find workarounds at the software level: Canonical, for instance, was one of the first Linux distros to introduce a signed bootloader (shim) that checks for vendor-signed known keys. With respect to *Figure 2.3, Boot Sequence*, this practice was working only from the Boot Manager to the User Mode. Later, that evolved in what is today known as **Unified Extensible Firmware Interface Secure Boot**.

The Unified Extensible Firmware Interface Secure Boot ensures that only signed software are loaded at the boot time. It uses digital signatures stored in the Allow DB database, and it preserves a list of revoked digital signatures within the Disallow DB. The digital signature package is updated as part of the firmware update process. In 2020, a new high vulnerability was discovered, **CVE-2020-10713**; it is also known as the BootHole, a buffer overflow vulnerability in **GRUB2**.

As far as this seems anachronistic, readers would note that UEFI Secure Boot is still a security feature claimed in many virtualized environments like VMware ESXi, AWS EC2 Instances, Google Cloud Shielded VMs, and generation 2 "**Trusted Launch**" of Azure Virtual Machines.

Virtualization-based security

Virtualization-based security, or **VBS**, uses CPUs hardware virtualization capability to create and isolate a portion of the memory specifically for the operating system. Refer to the following figure:

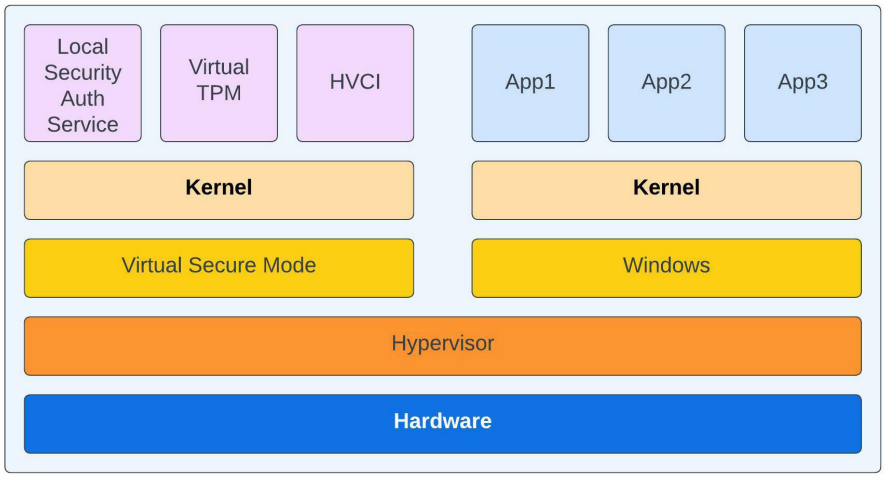

***Figure 2.5**: Hypervisor Code Integrity*

https://t.me/learningnets

The segmentation of the memory is exclusively allocated to the Host OS, and it can use the "virtual secure mode" to load or embed several security solutions, providing them enhanced protection from vulnerabilities and also preventing the use of exploit attempts for bypassing those protections. Virtualization-based security leverages modern CPU hardware capabilities like Intel and AMD with their respective Intel-VT and AMD-V features. The Microsoft hypervisor security solution for virtualization-based security is called HVCI, which is the acronym for Hypervisor-Enforced Code Integrity, commonly referred to as Memory integrity and used to strengthen code integrity policy enforcement. Kernel mode code integrity checks all kernel mode drivers and binaries before they are started and prevents unsigned drivers or system files from being loaded into system memory.

Figure 2.5, Hypervisor Code Integrity, shows how Microsoft implemented VBS in Windows 10 with the introduction of Windows Defender Device Guard and Credential Guard features. Device Guard is a set of three components that prevent untrusted code from running on Windows 10. These features are enforcing only the execution of trusted, signed and verified firmware on the device.

Black Hat has published an interesting analysis of the attack surface of Windows 10 virtualization security based on VBS architecture in 2016. The presenter was an engineer from Bromium, a company later acquired by HP (Hewlett Packard). Bromium “micro-virtualization” technology was designed to protect computers from malicious code execution, such as rogue web links, email attachments and downloaded files.

The link for the material is at **https://www.blackhat.com/docs/us-16/materials/us-16-Wojtczuk-Analysis-Of-The-Attack-Surface-Of-Windows-10-Virtualization-Based-Security-wp.pdf**.

TPM

Trusted Platform Module (**TPM**) is designed to provide security-related functions at the hardware layer. A TPM chip is a secure crypto processor that is designed to execute cryptographic operations. The chip includes multiple physical security mechanisms to make it tamper-resilient. The latest version of TPM is 2.0. It offers device encryption and a few other features not available in the previous version, but also much stronger crypto algorithms like **SHA-256** and ECC P256.

The most common TPM functions are used for system integrity and key management. During the boot process of a system, the boot code that is loaded (including firmware and the operating system components) can be measured and recorded into the TPM chip.

Like any other device or software, TPM has its own life cycle and age. In 2021, researchers from the Dolos Group demonstrated that a TPM platform is hackable

https://t.me/learningnets

with few ad-hoc tools. The Trusted Platform Module communicates with the Central Processing Unit using the serial peripheral interface (SPI), a communications protocol for embedded systems. The serial peripheral interface firmware provides no encryption capabilities, so any encryption must be handled by the devices the Trusted Platform Module is communicating with. By connecting a Saleae logic analyser to the CMOS, the researchers were able to extract every byte moving through the chip, perpetuating an **Evil-Maid attack**, no soldering was required. The last step was using the **bitlocker-spi-toolkit** to isolate the key inside the data extracted by the serial peripheral interface.

The most common TPM practical application is related to Certificates installation or creation. After a computer is provisioned, the **RSA** private key for a certificate is bound to the TPM and cannot be altered. *Figure 2.6, TPM Input Output Logic*, highlights how the coding section in the top-right corner of the image interacts with the TPM platform through the Input/Output logic, and how the TPM platform releases an encryption mechanism to secure the software-hardware interaction.

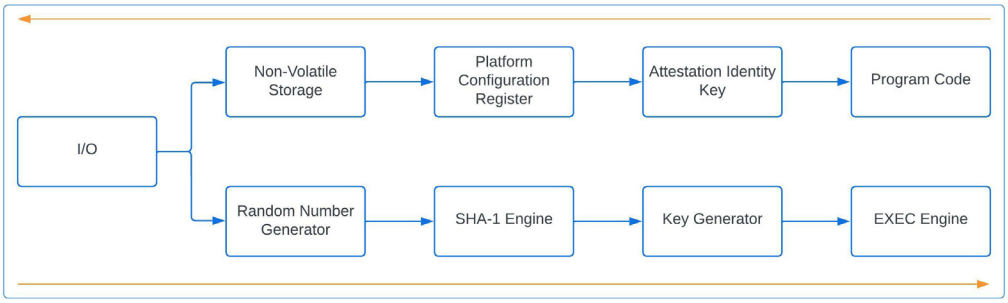

***Figure 2.6**: TPM Input Output Logic*

Anti-malware software can use the boot measurements of the operating system start state to prove the integrity of a computer. These measurements include the use of virtualization platforms like VMWare, XEN, and Hyper-V to test that data centers using virtualized hardware are not running **untrusted** hypervisors.

Of the most common hardware vendors, only Apple has a different approach to hardware security: the **T2 Chip**. This is obviously related to Apple's choice to move away from the Intel platform adopting the Apple Silicon solution, but the T2 Chip has similar capabilities.

When combining technologies together is the only way to overcome platform limitation, such as in the case of Parallel and Apple, **vTPM** (virtual TPM) represents the optimal solution. vTPM is a software abstraction of the hardware TPM Chip, and it allows performing cryptographic coprocessor capabilities within the software layer. That explains why Parallel can run Windows on a MacBook and why VMWare

https://t.me/learningnets

can do the same with the Microsoft operating systems using vSphere rather than Hyper-V.

To enforce trusted security and filesystem encryption, those measurements can be leveraged through the virtualized environment with tools like BitLocker, TrueCrypt or VeraCrypt, and LUKS. Each of those cryptographic solutions has its own pros and cons, but it's sufficient to say that with TPM functionality, it is possible to bind the encryption of a virtualized platform with the underneath physical layer.

BitLocker cannot handle native containers encryption; it needs to encrypt the entire partition on which the containers are running, and it is a Microsoft solution for the Microsoft world, but technically valid. TrueCrypt made history some years ago following an FBI failed 1-year attempt to decrypt five seized hard drives of a Brazilian banker accused of financial crimes. The article is available at **https://g1.globo.com/English/noticia/2010/06/not-even-fbi-can-de-crypt-files-daniel-dantas.html**.

LUKS can communicate with TPM 2.0 natively and can be used in combination with Docker as well, providing an end-to-end solution from the hardware layer up to the container layer.

Finally, the joint effort between Intel and Docker promoting trusted containers, where all the aspects of the container stack, are authenticated through TPM to provide enterprise-level security. The article was published on the Intel website a few years ago and is still available at **https://www.intel.com/content/www/us/en/developer/articles/technical/secure-the-iot-edge-with-trusted-docker-containers.html**.

Trusted execution environment

With the extreme complexity of modern technology, and the increased difficulty of securing complex hardware platforms, a new concept arose a few years ago: the Trusted Execution Environment. The **Trusted Execution Environment** (**TEE**) protects the area of the hardware where the code is executed by applying hardware isolation. To be considered a Trusted Execution Environment, a hardware platform must satisfy the following three requirements:

- **Data Confidentiality**: Unauthorised entities cannot view the data in the TEE.
- **Data Integrity**: Unauthorised entities cannot add, remove, or alter the data in the TEE.
- **Code Integrity**: Unauthorised entities cannot add, remove, or alter code executing in the TEE.

The rise of the trusted execution environment is to satisfy the need to process the new kind of data generated beyond the use of a simple password, like facial scans or fingerprints: biometrics. Let us think of registering the fingerprint on a mobile

https://t.me/learningnets

device. Upon a login attempt the fingerprint provided for identification by the user is verified against the fingerprint stored on the device. Biometric data is generated by what in security terms is called "something you are", and must be stored in a segregated environment although maintaining the needed availability.

Two of the most prominent implementations of the Trusted Execution Environments are *TrustZone* developed by ARM, and SGX developed by Intel, as shown in *Figure 2.7, Intel SGX*. Refer to the following figure:

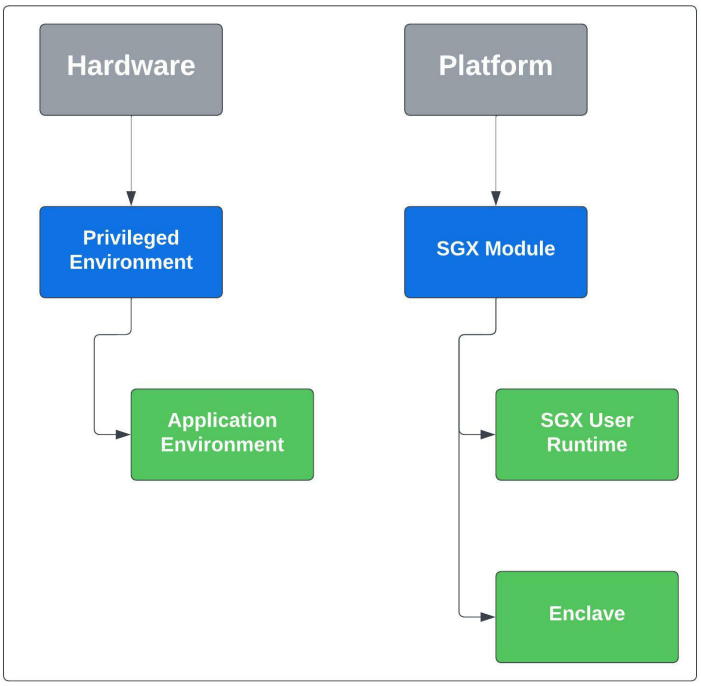

***Figure 2.7**: Intel SGX*

For instance, the popular messaging application Signal uses the Intel Software Guard Extension (SGX). That is a good example of how Intel technology provides a mechanism to apply security all the way from the hardware layer up to the application layer. The interesting fact is that Intel has found a way to containerize a portion of the memory, so-called "enclaves," the data is executed securely because each enclave is verified by a cryptographic attestation key and a hardware root of trust.

DICE

Device Identifier Composition Engine (**DICE**) is a hardware security standard created by the Trusted Computing Group designed specifically for the security of the Internet of Things (IoT) devices, targeting products like Micro Controller Unit

https://t.me/learningnets

(MCU), and systems on chip (SoC), where a TPM hardware device would be too expensive to implement in terms of transistor and power consumption. DICE works by organizing the boot phases (Power ON, Hardware INIT, Disk0 MBR, Boot sector, Bootloader) into logical layers with a unique assigned secret, called the Unique Device Secret (UDS). When a new code is booted, at any point in the boot sequence chain, any logical boot layer will have a different secret. Each layer is isolated from the others, keeping the secret it receives completely confidential. Any change of code in the boot sequence will automatically trigger a secret rotation. System On Chip contains non-volatile memory, which is not a Trusted Platform Module, due to space constraints on the motherboard; it can be used to store cryptographic keys. The risk associated with this approach is that if the code stored in the non-volatile memory is compromised, the secret key could leak. ARM processors, as a workaround, have implemented Trust Zone™ to limit non-volatile memory access to Trusted Execution Environments (TEEs) only.

Host OS Hardening

In computer security, host hardening is the process of reducing the attack surface of system by using two methodologies: removing all the unnecessary packages, services and libraries, and enforcing all the security features a system makes available, as shown in *Figure 2.8, Host Hardening*, aiming to reduce security risks and minimize or remove potential attack vectors.

***Figure 2.8**: Host Hardening*

https://t.me/learningnets

A comprehensive list of all the possible features applicable to any system or application is available on the Center for Internet Security website at **https://www.cisecurity.org**, which includes not only the various Linux distributions like RedHat, Ubuntu, OpenSUSE, CentOS or server system such as Apache, NGINX, MySQL but also hardening guides for entire cloud platforms. The aim of this paragraph is not to exhaustively treat a single aspect of any operating system; that would be impossible. Rather, it is to linger on the most important aspects of a system that directly impact our journey toward the Security for Containers and Kubernetes.

In the DevSecOps world, it is worth mentioning that today it is easy to find ready-to-go scripts or checklists that can provide a baseline to apply hardening principles to a system, but those principles may vary from distro to distro. In this sense, a couple of resources should be considered:

- The DevSec Hardening Framework
- STIG
- Lynis

The **DevSec** Hardening Framework offers various hardening collections with respect to the most up-to-date security frameworks, including the **CIS** mentioned earlier. Automation systems are also considered, such as Ansible, Puppet or Chef, but prominent server systems like Apache or MySQL are not neglected, and of course there's Docker.

STIG stands for Security Technical Implementation Guide; it is a configuration standard of cybersecurity requirements promoted by the Defense Information Systems Agency (DISA), available at **https://public.cyber.mil/stigs**. Readers would appreciate that applying hardware measures at the STIG level would guarantee the maximum layer of security achievable in any given system, but it will also reduce drastically the availability of the same system to the minimum acceptable. Dell is one of the few computer manufacturers offering this kind of hardening solution: **https://www.dell.com/en-us/dt/services/deployment-services/STIG.htm**.

Lynis is a security tool for system running *-nix based operating systems (including macOS), aiming to audit, via an extensive health scan, systems based on the components that are actually installed and configured.

Note: If Lynis detects the httpd daemon running for example, it will perform a series of tests related to web server configurations. If it detects SSL/TLS configuration during the initial tests, it will perform additional auditing steps based on the SSL/TLS configuration, for example collecting any certificates loaded on the server, so that they can also be scanned via additional certificates tests.

https://t.me/learningnets

Lynis is available at **https://cisofy.com/lynis/** and is prone to automation implementation through CI/CD pipelines.

Linux namespaces

Namespaces are a particular feature of the Linux kernel that allows processes to be loaded in a "containerized" environment, meaning that a Linux process can only access compute resources within the confinement of its own namespace; it can't intercept or interfere with another process's resources.

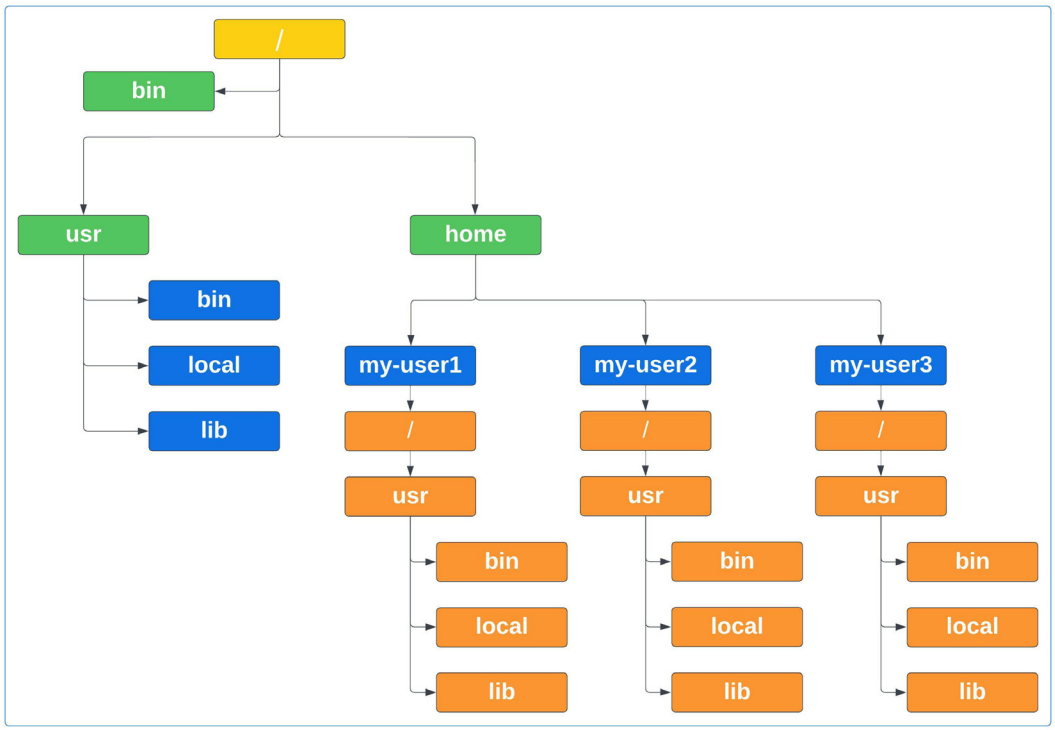

***Figure 2.9**: chroot*

Figure 2.9, chroot, illustrates the *chroot* principle; refer to the *System Level Virtualization* section in *Chapter 1, Containers and Kubernetes Risk Analysis*. The fundamental difference with the namespace mechanism is the Linux kernel isolated kernel processes, identified as **PID**, also known as Process Identification Number. Readers can think of namespace-like kernel-enforced user space views.

As explained in *Figure 2.10, Namespaces*, **PID1** is root and is generated by the actual running process. It assumes the Parent PID characteristic when a PID in the chain, in this case, PID 8 launches sub-processes, then becoming the Child PID. Processes in

https://t.me/learningnets

the Child PID space never interact with the Parent PID, but it is true otherwise, that Parent PID processes can access Child PID processes.

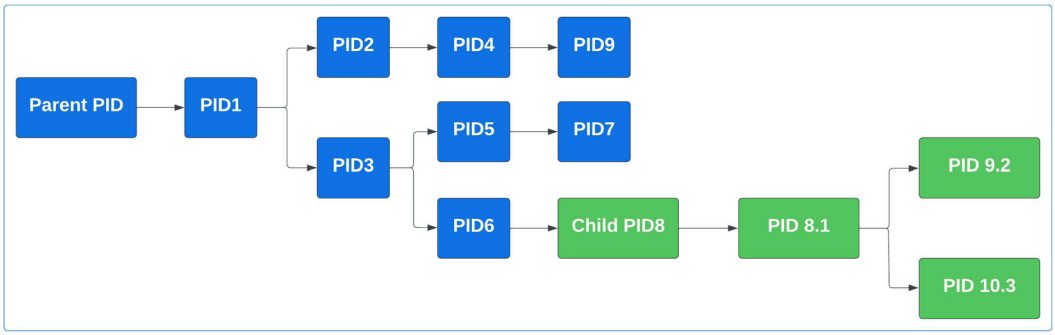

***Figure 2.10**: Namespaces*

There are other namespaces defined in the Linux Kernel, as shown in the *Table 2.1, Namespaces*:

Namespace	Isolates
PID	Processes
Network	Network devices, stacks, ports
User	User and Group IDs
MOUNT	Mount points
IPC	System V IPC
UTS	HOST and NIS domain name
Control Group	Cgroup root directory
Time	Boot and monotonic clocks

***Table 2.1**: Namespaces*

Namespace tools like Docker allow control on the usage of the processes' resources. Example of applications or services using this solution natively are *Heroku* and *Google App Engine* to isolate and run multiple server applications on the same underlying hardware. Linux namespaces are considered fairly secure; they isolate host system resources from independent processes running on top of it, and they are also considered the underlying technology behind container solutions, as they apply the same principle to containers' interaction: a container should not be allowed to obtain control over another container or its resources. In essence, when a new container is created, a specific namespace should be assigned to it.

https://t.me/learningnets

Note: There is no relationship between Kubernetes namespaces and Linux namespaces. Kubernetes namespace is a logical construct to segregate cluster resources, such as pods, between multiple users. You can apply RBAC using Role and RoleBinding, which defines who can do what in any given namespace. Linux namespaces are applied at the HOST layer, and they do not depend on any cluster interaction.

The drawback in systems like Kubernetes is that the orchestrator was originally designed without security tenancy or segmentation, exposing Linux kernel namespaces as shared resources, especially in components like **`kubelet`**, **`etcd`**, **`kube-proxy`** and the *API* server. Therefore, if a malicious actor gains access to the API server, it means the attack vector has already bypassed the Kubernetes Namespace. The later movement from the API server to the **etcd** datastore or to the kube-controller-manager will depend on whether or not Linux Namespaces have been applied correctly. Refer to the following figure:

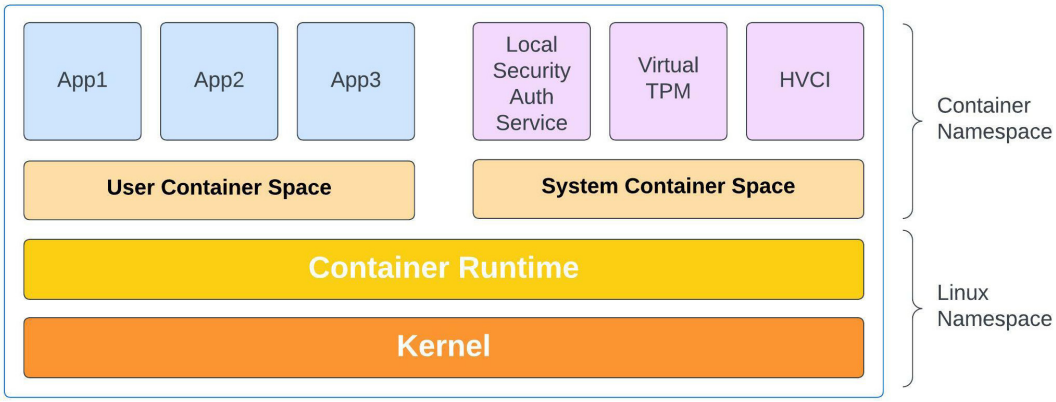

***Figure 2.11**: Linux and Kubernetes namespaces*

Figure 2.11, Linux and Kubernetes namespaces, recalls *Figure 1.12, Container*-specific OS, from the *Chapter 1, Containers and Kubernetes Risk Analysis*, but it highlights the belonging domains of applicability with regard to the namespaces feature.

Control groups

Control groups, abbreviated in Cgroups, are Linux namespaces complementary features. They are the allocation of a specific amount of resources to any given process.

A cgroup enables a Linux process with a mechanism to aggregate a set of tasks, including parent and children tasks, into hierarchical groups for one or more subsystems, as briefly described in *Table 2.2, Cgroups Definitions*. The main purpose

https://t.me/learningnets

of the Cgroups is to limit, isolate and account for the amount of physical resources in terms of central processing unit, memory, disk input and output, and network to any given software process.

Feature	Description
subsystem	It is a module that enables group aggregation, typically used as a resource controller, to apply resource limits to the scheduled resources. It is also acts as a virtualization subsystem.
hierarchy	The cgroups are arranged hierarchically in a parent-children fashion.

***Table 2.2**: Cgroups Definitions*

Table 2.3, Control Groups, shows the four main Control Groups features:

Feature	Description
Resource limiting	Cgroups can be set to not exceed a configured memory limit
Prioritization	Some groups may be released a larger share of CPU utilization or disk I/O throughput
Accounting	Measures a Control group's resource usage
Control	Freezing Cgroups of processes, their checkpointing and restarting

***Table 2.3**: Control Groups*

In Linux distributions, the cgroups is mounted as read-only filesystem via tmpfs, and it is located in the **/sys/fs/cgroup** folder, where the cgroups resources can be invoked. Resources are identified as a set of cpu, cpuset, hugetlb, io, memory, and pid. A typical application of a Control Group can be invoked through the **echo** command as per the following example code, where the amended cgroup task can access only CPUs 2 and 4, and Memory Node 2.

```
1. $ echo 2-4 > cpuset.cpus
2. $ echo 2 > cpuset.mems
3. $ echo $$ > tasks
```

The translation of the preceding methodology can be executed manually through the Docker command when running the container, for example, the following string aims to start the **my-ubuntu** container with access to maximum two CPUs and limiting the memory access up to 1 GB of RAM with a RAM reservation of 256 MB.

https://t.me/learningnets

```
1. $ docker run -it -d --cpus=2 -m 1024m --memory-
   reservation=256m --name my-ubuntu ubuntu:22.04
```

The preceding command can be also invoked in the following example for resources limitation on a Dockerfile for the Apache web server **httpd**. The same principle is applicable to Kubernetes deployments.

```
1.  services:
2.    service:
3.      image: httpd
4.      deploy:
5.          resources:
6.              limits:
7.                cpus: 2.00
8.                memory: 1024M
9.              reservations:
10.               cpus: 0.50
11.               memory: 256M
```

At the beginning of 2022, a **Unit 42** researcher at Palo Alto Network discovered what would later be classified as CVE-2022-0492. The **CVE-2022-0492** is a privilege escalation vulnerability. It highlights a logical bug in control groups because the Linux kernel mistakenly exposes a privileged operation to unprivileged users. The default security hardenings in most container environments are not sufficient to prevent container escape, but for those running with AppArmor, SELinux or Seccomp, there is no risk. The root cause analysis revealed that when a process is terminated, the kernel checks whether the related cgroups had "notify_on_release" enabled, and if so, spawns the configured "release_agent" binary.

```
1. /sys/fs/cgroup/memory/release_agent
```

The release agent runs with the highest possible permissions: a root process with all capabilities in the initial namespaces; as such, the release agent is considered a privileged operation, as it allows you to decide which binary can run with root permissions.

Capabilities

The root user, or any ID with UID of 0 for that matter, has special "powers" when running processes. The kernel and applications are programmed to disregard the restriction related to some activities when processing requests with user ID equal to

https://t.me/learningnets

0. In other words, the root user is allowed to do anything; it is, indeed, also known as the "super user". Applying "**capabilities**", we can strengthen applications and containers security. Unfortunately, this powerful tool is still underutilized.

Any **non-root user** is a non-privileged user. A non-root user can only access data or execute commands owned by itself or the group it belongs to, or when that data is marked for access by all users. The latter is also translated in Linux permissions as 777. When the process related to non-root data requires more permissions, like when opening a network socket, the process needs to escalate privileges and become a temporary privileged user. This is where a command like *sudo* is utilized.

There is a common misconception about the Super User DO (sudo) command. It is considered secure by default, which is not completely true; most of the time, it depends from the level of secure configuration applied to the **/etc/sudoers** file, especially in environments where the *sudo* rules are centralized through Lightweight Directory Access Protocol, also known as LDAP. Also, it is hard to imagine that DevOps team running CI/CD pipelines for EC2 (Elastic Compute Cloud) instances rotation of the web servers fleet are using **sudoers** configured in LDAP. In this scenario, often machine users are implemented, also known as service accounts. These are used to replace a real human interaction with the automation system to facilitate the user access from system to system or application to application. An interesting tool to verify the exploitability of the Linux sudo rules is **FallOfSudo**.

Capabilities finds its way in this scenario: the web server service of the previous example will be able to establish the network socket on the port 80 because a capability like CAP_NET_BIND_SERVICE is set on the related httpd binary file. The web service won't need any human input in granting a higher level of permissions, so **sudo** won't be needed. For a full list of capabilities and their applications, refer to **https://man7.org/linux/man-pages/man7/capabilities.7.html**. Capabilities are a **per-thread** attribute or a **per-file** attribute, and as such, every thread or file has the following capability set, as shown in *Table 2.4 - Capabilities*:

Feature	Thread	File
Permitted Set	Yes	Yes
Inheritable Set	Yes	Yes
Effective Set	Yes	Yes
Bounding Set	Yes	No
Ambient Set	Yes	No

***Table 2.4**: Capabilities*

Container runtimes have some of these capabilities enabled by default, for example, it is possible to check the default capabilities enabled by the CRI-O runtime on its latest version. When running containers with UID 0, default capabilities configured

https://t.me/learningnets

by the runtime will be configured in the effective set for the container thread. When running containers with non-root UID, default capabilities configured by the runtime are dropped. This means that containers running non-root UID, which is the preferred way, will be able to establish a network socket connection only if the related capability is defined. Refer to the following code:

```
1. version: '3'
2. services:
3.   web:
4.     cap_add:
5.     - NET_BIND_SERVICE
```

Similarly, **Kubernetes** has adopted a mechanism that achieves the same goal leveraging the privileged flag on the *Security Context* for PODs. The code for this achievement is as follows:

```
1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   name: security-context-cap-admin
5. spec:
6.   containers:
7.   - name: sec-ctx-cap-admin
8.     image: gcr.io/google-samples/node-hello:1.0
9.     securityContext:
10.      capabilities:
11.        add: ["NET_ADMIN", "SYS_TIME"]
```

Security Enhanced Linux

Security Enhanced Linux is a security enhancement to Linux originally developed by the United States **National Security Agency** (**NSA**), which allows users and administrators more granular control over access control. **SELinux** defines access

https://t.me/learningnets

controls for the applications, processes, and files on a system. It uses security policies, which are a set of rules that instruct SELinux on what can or can't be accessed.

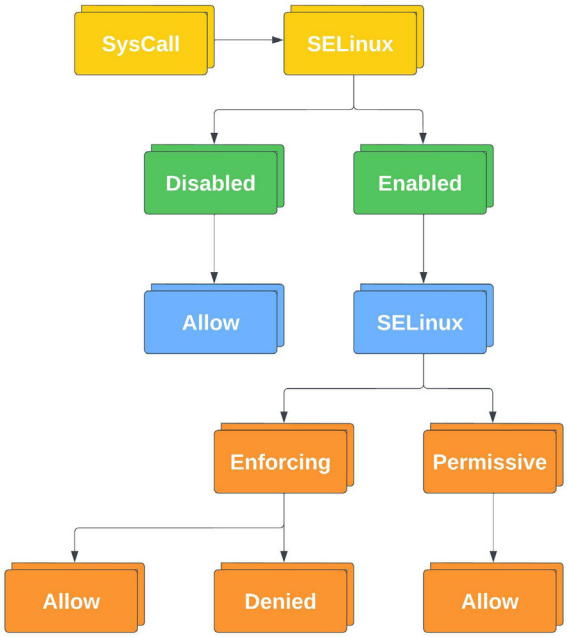

***Figure 2.12**: SELinux*

When an application or process makes a request to access an object, SELinux sends the request to the security server, the security server checks for the security context of the app or process and the file. Security context is applied from the SELinux policy database. If the permissions are cached for subjects and objects, SELinux checks with the **Access Vector Cache** (**AVC**) and behaves accordingly. Permission is then granted or denied. *Figure 2.12, SELinux*, illustrates a typical policy check scenario.

Traditionally, *-nix systems have used **Discretionary Access Control** (**DAC**), where files and processes have owners. The root user has full access control with a DAC system. SELinux has a different approach to security; it uses a **Mandatory Access Control** (**MAC**) methodology where there is an administrative policy set around access.

Let's go back to the web server example in the previous paragraph, where the webserver, thanks to the Linux Capabilities, was able to establish a network socket on port 80. In this case, through the SELinux policy system, the web server will be able to serve web pages because it has read permissions on the document root. However, SELinux will block any write attempt to any directory other than the one the SELinux policy allows **Secure File Transfer Protocol** (**SFTP**) connections to.

https://t.me/learningnets

In 2019, a flaw in ***runc*** (CVE-2019-5736) allowed container processes to "**escape**" the container layer and execute process on the operating system itself. The attacker's executable was able to overwrite the *runc* command on the operating system, giving the attacker access to the underling host, in essence "tampering" the **/proc/self/exe** file that was pointing to **`runc`**, assuming containers running as root. Container processes are executed as type **`container_t`**. SELinux policy states that only files of **`container_t`** types can have write permissions and must be labelled as **`container_file_t`**. The default file label in SELinux of **`runc`** is **`container_runtime_exec_t`**, so **`container_t`** types have forbidden writing permissions.

The same principle is applicable to Kubernetes, and the same vulnerability was documented in the Kubernetes Blog at **https://kubernetes.io/blog/2019/02/11/runc-and-cve-2019-5736/**. SELinux can be enabled into Kubernetes with security features **seLinuxOptions** declared in the **securityContext** section of the manifest file, under the POD configuration, as the following code illustrates:

```
1. ...
2. securityContext:
3.   seLinuxOptions:
4.     level: "{SeLinuxOptions}"
```

Historically, SELinux has been addressed as a complex system, and it still feels the same, where the configuration burden is too heavy to be satisfied. Interestingly, solutions like **UDICA**, an SELinux policy generator for Containers, are more efficient in virtualized environments.

AppArmor

AppArmor is a Linux application security system designed to protect operating systems and applications from a platform's external or internal threats, including **zero-day attacks**. It replaces the user- and group-based permissions, classic of the Linux system, to enclose programs to a limited set of resources; indeed, AppArmor security model is to bind access control attributes to programs rather than to users. AppArmor confinement is provided via profiles loaded into the kernel, typically on boot. As shown in *Table 2.5, AppArmor Modes* operates in two types:

Mode	Description
Enforced	In enforce mode, AppArmor enforces the rules and logs the violation attempts in **syslog** or **auditd**, blocking any process that is not matching the rules.
Complain	In complain mode, AppArmor does not enforce the rules. It will only log any security event detected as a violation.

***Table 2.5**: AppArmor Modes*

https://t.me/learningnets

It is possible to check the AppArmor system status by executing in the console the following command:

```
$ sudo apparmor_status
apparmor module is loaded.
6 profiles are loaded.
6 profiles are in enforce mode.
   /sbin/dhclient
   /usr/lib/NetworkManager/nm-dhcp-client.action
   /usr/sbin/httpd
   /usr/sbin/mysqld
   /usr/sbin/tcpdump
0 profiles are in complain mode.
3 processes have profiles defined.
3 processes are in enforce mode.
   /sbin/dhclient (352)
   /usr/sbin/httpd (842)
   /usr/sbin/mysqld (499)
0 processes are in complain mode.
0 processes are unconfined but have a profile defined.
```

When applying AppArmor policies to containers, an AppArmor security profile must be associated with each program. Docker expects to find an AppArmor policy loaded and enforced, and it automatically generates and loads a default profile for containers named docker-default. An example of AppArmor profile for MySQL is as follows:

```
# vim:syntax=apparmor
#include <tunables/global>

/usr/sbin/mysqld {
  #include <abstractions/base>
  #include <abstractions/nameservice>
  #include <abstractions/user-tmp>
  #include <abstractions/mysql>
```

https://t.me/learningnets

```
9.    #include <abstractions/winbind>
10.
11.# Allow system resource access
12.   /sys/devices/system/cpu/ r,
13.   capability sys_resource,
14.   capability dac_override,
15.   capability setuid,
16.   capability setgid,
17.
18.# Allow network access
19.   network tcp,
20.
21.   /etc/hosts.allow r,
22.   /etc/hosts.deny r,
23.
24.# Allow config access
25.   /etc/mysql/** r,
26.
27.# Allow pid, socket, socket lock file access
28.   /var/run/mysqld/mysqld.pid rw,
29.   /var/run/mysqld/mysqld.sock rw,
30.   /var/run/mysqld/mysqld.sock.lock rw,
31.   /run/mysqld/mysqld.pid rw,
32.   /run/mysqld/mysqld.sock rw,
33.   /run/mysqld/mysqld.sock.lock rw,
34.
35.# Allow execution of server binary
36.   /usr/sbin/mysqld mr,
37.   /usr/sbin/mysqld-debug mr,
38.}
```

https://t.me/learningnets

AppArmor is supported by all common Kubernetes container runtimes, including Docker, CRI-O and containerd. AppArmor will be defined in the Kubernetes Pod manifest section by specifying an AppArmor profile that any given container should be run with. In order to apply the AppArmor profile, the Pod will need an annotation metadata, such as the following:

```
1. container.apparmor.security.beta.kubernetes.io/<container_
   name>: <profile_ref>
```

The whole process to manage AppArmor profiles can be cumbersome and time consuming. Although AppArmor provides a set of pre-defined profiles for the most popular applications or services, there would be the need, according to the specific environment, to create, edit or manage multiple profiles. This is the scenario where tools like Kube-apparmor-manager is appreciated.

With **Kube-apparmor-manager**, the AppArmor profile becomes a Kubernetes object that can be saved into the **etcd** database, synchronized between nodes, and therefore, distributed at scale. We will look at some of these methodologies in *Chapter 7, Kubernetes Hardening*.

Seccomp

Secure Computing Mode is a Linux kernel feature aiming to restrict the *syscall* a Linux process is allowed to do. *Figure 2.13, SECCOMP Logic*, shows the Secure Computing Mode logic:

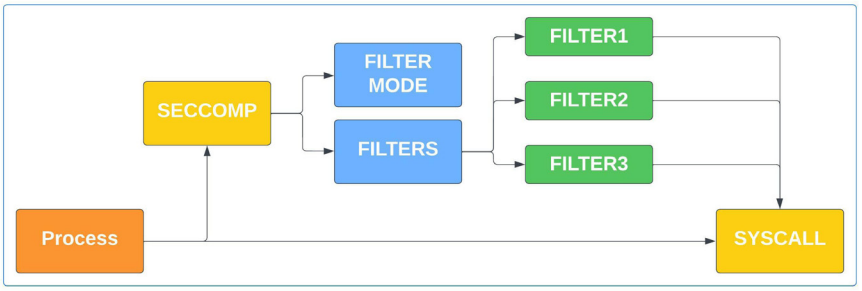

***Figure 2.13**: SECCOMP Logic*

The kernel is essentially sandboxing a process when seccomp is enabled, limiting the system calls only to **`exit()`**, **`read()`**, **`sigreturn()`**, and **`write()`**. Many software projects, such as Android, Chrome, and Firefox, use seccomp to tighten security further.

The concept behind **SECCOMP** is simple: the user space processes do not need to be exposed to all the system calls available to the kernel; most often, such processes will use a limited number of those calls. It is dangerous, anyway, to leave the door

https://t.me/learningnets

open for those processes to eventually be able to make system calls, because they can become an attack vector for malicious code. If a process is compromised, it could run syscalls that usually are not executed in a normal execution mode, such as **`execve()`**.

At the beginning of 2022, the Deepwatch Threat Intel Team published CVE-2022-20699, affecting Cisco RV340/RV345 series SSL VPN devices. The shellcode uses "**`execve()`**" to execute **/bin/sh**. It allows the attacker to specify host and port to establish interactive communication with the compromised device via a reverse shell.

Figure 2.14, SECCOMP Container Workflow, illustrates the process chain needed to reach the kernel syscall. The default Docker profile is usually sufficient to cover most common scenarios, but a custom profile option is available. Kubernetes integrates SECCOMP as a measure of secure computing for the cluster environment. The SECCOMP profiles can be applied either at the node level or pod level and distributed to the cluster. Refer to the following figure:

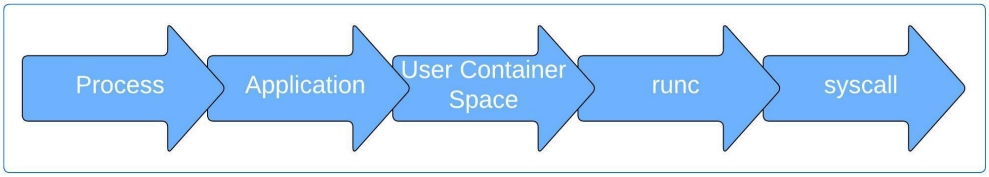

***Figure 2.14**: SECCOMP Container Workflow*

Kubernetes also makes available observability processes to audit syscall made by a specific container, and it detects behaviour when a missing profile is specified or a violation of a SECCOMP profile.

```
1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   name: audit-pod
5.   labels:
6.     app: audit-pod
7. spec:
8.   securityContext:
9.     seccompProfile:
10.      type: Localhost
11.      localhostProfile: profiles/audit.json
12.  containers:
```

https://t.me/learningnets

```
13.   - name: test-container
14.     image: hashicorp/http-echo:0.2.3
15.     args:
16.     - "-text=just made some syscalls!"
17.     securityContext:
18.       allowPrivilegeEscalation: false
```

The auditing process is likely one of the most interesting ones; Kubernetes will create an "audit" POD via the **audit.json** manifest mentioned earlier. Kubernetes Security Profile Operator is likely the most active project in the community for promoting security for the orchestrator platform. The project promotes the application of three of the main security principles discussed in this chapter: SECCOMP, SELinux and AppArmor.

Conclusion

In this chapter, we acknowledged that even in the age of the Cloud Native applications, hardware and operating systems are still active part of the security landscape, and they can help significantly to reduce the attack surface due to the deep integration and interaction between all the layers of the Container Stack. We also analyzed the main security hardware technologies and their capability to bind the upper-level software security layer. Moving on, we discussed the operating system security features that can be leveraged by the container and the orchestrator layers.

In the next chapter, we will learn about Container Stack Security and the best practices to secure a container environment.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

https://t.me/learningnets

Chapter 3
Container Stack Security

Introduction

There was a time when speaking of containers would have meant speaking of **Docker**. It's the same today, but the container stack evolution has gone very far since the rise of the Cloud Native Computing Foundation. The biggest sign of such a change was announced by the **Kubernetes** version 1.20, the so called Raddest Release in December 2020, where the *dockershim* component – what Kubernetes calls a "special direct integration" – was removed starting from the Kubernetes release v1.24.

From Kubernetes v1.25, *dockershim* is not the default installed container runtime anymore, and it is necessary to install a runtime that will interface with the **Container Runtime Interface** (**CRI**). The change is significant because Kubernetes v1.25 expects that the container runtime installed on the nodes will support the container runtime interface, with a minimum version of **v1alpha2**. It makes sense then, within the scope of the book, to consider the container runtimes supported by Kubernetes at the time of writing, and the necessary steps to elevate such important components of the containerized stack to a sufficient level of security:

- Containerd
- CRI-O

https://t.me/learningnets

- Docker Engine
- Mirantis Container Runtime

A quick note about Mirantis for the readers less familiar with commercially available software, and why it could be a valid choice for some businesses. Mirantis is a commercial container runtime that was previously known as **Docker Enterprise Edition**. The **Mirantis Container Runtime** (**MCR**) can interface with Kubernetes via the open-source component *cri-dockerd*, which is also used by the Docker Engine. Readers can think of the Mirantis Container Runtime as the Docker Engine's commercial version.

Structure

In this chapter, we will discuss the following topics:

- Container security
    - Containerd
    - CRI-O
    - Docker
        - Least Privilege
        - Resource Limitation
        - Container Isolation
- Network Security
    - Mirantis Container Runtime
    - An interesting exclusion
- Secure connection
    - Server Certificate
    - Client Certificate
    - Enable dockerd TLS
    - Secure CI/CD
- Update life cycle

Objectives

This chapter aims to provide a complete overview for the most common container services security best practices, including the Docker Engine. Despite being

https://t.me/learningnets

deprecated in the Kubernetes v1.24, Docker Engine is still a prominent container runtime and the most wildly used container runtime service.

Container security

Many names in a container stack have similar nomenclature and meaning. The vast world of containers can be challenging and confusing. Applying security best practices at the appropriate layer of the container stack is vital to protect the container ecosystem. Thinking of a container like a standalone process is a good start, but the real challenge is to address a containerization process as the child process of a chain of interconnected Linux sub-services. Abstracting physical resources like CPU, RAM, Storage, and Network into virtualized resources that the container can understand and process as its own requires indeed multiple intermediate processing. Kubernetes has introduced an additional layer of complexity, where distinguishing between the container system and the orchestrator system has become even more difficult.

As discussed in *Chapter 2, Hardware and Host OS Security*, a container platform is not immune to hardware security issues or operating systems security issues, but while the physical layer is only the start of the security journey, the abstraction of the physical resources into virtualized or containerized resources brings far more complexity into applying safeguards and security best practices to the container stack. Refer to the following figure:

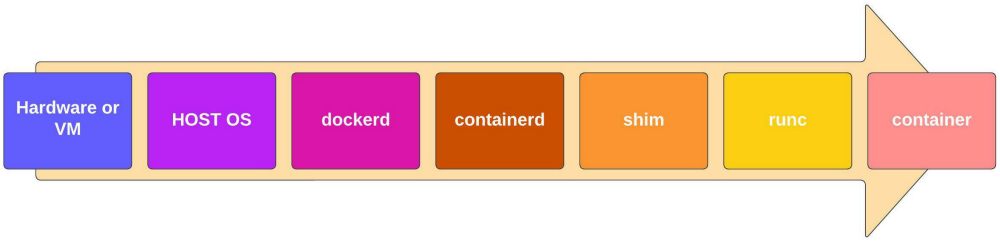

***Figure 3.1**: Docker Container Processes*

To demonstrate such complexity, *Figure 3.1, Docker Container Processes*, illustrates the sequence of elements involved in the Docker containerization process, and also illustrates, from left to right, how a container is created, including all the intermediate

https://t.me/learningnets

required steps, like a sequence of folding processes. Let's abstract *Figure 3.1, Docker Container Processes*, in the following image:

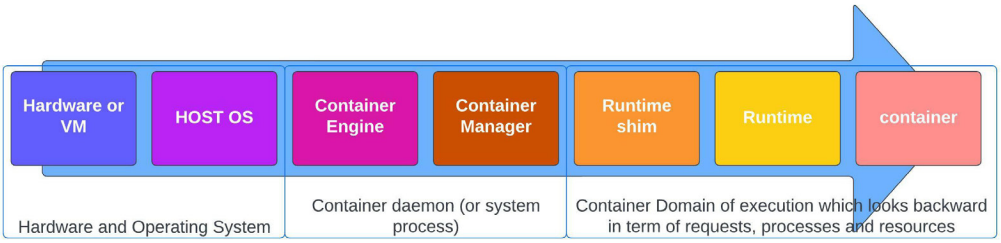

***Figure 3.2**: Abstract Container Processes*

Figure 3.2, Abstract Container Processes, shows the containerization from a different angle: the one that is applicable as a general concept to any other container platform. As readers might remember, in *Chapter 2, Hardware and Host OS Security, Figure 2.1, NIST Container Technology Stack*, the National Institute of Standard and Technology does not refer specifically to Docker, it is indeed referring to any container technology from the security standpoint. *Figure 3.3, Description of Container Processes*, aims to describe each single component involved in the containerization process. **Container Engines** execute virtualization at the operating system layer providing a "control center" environment for running applications and their dependencies. It is the first level in the chain of Linux processes that constitutes a container platform. Refer to the following figure:

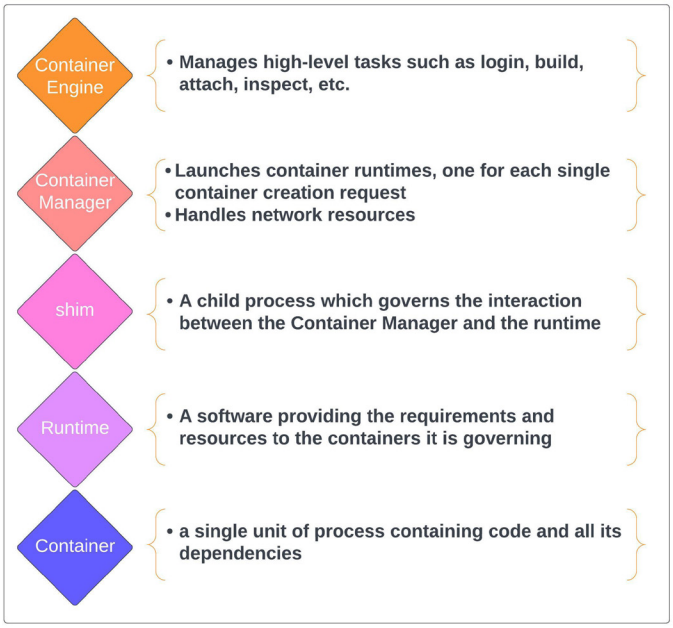

***Figure 3.3**: Description of Container Processes*

https://t.me/learningnets

Container engines usually run as a Linux daemon on the host platform, offering various interaction methodologies like a command line interface or an API system. **Container Managers** are components of a container platform sitting behind a container engine. They receive and interpret the parameters used to manage the containers' life cycle, keeping track of the container status. The Container Manager creates, destroys, starts, and stops containers at a higher level; it manages the interaction with the registry pulling or pushing images, and it also manages the network communications. The Docker platform is the oldest and more complex container system on the market, with several moving parts, in contrast with younger technologies that have, in general, a much smaller footprint and the tendency to incorporate components. Systems such as **containerd** or **CRI-O** have done a remarkable job reducing their computing footprint, and **containerd** has been able also to incorporate an API system. *Figure 3.4, Container Engines and Managers*, provides a quick distinction of the most common platform in their respective sections. As explained previously, some systems can assume more than a single function.

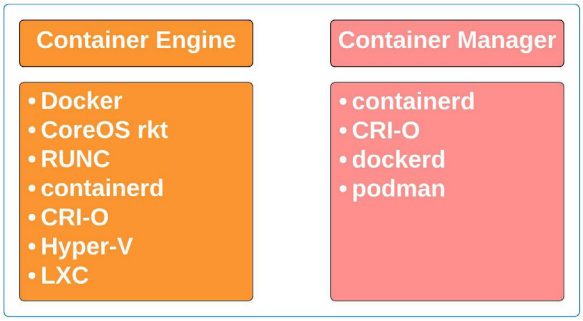

***Figure 3.4**: Container Engines and Managers*

Figure 3.5, Comparing Container Runtimes, aims to explain the container processing among the Container Runtimes supported by Kubernetes today. The Mirantis Container Runtime has been purposefully excluded because it is based on the Docker Enterprise Engine; therefore, it has a container processing mechanism like the Dockershim container runtime.

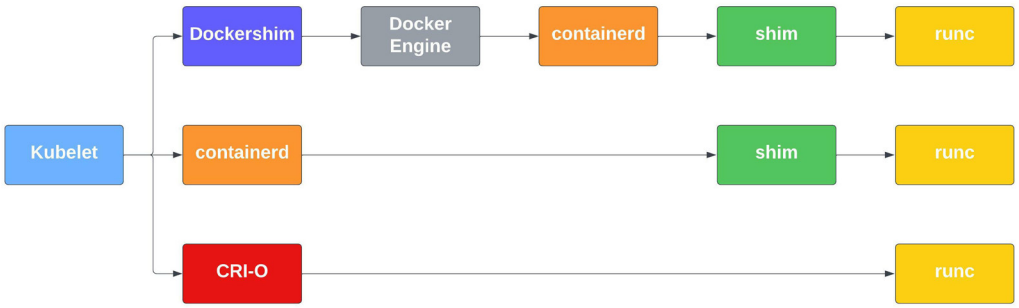

***Figure 3.5**: Comparing Container Runtimes*

https://t.me/learningnets

The distinction between the three contenders is visible at eye, and those differences are also a sign of the complexity around each of the project, and consequently, the need to apply security is proportionated to the extension of the attack surface. The more systems or parts of a system are exposed, the more layers of security would need to be applied.

As an interesting anomaly, the **CoreOS rkt** (pronounced "rocket") project development has been halted at the beginning of the 2020, 2 years after RedHat acquisition and announcement of the transition of the overall platform in what is now known as the **Fedora CoreOS** project. It is interesting to note that **rkt** was developed with the principle of secure-by-default in mind, including a series of remarkable security features like SELinux, TPM (refer to the *TPM* section in *Chapter 2, Hardware and Host OS Security*) and hardware-isolated containers from the underlying operating system using a namespaced POD concept similar to the one in Kubernetes.

This is in contrast with the common assumption that containers are running on "shared" resources consumed through the host on which they are running. The rocket project is a non-common containerization model, and curious readers will find the hardware-isolated container concepts interesting. To achieve container isolation, CoreOS rkt uses the **systemd-nspawn** container runtime feature, a layer of the systemd software suite developed by Red Hat. By bringing the containerization mechanism into systemd, CoreOS rkt provides configuration consistency across the various flavors of the Linux distributions.

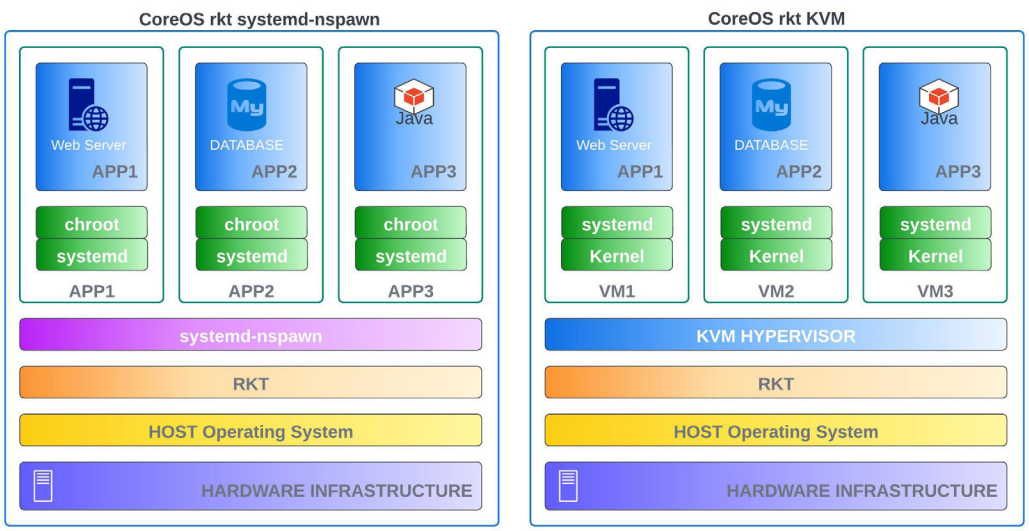

***Figure 3.6**: CoreOS rkt Models*

https://t.me/learningnets

The systemd-nspawn is conceptually similar to chroot but much more powerful because it can achieve container isolation through namespaces, implementing full virtualization of the process tree, including filesystem and user IDs. Interestingly, CoreOS rkt can also implement virtualization through a full **KVM** hypervisor system like QEMU or LKVM, on top of which it injects application running containers in a full Virtual Machine. The security advantage of this approach is that the hypervisor will create a separate kernel instance for each virtual machine, eliminating the classic shared kernel security issue of modern container platforms, as illustrated in *Figure 3.6, CoreOS rkt Models*:

Containerd

Containerd was initially developed by Docker as a subset of its original Docker Engine, and then it was donated to the **Cloud Native Computing Foundation** (**CNCF**) for the purpose of supporting the standards established by the **Open Container Initiative** (**OCI**). Readers willing to dig into the OCI project can use this URL: **https://opencontainers.org.** The technical goal initially pursued by the Docker's developers was abstracting system calls or OS-specific functionalities. When the Cloud Native Computing Foundation graduated in the early 2019, **containerd** moved from being a component of the Docker ecosystem to a full standalone container runtime process.

It is important to note that, in principle, **containerd** share a very similar attack surface with Docker. It does not offer all the refined tools and features that its parent project has, but still, users with access to the socket file are able to download *crictl* or *nerdctl* commands that can be used as debugging tools to gather information about the environment or *curl* the socket to accomplish any desired actions, including malicious or otherwise. For a deep look at the security requirements needed, you can refer to *Chapter 7, Kubernetes Hardening*. At the time of writing **containerd** is known to be used in systems like **Google Kubernetes Engine** (**GKE**), IBM Kubernetes Service, and Alibaba Cloud.

CRI-O

As the name suggests, Container Runtime Interface for Open Container Initiative, abbreviated to CRI-O, has adopted the security requirement provided by the Open Container Initiative framework. The security specification that CRI-O has completely adopted via the OCI framework are partially recalled in the *Host Hardening* section of *Chapter 2, Hardware and Host OS Security*. For a better understanding of those requirements and a more detailed description of what has not been addressed or has been partially addressed, refer to *Table 3.1, OCI Requirements*:

https://t.me/learningnets

OCI Requirement	Description
Namespaces	Allows processes isolation
User Namespace Mapping	Adds to the previous feature mapping of the user UID from the host to the container
Devices	List of devices that MUST be available in the container
Default Devices	Default devices list provided to the container from the host
Control Groups	Allocation of a specific number of resources to any given process
Cgroups Path	Control the Control Groups hierarchy inside the container
Control Groups Ownership	Used to control the Cgroups delegation and forbid change of the ownership for a container cgroups
Device List	List of allowed devices inherited by the container from the host
Memory	Enables Cgroups to set the memory usage limits for the container
CPU	Like the Memory Requirement but for CPU
BLOCK IO	Implements the Cgroups block input and output controller
Huge Page Limits	Implements the Cgroups controller to limit HugeTLB
Network	Implements the Cgroups subsystem for Network Classifier
PIDs	Implements the Cgroups subsystem for Process Number Controller
RDMA	Limits specific resources that a given process can use
Unified	Allows version 2 of Cgroups to be set for the container
IntelRdt	Container runtime writes the container process ID into a task file
Sysctl	Runtime can modify container kernel parameters
Seccomp	Process syscall restrictions
Process State	Container must notify its state
Rootfs Mount	Sets the status of the rootfs mount propagation to containers
Masked Paths	Masks paths within the container
Read-only Paths	Sets paths within the container to read-only
Mount Label	Implements SELinux for the mount points in the container
Personality	Implements process execution domain

***Table 3.1**: OCI Requirements*

For an exhaustive description of the previous characteristic, remember that the Control Groups are Linux Kernel features, information from the Linux documentation is available at **https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1**, and their implementation has been described in *Chapter 2, Hardware and Host OS Security*. In this chapter, Cgroups are discussed in the *Resource Limitation* section.

https://t.me/learningnets

CRI-O container runtime is a new development; it was explicitly built to talk with Kubernetes and has limited use on its own. The logic behind CRI-O is to remove all the interdependency processes between the *kubelet* system and the *runc* system, achieving a lightweight, efficient and full OCI implementation of Kubernetes Container Runtime. At the time of writing this book, CRI-O is known to be used in systems like RedHat OpenShift and its various shapes and forms, either on Microsoft Azure or IBM Cloud, and in SUSE Container as a Service.

Docker

There are limitless resources on the web that can help in reducing Docker's attack surface. Many have also become a set of standards, such the **CIS Docker Benchmark**, but many of these sets are sometimes just a checklist or a mix of different parts of the container platform.

The focus of this section is to identify the most common security features any Docker system should have and provide the rationale behind the choice of adopting that solution or characteristic. Topics of a larger footprint like secure containers connection or updating container life cycle are discussed later in this chapter.

To start this section, you should take a look at **Docker Bench for Security**, a script that checks for many Docker security best practices. The script represents a good starting point in understanding the initial security posture of the Docker deployment. It is based on the CIS Docker Benchmark, version 1.4.0 at the time of writing.

Least privilege

Docker daemon provides a non-root user mode, also known as the "rootless mode". The rootless mode does not need root privileges, neither to run Docker daemons nor to run containers. This is a very important feature that should always be considered. It mitigates any vulnerability in the Docker daemon and within the container runtime processes. The main principle is that the rootless mode executes any Docker process inside the **user namespace**.

For the rootless mode to work, the host must have installed the **uidmap** package. Also, the user should have at least 65,536 subordinates for UID and GID. It is possible to verify this by inspecting **/etc/subuid** and **/etc/subgid**. To verify that the user has those requirements, take a look at the following code:

```
1. $ id -u
2. 1000
3. $ whoami
4. dockeruser
```

https://t.me/learningnets

```
$ grep ^$(whoami): /etc/subuid
dockeruser:231072:65536
$ grep ^$(whoami): /etc/subgid
dockeruser:231072:65536
```

Docker rootless mode comes with the **docker-ce-rootless-extras** package in any major Linux distro repository. To install the rootless package, run the following command from a non-root user shell.

```
$ /usr/bin/dockerd-rootless-setuptool.sh install
[INFO] Creating /home/dockeruser/.config/systemd/user/docker.service
…
[INFO]Installed docker.service successfully
[INFO] To control docker.service, run: `systemctl --user
(start|stop|restart) docker.service`
[INFO] To run docker.service on system startup, run: `sudo loginctl
enable-linger dockeruser`
```

The following environment variables must be set:

```
$ export PATH=/usr/bin:$PATH
$ export DOCKER_HOST=unix:///run/user/1000/docker.sock
```

To enable **`docker.service`** on startup and run the Docker daemon when the system starts:

```
$ systemctl --user enable docker
$ sudo loginctl enable-linger dockeruser
```

The major limitations of the rootless mode are as follows:

- AppArmor is not supported
- Overlay network is not supported
- Cgroups is supported only with version 2 and systemd
- Docker inspect has visibility inside RootlessKit's network namespace

When the rootless mode is enabled, the containers that the dockerd daemon is running should run with rootless privileges. To verify a container privilege mode, run the following:

https://t.me/learningnets

```
1. $ docker container inspect \
2.    --format='{{.HostConfig.Privileged}}' \
3.    [container_id]
```

If it returns "true", the container is having privileged permissions, otherwise it will return "false", confirming that the container is not running in privileged mode. Containers running with root privileges are a serious risk to any container platform, allowing attackers to escalate privileges when the container is compromised. As a security measure, Docker has defaulted containers to unprivileged.

Option	Description
--cap-add	Add Linux capabilities
--cap-drop	Drop Linux capabilities
--privileged	Give extended privileges to this container
--device=[]	Allows to run devices inside the container without the --privileged flag

***Table 3.2**: Docker Runtime Privilege*

Table 3.2, Docker Runtime Privilege, provides the four options available to manage privileged containers. When the **docker run** command has the "--privileged" flag, the container will have access to all the devices on the host plus the set of configurations in AppArmor and SELinux.

Resource limitation

As per Docker design, containers have no limitations on the hardware resources they can use; that's why supporting the Linux Cgroups features is essential. Linux Cgroups have been discussed in *Chapter 2, Hardware and Host OS Security*. The reason behind this choice is obvious; the container does not know which service or application is going to run, so it won't be able to predict the quantity of resources to be allocated for that specific task. This kind of intelligent mechanism is achievable only in complex environments using **Artificial Intelligence** (**AI**) or with advanced governance methodologies.

If a container is compromised, the attacker can use all the hardware resources the container engine can provide; if the container escape is successful, the attacker could get access to the underlying host resources, so it could bring lateral movements and the compromise of the network. It is a good practice to set a resource quota for containers to put resources constraints in place where those are missing by design. Many of these features are directly related to the Linux kernel capabilities; to verify that the host is not supporting those, the **docker info** command will reveal it, issuing a warning: "WARNING: No swap limit support".

https://t.me/learningnets

Option	Description
--memory	max memory allocated to a container
--memory-swap	max memory a container can swap to disk
-- memory-swappiness	percentage of memory page a container can swap
-- memory-reservation	soft limit smaller than –memory used as reserved
--kernel-memory	max kernel memory allocated to a container
--oom-kill-disable	forbid the kernel from killing container when out-of-memory

***Table 3.3**: RAM parameters*

Memory resource is one of the two computational resources of extreme relevance; a missing constraint could lead to an **Out Of Memory Exception** (**OOME**) that could potentially bring down the host. Docker can set memory limitation, which enable the container to consume only the allocated memory; this methodology is also known as **hard limit**. Alternatively, Docker can allow container to consume as much memory as it needs, but only if certain conditions are met, also known as **soft limit**, as per *Table 3.3, RAM parameters.* The **Central Processing Unit CPU** resource is the second important factor when considering computational resources. Similarly, by Docker design, containers access the host hardware's CPU resources are potentially unlimited. There are two ways to configure Central Processing Unit resources constraints to containers:

- CFS Scheduler
- Realtime Scheduler

The **Completely Fair Scheduler (CFS**) is a feature of the Linux Kernel that allocates and handles CPU resources for processes execution. The main purpose of the CFS is to maximize the overall CPU efficiency in term of process handling. As illustrated in *Table 3.4*, *CFS Parameters*, Docker can modify the Cgroups setting for containers using the following flags:

Option	Description
--cpus=<value>	Allocates the available host CPU resources to container
--cpu-period=<value>	Defines the CPU CFS Scheduler period and is used with –cpu-quota
--cpu-quota=<value>	Sets a CFS quota on the container that represents the microseconds per each –cpu-period limitations
--cpuset-cpus	Sets a limit to the number of CPUs or Cores a container can use; the first CPU has value 0

https://t.me/learningnets

--cpu-shares	Default value is 1024 and can be increased or decreased to increase or decrease the weight of the container in relation to the CPU cycles

***Table 3.4**: CFS parameters*

Let's apply a security scenario to the use of the CFS scheduler. In a scenario where the host has 1 CPU only, the container could, according to the application it is serving, consume much more CPU cycles than the host on which it resides. This could lead the host to be unstable and eventually crash or be abused by an attacker. With the following, a quota of maximum 50% of the CPU every second is assigned to the container preserving the stability of the host:

```
1. $ docker run -it --cpu-period=100000 --cpu-quota=50000 debian /bin/
   bash
```

In Docker, the Realtime Scheduler is often used for configuring tasks that cannot use the CFS Scheduler. As the name suggests, the Realtime Scheduler is used at the Kernel level to execute processes when a dedicated portion of the CPU time is available for the complete execution of the assigned tasks. If CPU cycles are used for other processes, and the Kernel determines that the CPU has not enough time available to fulfil the tasks processing via the Realtime Scheduler, this won't start. Requirements for getting the Realtime Scheduler working properly with Docker are as follows:

- Verify that the host's kernel **`CONFIG_RT_GROUP_SCHED`** is enabled by checking whether the file **`/sys/fs/cgroup/cpu.rt_runtime_us`** has been created.
- Verify the dockerd has the flag **`--cpu-rt-runtime`** set, and it also has an appropriate value. The default value is 1000000 microseconds, so check that the containers using the Realtime Scheduler can run for at least the period assigned as the value.
- Set appropriate configurations to individual containers to use the Realtime Scheduler.

Table 3.5, Realtime Scheduler, highlights the flag needed for a container to use the Realtime Scheduler:

Option	Description
--cap-add=sys_nice	Enable the CAP_SYS_NICE capability to the container
--cpu-rt-runtime=<value>	Max time the container can use the Realtime Scheduler
--ulimit rtprio=<value>	Max time the container can use Realtime priority

***Table 3.5**: Realtime Scheduler parameters*

https://t.me/learningnets

The following command describes how to use a container with Realtime Scheduler capability:

```
# Verify the host kernel supports the CPU real-time scheduler
$ zcat /proc/config.gz | grep CONFIG_RT_GROUP_SCHED
CONFIG_RT_GROUP_SCHED=y
$ docker run -it \
    --cpu-rt-runtime=950000 \
    --ulimit rtprio=99 \
    --cap-add=sys_nice \
    --name my-ubuntu \
    ubuntu:22.04
```

There are scenarios in which some application needs to run process at a given time, for a certain period. These processes can alter the normal usage of the resources and could lead to instability. In the previous example, where the host has 1 CPU only, a quota is settled within the container, so the container knows how many CPU cycles it can use at any given time.

Note: The CPU Realtime Scheduler is an advanced feature of the Linux Kernel, and it needs special attention, especially when used for scoping security scenarios.

In this scenario, the container is allowed to process tasks for the CPU cycles within the Realtime Scheduler, given a fixed amount of time in which the task must be completed. This means that with resource constrains, no exploitation or compromise of the container would lead to an escalation of privileges, as the tasks are confined in a determined set of CPU cycles.

Container isolation

Sysadmins should create an ad-hoc environment to run containers. From the security standpoint, the host would be hardened as per the description in *Chapter 2, Hardware and Host OS Security*, but the operating system should also be configured with sufficient security features to preserve the host machine from containers escape and forbid mutual influence.

This is a critical part of the container security process due to the very nature of the containerization system, where the kernel is **shared** between the host and the containers. Refer to *Chapter 1, Containers and Kubernetes Risk Analysis*, for System Level Virtualization and more info. Docker can leverage the following Linux kernel features (see *Chapter 2, Hardware and Host OS Security*):

https://t.me/learningnets

- Namespaces
- AppArmor
- Seccomp

Namespaces

Process isolation is one of the most effective ways to prevent privilege-escalation attacks. This feature is very well integrated with the Least Privilege topic discussed in the previous section, and it is pursued through user remapping and subordinate. Where root privileges are still needed, remapping the user to a less privileged one would mean that the user can still communicate with the Docker system but has no privileges on the host itself.

Docker provides a **default** remapped user called **dockermap**, which has effectively that purpose, but as explained in the *Least Privilege* section of this chapter, a custom user can be created with the characteristic set by **`/etc/subuid`** and **`/etc/subgid`**.

The customer user must already exist, which means **`/etc/passwd`** and **`/etc/group`** are already populated with the relevant entries. It is possible to verify this by running the following command:

```
1. $  id myremapuser
2. uid=1010(myremapuser) gid=1010(myremapuser) groups=1010(myremapuser)
```

To verify that **myremapuser** has entries in both **`/etc/subuid`** and **`/etc/subgid`**, it is possible to **inspect** the respective files, such as in the following example:

```
1. $ cat /etc/subuid
2. myremapuser:152064:65536
3. $ cat /etc/subgid
4. myremapuser:152064:65536
```

This functionality is enabled via the **`--userns-remap`** flag of the dockerd daemon or via the **`daemon.json`** file. A simple command like the following would enable the remap feature via command line command:

```
5. $  dockerd --userns-remap="myremapuser:myremapuser"
```

Or editing the **`/etc/docker/daemon.json`** `file`:

```
1. {
2.   "userns-remap": "myremapuser"
3. }
```

https://t.me/learningnets

To use the default **dockermap** user, just replace **myremapuser** with **default**. To verify that the namespace is applied, start a new container and list the directory in **/var/lib/docker**; the namespaced directory would be equal to {**subuid. subuid**}; for example, let's assume that the **subuid** and **subgid** values for the user **myremapuser** are as follows:

```
1. myremapuser:152064:65536
```

Then, listing the folder in the container would return the following:

```
1. $ sudo ls -l /var/lib/docker/152064.152064/
2.
3. total 14
4. drwx------ 5 152064 152064 5 Jun 21 21:19 aufs
5. drwx------ 3 152064 152064 3 Jun 21 21:21 containers
6. drwx------ 3 root   root   3 Jun 21 21:19 image
7. drwxr-x--- 3 root   root   3 Jun 21 21:19 network
8. drwx------ 4 root   root   4 Jun 21 21:19 plugins
9. drwx------ 2 root   root   2 Jun 21 21:19 swarm
10. drwx------ 2 152064 152064 2 Jun 21 21:21 tmp
11. drwx------ 2 root   root   2 Jun 21 21:19 trust
12. drwx------ 2 152064 152064 3 Jun 21 21:19 volumes
```

AppArmor

Not many may know that Docker generates a default AppArmor profile that can be applied to containers, called **docker-default**. The AppArmor logic has been explained in *Chapter 2, Hardware and Host OS Security* but to recall the concepts, AppArmor provides application security profiles via policies. It is worth noting that the **docker-default** profile is applied only to the containers and not to the Docker Engine itself, even though a Docker Engine AppArmor profile does exist.

The Docker AppArmor default profile is likely to be considered a moderate protection profile; it provides security but also permits application compatibility. The template from which the profile is generated has been published on the Docker GitHub repository.

When a container starts, Docker uses the **docker-default** policy unless specified otherwise via the **security-opt** parameter:

```
1. $  docker run --rm -it --security-opt apparmor=my-apparmor-
   policy hello-world
```

https://t.me/learningnets

To generate new profiles that satisfy specific security requirements, GitLab hosts a quick guide for QuickProfileLanguage in their AppArmor Wiki document page, which is available at **https://gitlab.com/apparmor/apparmor/-/wikis/QuickProfileLanguage**.

Save the new profile in the **/etc/apparmor.d/containers/my-docker-apparmor** file and then load it into AppArmor via parser:

```
$  sudo apparmor_parser -r -W /etc/apparmor.d/containers/my-docker-apparmor
```

Start a new container with the newly created AppArmor profile:

```
$   docker run --security-opt "apparmor=my-docker-apparmor" \
      -p 80:80 -d --name apparmor-nginx nginx
```

Exec into the container will eventually confirm that the profile has been applied and is denying application to be executed:

```
root@6da5b2c830b1:~# ping 8.8.8.8
ping: Lacking privilege for raw socket.

root@6da5b2c830b1:/# top
bash: /usr/bin/top: Permission denied

root@6da5b2c830b1:~# touch ~/some-file
touch: cannot touch 'thing': Permission denied
```

Both **dmesg** and **aa-status** can be used for debugging.

Seccomp

Secure Computing Mode is a feature of the Linux kernel aiming to restrict the syscalls a process is allowed to perform. A deeper tech note is in *Chapter 2, Hardware and Host OS Security*. To use this feature in Docker, dockerd must be built with the **seccomp** flag and the kernel must support it via **CONFIG_SECCOMP**.

To verify that the kernel has SECCOMP enabled, run the following:

```
$ grep CONFIG_SECCOMP= /boot/config-$(uname -r)
CONFIG_SECCOMP=y
```

As per AppArmor, Docker provides a default seccomp profile that disables 44 syscalls out of 300, again a moderately secure policy that allows application compatibility.

https://t.me/learningnets

The default Docker Seccomp profile works by defining a **defaultAction** for **SCMP_ACT_ERRNO**, which returns a **Permission Denied** error unless a specific syscall is allowed. The default profile can be replaced using the **-security-opt** flag.

```
1. $ docker run --rm -it \
2.      --security-opt seccomp=/path/to/seccomp/my-profile.json hello-
   world
```

Network security

Docker has an in-house security solution often not very well known. While the principle of network segregation is always valid, even and much more at the container level, it is not always recognized as a best practice. Docker provides a default bridge network, and when a new container starts, it automatically connects to it unless specified otherwise. The idea is to treat the container network as any other **Virtual Local Area Network** (VLAN) to generate a natural separation between any other corporate vertical. The Docker network is no different from the accounting network, the developer network or the lab network; each of these networks must exist independently from the others.

When there is a reason for which a container should talk to another container, let's say the **web** container talking to the **database** container, both should live only within the environment they belong to. There is no reason for which the **staging** web container should talk to the **production** database container or vice versa, and it is also recommended not to publicly expose container without providing the necessary security guardrails. As far as logic approach is concerned, in this section, we are going to look at something a little different, which applies one of the most relevant security principles to the Docker network mechanism creating a Docker encrypted network.

Note: This small lab requires two Linux Docker Hosts member of a Docker Swarm with one manager node and one worker node.

On systems with more than one network card, Docker Swarm will ask for the IP address to be specified; with **--advertise-addr**, creating a Docker Swarm is as simple as running:

```
1. $ docker swarm init --advertise-addr 192.168.1.226
2. Swarm initialized: current node (p6khjpzdi76dopd76zyysy-
   d2z) is now a manager.
3. To add a worker to this swarm, run the following command:
4.
```

https://t.me/learningnets

```
5.      docker swarm join \
6.      --token SWMTKN-1-{a-very-long-token} 192.168.1.226:2377
7.  ...
```

Docker Swarm nodes can be listed with the **`docker node ls`** command. Docker overlay network encrypts control plane traffic by default, and then the purpose of this small exercise is to get the data plane traffic encrypted as well; it is not encrypted by default. The encryption mechanism relies on the **encrypted** option flag with the **`docker network`** command. To start, it is necessary to create a Docker network type **overlay**, adding the encryption parameter, such as the following example:

```
1. $ sudo docker network create --driver overlay --opt encrypted my-
   encrypted-network
```

Note: The Docker network create command will not work without the Docker Swarm.

Inspect **`my-encrypted-network`** to check for the encrypted flag at line 21:

```
1. [
2.     {
3.         "Name": "my-encrypted-network",
4.         "Id": "klx0f6g300vv861xs4coawyci",
5.         "Created": "2023-03-05T19:00:23.583326803Z",
6.         "Scope": "swarm",
7.         "Driver": "overlay",
8.         "EnableIPv6": false,
9.         "IPAM": {
10.            "Driver": "default",
11.            "Options": null,
12.            "Config": [
13.                {
14.                    "Subnet": "10.0.2.0/24",
15.                    "Gateway": "10.0.2.1"
16.                }
17.            ]
18.        },
```

https://t.me/learningnets

```
19.         "Options": {
20.             "com.docker.network.driver.overlay.vxlanid_
    list": "4098",
21.             "encrypted": ""
22.         },
23.         "Labels": null
24.     }
25. ]
```

The flag VXLAN ID at line 20 indicates that the application traffic on the data plane traffic is encrypted. When listing the networks from the manager node, the encrypted network will be visible:

```
1. manager$ sudo docker network ls
2. NETWORK ID          NAME                  DRIVER            SCOPE
3. klx0f6g300vv        my-encrypted-network  overlay           swarm
```

To let the node join Docker Swarm, log in to the worker node and run the following command:

```
1. $ sudo docker swarm join --token SWMTKN-1--{a-very-long-
   token} 192.168.1.226:2377
2. This node joined a swarm as a worker.
```

But the worker node won't have to access to the encrypted network because a worker node is able to identify a network only if runs a container that requires that network, as per the following example:

```
1. node$ sudo docker network ls
2. NETWORK ID          NAME          DRIVER            SCOPE
3.
```

Let's then deploy a service on **`my-encrypted-network`** so that the worker node can create a container that will be able to connect to the encrypted network:

```
1. $ sudo docker service create --name service1 \
2. --network=my-encrypted-network --replicas=4 \
3. alpine:latest sleep 1d
```

https://t.me/learningnets

```
4.
5. c6e97a29h3bz
```

and verify that the service has been deployed successfully on the worker node:

```
1. $ sudo docker service ls
2. ID            NAME      MODE        REPLICAS  IMAGE
3. c6e97a29h3bz  service1  replicated  4/4       alpine:latest
```

Listing the network from the worker node:

```
1. node$ sudo docker network ls
2. NETWORK ID          NAME                  DRIVER            SCOPE
3. klx0f6g300vv        my-encrypted-network  overlay           swarm
```

This will successfully return the entry with the encrypted network, meaning that the data plane has been encrypted and the containers on the worker nodes are communicating within a secure encrypted channel.

Mirantis container runtime

Mirantis is the only container runtime claiming to be STIG complaint. STIG stands for **Security Technical Implementation Guide**, and it is a security framework created by the DISA, Defense Information System Agency aimed to provide security guidelines for the US military systems.

Peculiarity of the Mirantis Container Runtime and of all the Mirantis' software suite is the target audience: a secure, trusted container runtime for mission-critical workloads and business-critical applications providing in-house solutions like **Secure and Validated Containers**, and **Secure Validated Encryption** for regulated industries that need to meet federal regulations with encryption standards like **Federal Information Processing Standard** (**FIPS**) 140-2.

Federal Information Processing Standard (**FIPS**) 140-2 is a United States Federal set of security requirements to be satisfied by a cryptographic module for federal applications. The cryptographic module must satisfy those requirements to be considered a viable solution for protecting data within the US Federal System. Readers with interest in security elevation standards will find the **National Institute of Standards and Technology** (**NIST**) publication about the FIPS encryption standard mechanism for federal agencies available at **https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.140-2.pdf**.

https://t.me/learningnets

An interesting exclusion

In the Container Manager List, *Figure 3.4, Container Engines and Managers*, readers would have probably noted a container system that Kubernetes has excluded from the Container Runtime Interface list: Podman. Although it is possible use Podman with Kubernetes, that's not really where the industry is going today; nevertheless, for the scope of this chapter and to demonstrate the potentiality of the Podman system, following a description of a possible security scenario for Podman, that readers may find interesting.

The main technical difference between Podman and other container systems like Docker is that Podman works on a **fork/exec** model rather than a **client-server** model. This essentially means that Podman has no daemon service running on the computer that "answers" or "interprets" the command and parameters input from a command line tool or API client; rather, it creates sub-processes of the main process, like what Apache Web Server is doing with its **httpd** daemon, creating an intrinsically more secure environment.

This can help in many ways, but for the sake of this argument, let's introduce one of the features of the Linux Kernel, the so-called **audit** security feature. The **audit** feature is one of the system administrator's favorite as it allows the tracking of security events and logs them to the **audit.log** file. The **audit.log** file can be stored on the same Linux box, and as a very useful option for containers, can be stored remotely to preserve the log integrity.

Thanks to the **audit** system, it is possible to track down, the user activities among other things. When users access, modify or delete system files, such as the **shadow** file, the **audit** system records this activity. Two of the parameters that **audit** records is the Process ID and the **User ID** (**UID**), so the logs can show the association between the user and the process. The key difference between Podman and Docker is how **audit** logs the information through the container platform.

Docker	Podman
O_NONBLOCK a3=0x1b6 items=2 ppid=12763 pid=12782 auid=unset uid=root gid=root	O_NONBLOCK a3=0x1b6 items=2 ppid=10571 pid=10583 auid=luigi uid=root gid=root

***Table 3.6**: Docker and Podman*

Table 3.6, Docker and Podman, shows the audit logs for user activities. While Podman preserves the AUID, Docker's audit log for AUID is **unset**. Since the container is a child process of the Docker daemon, which is a child process of the **systemd** system, the Process ID that **audit.log** can record is the same for all the three subsystems of the **init** system, making it impossible for Docker to determine the User ID. This means that the auditor or security analyst looking at those logs might know that a file has

https://t.me/learningnets

been modified, but the user identity would be lost. On the other hand, Podman, with its fork/exec model for containers, can provide the information about the User ID, because by kernel design, a forked process is executed from the initial login process, and therefore, the container would inherit the same login UID as its parent process.

Secure connection

The Docker daemon or **dockerd** can listen through three different sockets: **unix**, **tcp**, and **fd**. The default socket for dockerd is **unix**, and it runs with **root** permissions or **docker group** membership. Being the unix socket a localhost socket, when in a corporate environment, **dockerd** needs to be accessed remotely; the **tcp** socket must be enabled. In doing so, the security risk associated with the socket exposure will increase drastically.

Enabling the **tcp** socket is translated into the capability to communicate with the **Docker Engine API**. The Docker Engine API is a RESTful API system running within the Docker Engine platform, similar to what **kube-apiserver** is to Kubernetes. The footprint of the Docker API system is the same as that of the Docker command line interface, but while the **docker-cli** is a client-server system interacting with the Linux Docker daemon, the Docker API is network system accessed by an HTTP client, expanding the attack surface.

In the following exercise, readers will understand how to secure the Docker API by creating and implementing a signed certificate into the API system that an API client can consume, securing the overall network communication. The main benefit of using the Docker Engine API is in conjuction with CI/CD pipelines, with a system like Jenkins for example. *Table 3.7, Docker API requirements*, summarizes the Docker API characteristics and features.

Information	Description
TCP Socket	/var/run/docker.sock
TCP Port	2375 non-encrypted connections
TCP Port	2376 encrypted connections
Docker daemon file	/etc/docker/daemon.json
Configuration override	/etc/systemd/system/docker.service.d/override.conf
openssl	Linux toolkit to handle encryption

***Table 3.7**: Docker API Requirements*

To enable the Docker API **tcp** socket on Ubuntu, if Docker has been installed as part of the installation of the operating system, usually the configuration file can be modified through the **snap** command. To verify that Docker is part of the snap system, run the following command:

https://t.me/learningnets

```
1. $ snap list
2. Name    Version        Rev    Tracking       Publisher   Notes
3. core18  20230207       2697   latest/stable  canonical✓  base
4. core20  20230207       1828   latest/stable  canonical✓  base
5. docker  20.10.17       2746   latest/stable  canonical✓  -
6. lxd     5.0.2-838e1b2  24322  5.0/stable/…   canonical✓  -
7. snapd   2.58.2         18357  latest/stable  canonical✓  snapd
```

In this scenario, the Docker **daemon.json** file is located in **/var/snap/docker/current/config** and can be edited to enable the TCP socket by appending line 4 of the following code to the original file configuration settings, including the **hosts** parameters to open a socket connection with an assigned IP address. Usually, this would be the IP address of the system where **dockerd** is running, but it is initially recommended to use the localhost IP only to limit the connection to the host machine using a non-encrypted connection port, given that a TLS certificate is yet to be created.

```
1. {
2.     "log-level":        "error",
3.     "storage-driver":   "overlay2",
4.     "hosts": ["tcp://127.0.0.1:2375", "unix:///var/run/docker.sock"]
5. }
```

To reload the new configuration via snap, you can run **sudo snap restart docker**, and once restarted, the new configuration is verifiable using the following netstat command:

```
1. $ sudo netstat -lntp | grep dockerd
2. Proto Recv-Q Send-Q  Local Address   Foreign Address  State  PID/
   Program name
3. tcp        0      0  127.0.0.1:2375  0.0.0.0:*        LISTEN 33941/
   dockerd
```

Similarly, in any other Linux distribution it is possible to open, edit if already present, or create the **/etc/docker/daemon.json**, and enabling the tcp socket with the following code:

```
1. {
2.   "hosts": ["unix:///var/run/docker.sock", "tcp://127.0.0.1:2375"]
3. }
```

https://t.me/learningnets

In systemd-based operating systems, this is likely to cause a conflict because the dockerd daemon cannot execute the same option twice from two different places; therefore, this approach is not recommended. However, for providing a visual output of the command, it is possible to override systemd with the **`--config-file`** parameter, which will start a standalone instance of the dockerd system and fail the dockerd systemd unit, as illustrated in the following example:

```
$ sudo dockerd --config-file /etc/docker/daemon.json
INFO[2023-03-06T21:54:54.733800526Z] Starting up
WARN[2023-03-06T21:54:54.734946794Z] Binding to IP address without --tlsverify is insecure and gives root access on this machine to everyone who has access to your network.  host="tcp://127.0.0.1:2375"
WARN[2023-03-06T21:54:54.735034466Z] Binding to an IP address, even on localhost, can also give access to scripts run in a browser. Be safe out there!-  host="tcp://127.0.0.1:2375"
...
# TLDR - Omitted
# TLDR - Omitted
...
INFO[2023-03-06T21:54:55.976365909Z] Loading containers: start.
INFO[2023-03-06T21:54:56.741562573Z] Default bridge (docker0) is assigned with an IP address 172.17.0.0/16.
INFO[2023-03-06T21:54:56.990946147Z] Daemon has completed initialization
INFO[2023-03-06T21:54:57.131262701Z] API listen on 127.0.0.1:2375
INFO[2023-03-06T21:54:57.155290037Z] API listen on /var/run/docker.sock
```

Note that the majority of the output in the previous command has been omitted for obvious reasons, but the significant piece, in relation to the security discussed in this part of the chapter, has been kept, because it can provide valuable insights.

Note: Changing configuration on both the Docker systemd unit and the daemon.json files is cause of conflict and will stop Docker daemon from starting. Therefore, it is recommended to choose only one of the options, and if systemd is preferred, delete the daemon.json file from the /etc/docker folder.

https://t.me/learningnets

More broadly, to enable the Docker API **tcp** socket through **systemd** unit file configuration, edit the **override.conf** file located in the **/etc/systemd/system/docker.service.d/** folder. Note that the **[Service]** section is initially commented out; remove the comment and edit the file to look like the following code:

```
1. [Service]
2. ExecStart=
3. ExecStart=/usr/bin/dockerd -H fd:// -H tcp://127.0.0.1:2375
```

The **systemd** unit needs the empty **ExecStart** at line two to honor the original configuration and the **ExecStart** at line three to initialize the Docker API Engine, missing any of the two directives would return an error, type "**oneshot services**". Reload systemctl and restart the Docker service with the following commands:

```
1. $ sudo systemctl daemon-reload
2. $ sudo systemctl restart docker.service
```

Then, verify that the Docker daemon picked up the new configuration a simple **sudo systemctl status docker.service** will return:

```
1. $ sudo systemctl status docker.service
2. ● docker.service - Docker Application Container Engine
3.      Loaded: loaded (/lib/systemd/system/docker.
   service; enabled; vendor preset: enabled)
4.     Drop-In: /etc/systemd/system/docker.service.d
5.              └─override.conf
6.      Active: active (running) since Wed 2023-03-
   08 00:13:55 UTC; 1min 55s ago
7. TriggeredBy: ● docker.socket
8. ...
9.      CGroup: /system.slice/docker.service
10.             └─89548 /usr/bin/
   dockerd -H fd:// -H tcp://127.0.0.1:2375
```

The three approaches explained in this section are all valid with the recommendation to use **snap** on Ubuntu systems and the **override.conf** file on systemd-based Linux systems. Both methodologies are persistent on reboot.

https://t.me/learningnets

Server certificate

With the Docker daemon listening on the localhost port 2375, it is possible to configure a TLS connection using a self-signed certificate to secure the communication with the Docker API Engine. This step requires having installed on the operating system the **openssl** package, which is quite common among the Linux distributions. The first step is to generate self-signed certificate private and public keys on the Docker host. When requested, enter the pass phrase (also known as password) for the private key:

```
$ sudo openssl genrsa -aes256 -out ca-private-key.pem 4096
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
```

Once the private key is obtained, it is possible to generate the public key using the private key as an input. Verify that the **Common Name** at point 15 matches the hostname of the host; this must also be used when creating the **Certificate Signing Request** (**CSR**).

```
$ sudo openssl req -new -x509 -days 365 -key ca-private-key.
pem -sha256 -out ca-public-key.pem
Enter pass phrase for ca-private-key.pem:
You are about to be asked to enter information that will be incorpo-
rated
into your certificate request.
What you are about to enter is what is called a Distin-
guished Name or a DN.
There are quite a few fields, but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:UK
State or Province Name (full name) [Some-State]:Hants
Locality Name (eg, city) []:.
Organization Name (eg, company) [Internet Widgits Pty Ltd]:.
Organizational Unit Name (eg, section) []:.
Common Name (e.g. server FQDN or YOUR name) []:server
Email Address []:.
```

https://t.me/learningnets

The next step is to create the server key:

```
$ sudo openssl genrsa -out server-key.pem 4096
```

And with the server key as input, we can create the **Certificate Signing Request** (**CSR**):

```
$ sudo openssl req -subj "/CN=server" -sha256 -new -key server-key.
pem -out server.csr
```

TLS connections can be established via IP address or DNS name, but the IP must be specified when creating the certificate. For the purpose of this exercise, we are going to use 192.168.1.226 as a LAN IP address through the creation of the **`extfile.cnf`** file. The IP might be different according to the various network scenarios, all of which are not reproducible in this example.

```
$ echo subjectAltName = DNS:server,IP:192.168.1.226,
IP:127.0.0.1 >> extfile.cnf
```

Then, configure **`extfile.cnf`** to use the Docker daemon key for server authentication.

```
$ echo extendedKeyUsage = serverAuth >> extfile.cnf
```

So far, the following five files have been created:

```
$ ls -l
-rw------- 1 luigi luigi 3434 Mar  8 18:44 ca-private-key.pem
-rw-rw-r-- 1 luigi luigi 2033 Mar  8 18:52 ca-public-key.pem
-rw-rw-r-- 1 luigi luigi   88 Mar  8 21:10 extfile.cnf
-rw-rw-r-- 1 luigi luigi 1582 Mar  8 19:47 server.csr
-rw------- 1 luigi luigi 3268 Mar  8 19:46 server-key.pem
```

The last step on the server certificate topic is to generate the signed certificate. If the process is progressing as expected, the **`subject=CN = server`** is auto populated, and the only input requested by the command will be the pass phrase chosen at the very first step, when creating the private key.

```
$ sudo openssl x509 -req -days 365 -sha256 -in server.csr -CA ca-
public-key.pem \
-CAkey ca-private-key.pem -CAcreateserial -out server-cert.pem \
-extfile extfile.cnf
Certificate request self-signature ok
subject=CN = server
Enter pass phrase for ca-private-key.pem:
```

https://t.me/learningnets

The outcome of the server certificate process to be considered for the purpose of this exercise is the **`server-cert.pem`** certificate file.

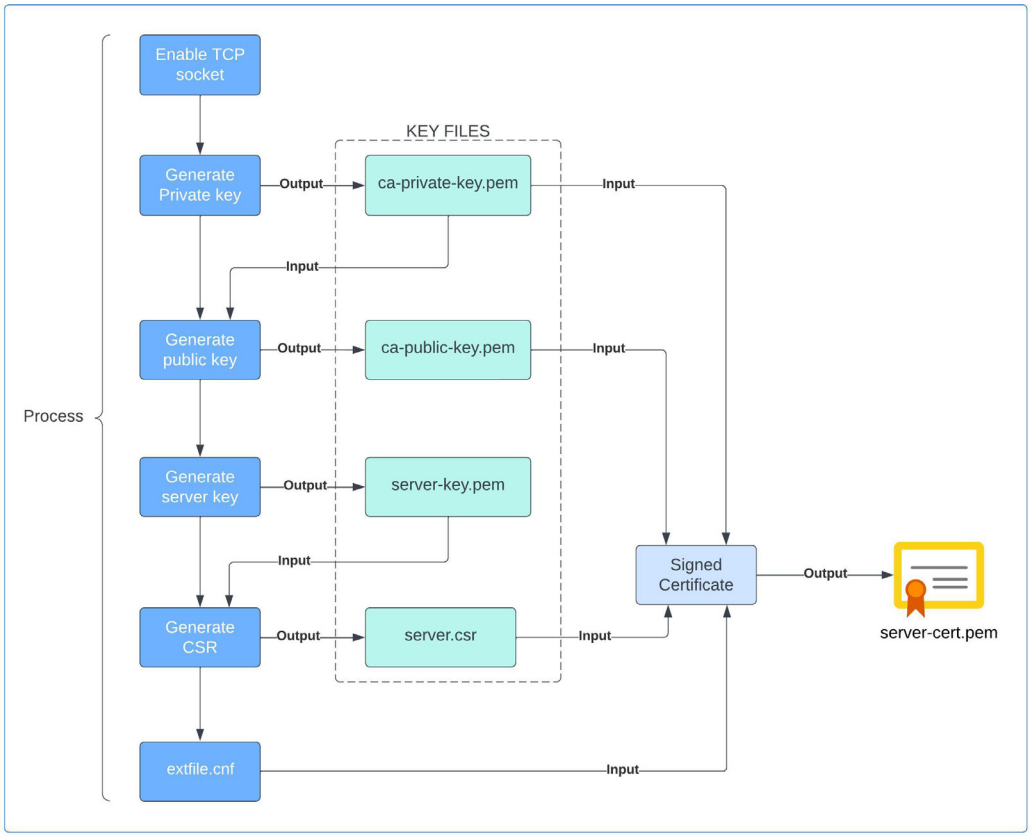

***Figure 3.7**: Server Certificate Process*

This topic is considered an advanced concept, so *Figure 3.7, Server Certificate Process*, can help readers visualize the overall process.

Client certificate

With the server certificate ready, a client certificate must be generated to allow the client to securely consume the Docker API Engine. Eventually, it is possible to generate the client key and client certificate signing request on the same host where the server certificate has been generated. To generate the client key, use the following example command:

```
1. $ sudo openssl genrsa -out client-key.pem 4096
```

https://t.me/learningnets

The **client-key.pem** will be then used as an input to generate the **Certificate Signing Request** (**CSR**).

```
1. $ sudo openssl req -subj '/CN=client' -new -key client-key.
   pem -out client.csr
```

To enable the client key to sustain client authentication, modify the extension file to use the following parameter:

```
1. $ echo extendedKeyUsage = clientAuth > extfile-client.cnf
```

The **extfile-client.cnf**, as the name suggests, is related to the client certificate creation and differs from **extfile.cnf** used during the server certificate process creation. With the client key, the client certificate signing request ready, and the input of the server private and public key, the next step is to generate the client signed certificate:

```
1. $ sudo openssl x509 -req -days 365 -sha256 -in client.csr -CA ca-
   public-key.pem \
2. -CAkey ca-private-key.pem -CAcreateserial -out client-cert.pem \
3. -extfile extfile-client.cnf
4. Signature ok
5. subject=CN = client
6. Getting CA Private Key
7. Enter pass phrase for ca-private-key.pem:
```

The outcome of the client certificate process to be considered for the purpose of this exercise is the **client-cert.pem** certificate.

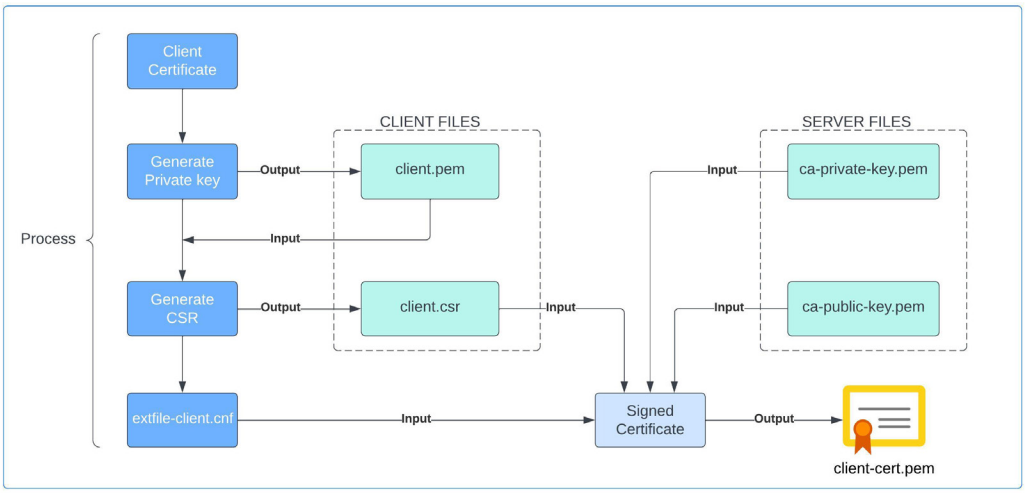

Figure 3.8: Client Certificate Process

https://t.me/learningnets

Figure 3.8, Client Certificate Process, helps in recalling the paths of some of the server certificate files needed to generate the client certificate.

```
$ ls -l
-rw------- 1 luigi luigi 3434 Mar  8 18:44 ca-private-key.pem
-rw-rw-r-- 1 luigi luigi 2033 Mar  8 18:52 ca-public-key.pem
-rw-rw-r-- 1 luigi luigi 1919 Mar  8 21:54 client-cert.pem
-rw-rw-r-- 1 luigi luigi 1582 Mar  8 21:48 client.csr
-rw------- 1 luigi luigi 3268 Mar  8 21:44 client-key.pem
-rw-rw-r-- 1 luigi luigi   30 Mar  8 21:50 extfile-client.cnf
-rw-rw-r-- 1 luigi luigi   88 Mar  8 21:10 extfile.cnf
-rw-rw-r-- 1 luigi luigi 1964 Mar  8 21:22 server-cert.pem
-rw-rw-r-- 1 luigi luigi 1582 Mar  8 19:47 server.csr
-rw------- 1 luigi luigi 3268 Mar  8 19:46 server-key.pem
```

In addition to the list of files generated for the server certificate process, the previous updated list includes the client certificate files.

Enable dockerd TLS

Once you have obtained the **`server-cert.pem`** and the **`client-cert.pem`** files, it is possible to safely delete the two extensions files and the two CSR files as part of an initial cleanup process:

```
$ rm server.csr client.csr extfile.cnf extfile-client.cnf
```

It is recommended to change permissions on the key files to make them readable only to the legitimate owner. While certificate files are meant to be openly distributable, it is anyway recommended to secure them from accidental overwrite or damage changing the file permissions:

```
$ chmod 0400 ca-private-key.pem client-key.pem server-key.pem
$ chmod 0444 ca-public-key.pem server-cert.pem client-cert.pem
```

With the preceding reviewed files permissions, the directory listing should look as follows:

```
$ ls -l
-r-------- 1 luigi luigi 3434 Mar  8 18:44 ca-private-key.pem
-r--r--r-- 1 luigi luigi 2033 Mar  8 18:52 ca-public-key.pem
```

https://t.me/learningnets

```
-r--r--r-- 1 luigi luigi 1919 Mar  8 21:54 client-cert.pem
-r-------- 1 luigi luigi 3268 Mar  8 21:44 client-key.pem
-r--r--r-- 1 luigi luigi 1964 Mar  8 21:22 server-cert.pem
-r-------- 1 luigi luigi 3268 Mar  8 19:46 server-key.pem
```

When the server certificate is obtained, the **dockerd** daemon would need to be updated to use the certificate files, and the listening port should be changed from 2375 to 2376. For a quick verification of the outcome of the entire process, if the Docker daemon is running, stop it by executing **sudo snap stop docker** on Ubuntu machines or using **sudo systemctl stop docker.service** on systemd-based Linux distributions, and then run the following command:

```
$ sudo dockerd \
    --tlsverify \
    --tlscacert=ca-public-key.pem \
    --tlscert=server-cert.pem \
    --tlskey=server-key.pem \
    -H=0.0.0.0:2376
```

The preceding command should start an instance of the dockerd daemon within the bash console with an output similar to the following:

```
INFO[2023-03-08T23:14:09.848617553Z] Starting up
INFO[2023-03-08T23:14:09.880586191Z] detected 127.0.0.53 nameserver, assuming systemd-resolved, so using resolv.conf: /run/systemd/resolve/resolv.conf
...
INFO[2023-03-08T23:14:10.285920627Z] Loading containers: start.
INFO[2023-03-08T23:14:10.976319524Z] Default bridge (docker0) is assigned with an IP address 172.17.0.0/16. D
INFO[2023-03-08T23:14:11.191231310Z] Loading containers: done.
INFO[2023-03-08T23:14:11.252291147Z] Daemon has completed initialization
INFO[2023-03-08T23:14:11.445213032Z] API listen on [::]:2376
```

Since the Docker daemon is listening on 0.0.0.0:2376, every machine on the same network can communicate on port 2376 as far as it has been provided with the following TLS certificates: **ca-public-key.pem**, **client-cert.pem** and **client-key.pem**. It is possible to verify that the secure connection with the Docker API has been established by running on the client system:

https://t.me/learningnets

```
$ sudo docker --tlsverify \
    --tlscacert=ca-public-key.pem \
    --tlscert=client-cert.pem \
    --tlskey=client-key.pem \
    -H=192.168.1.226:2376 version
#Client:
# Cloud integration: v1.0.31
# Omitted
Server:
 Engine:
  Version:          20.10.12
  API version:      1.41 (minimum version 1.12)
  Go version:       go1.17.3
  Git commit:       20.10.12-0ubuntu4
  Built:            Mon Mar  7 15:57:50 2022
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.5.9-0ubuntu3.1
  GitCommit:
 runc:
  Version:          1.1.0-0ubuntu1.1
  GitCommit:
 docker-init:
  Version:          0.19.0
  GitCommit:
```

The counter verification confirms that the Docker daemon running on the remote server refuses connections that are not presenting the TLS certification and key files:

```
$ sudo docker -H=192.168.1.226:2376 version
Error response from daemon: Client sent an HTTP re-
quest to an HTTPS server.
```

https://t.me/learningnets

To make changes permanent, the preferred methodology is to edit the systemd **override.conf** file located in the **/etc/systemd/system/docker.service.d/** folder, adding the **tcp** socket on port 2376 and the TLS parameters as per the following example code:

```
1. [Service]
2. ExecStart=
3. ExecStart=/usr/bin/dockerd -H fd:// -H tcp://0.0.0.0:2376 --tlsverify --tlscacert=/home/luigi/ca-public-key.pem --tlscert=/home/luigi/server-cert.pem --tlskey=/home/luigi/server-key.pem
```

Reload **systemctl** and restart the Docker service with the following commands:

```
1. $ sudo systemctl daemon-reload
2. $ sudo systemctl restart docker.service
```

Then, verify if the Docker daemon picked up the new configuration by a simple executing **sudo systemctl status docker.service** in the terminal. This command will confirm that the **dockerd** daemon API is listening on **[::]:2376.** The Docker client command used for the previous verification can be invoked again to verify that the connectivity from the client machine is working as expected. Of course, working with a remote Docker API Engine in need of additional parameters even for simple command like the following example can impact the user experience:

```
1. $ sudo docker --tlsverify --tlscacert=ca-public-key.pem \
2. --tlscert=client-cert.pem --tlskey=client-key.pem \
3. -H=192.168.1.226:2376 images
4. REPOSITORY        TAG       IMAGE ID       CREATED      SIZE
5. jenkins/jenkins   latest    f72705c021e2   2 days ago   471MB
6. grafana/grafana   latest    944e84f25bc7   7 days ago   329MB
7. hashicorp/vault   latest    166e81af0b3c   9 days ago   187MB
```

To simplify the connection from any client machine with the Docker API Engine, it is possible to apply the secure by default methodology suggested by Docker. Since Docker expects to find in the **~/.docker** folder a specific nomenclature for the key files provided, *Table 3.8, Docker Secure by Default Keys*, illustrates the mapping between the key used in this chapter and the key expected by the Docker client.

Key	Docker default keys
ca-public-key.pem	ca.pem
client-cert.pem	cert.pem
client-key.pem	key.pem

***Table 3.8**: Docker Secure by Default Keys*

https://t.me/learningnets

With the following simple commands, on any client machine that needs connectivity to the Docker API Engine, it is possible to leverage the API mechanism remotely as we would do locally, but through a TLS secure channel.

```
$ cp ca-public-key.pem ~/.docker/ca.pem
$ cp client-cert.pem ~/.docker/cert.pem
$ cp client-key.pem ~/.docker/key.pem
$ export DOCKER_HOST=tcp://192.168.1.226:2376 DOCKER_TLS_VERIFY=1
# Connection to 192.168.1.226 without declaring the TLS parameters.
# The docker command looks into the ~/.docker folder for the key
files.
$ docker images
REPOSITORY          TAG       IMAGE ID       CREATED      SIZE
jenkins/jenkins     latest    f72705c021e2   2 days ago   471MB
grafana/grafana     latest    944e84f25bc7   8 days ago   329MB
hashicorp/vault     latest    166e81af0b3c   9 days ago   187MB
```

Eventually, the **curl** command can be used to verify the full interaction with the Docker API Engine as per the following example:

```
$ curl https://192.168.1.226:2376/images/json \
--cert ~/.docker/cert.pem \
--key ~/.docker/key.pem \
--cacert ~/.docker/ca.pem
[{"Containers":-1,"Created":1678280611,"Id":"sha256:f72705c021e29072
0ba23cfc0211e6b0ac979db21c205669eb68b8b8731dc04a"]},
```

The **curl** output has been truncated to the first image listing since the purpose of the command is only to verify the effectiveness of the communication established over TLS with the Docker API Engine. While proceeding to the next section, it is recommended to disconnect the local Docker system from the remote Docker API Engine by simply removing the following two environment variables:

```
$ unset DOCKER_HOST
$ unset DOCKER_TLS_VERIFY
```

The unsetting of the two preceding variables will allow you to continue the implementation of the TLS mechanism into the next topic.

https://t.me/learningnets

Secure CI/CD

The readers familiar with DevOps methodologies would appreciate the previous exercise because it makes integrating containers into **Continuous Integration** (**CI**) and **Continuous Delivery** (**CD**) systems more efficient and secure. For instance, Jenkins, the well-known leading open-source automation server, available at **https://www.jenkins.io**, offers support for several tools, tasks automation aiming to create development and deployment pipelines. It can also integrate popular version control systems like **git**, and can also communicate securely with a Docker API Engine throughout a TLS connection. The security aspects of the Jenkins system itself, including topics like access control, controller isolation, securing builds, **Cross Site Request Forgery** (**CSRF**) protection, exposed service ports, and access control to builds, are outside the scope of this book. To enhance Jenkins security best practices on the various aspects mentioned earlier, follow the recommendations listed at **https://www.jenkins.io/doc/book/security/**.

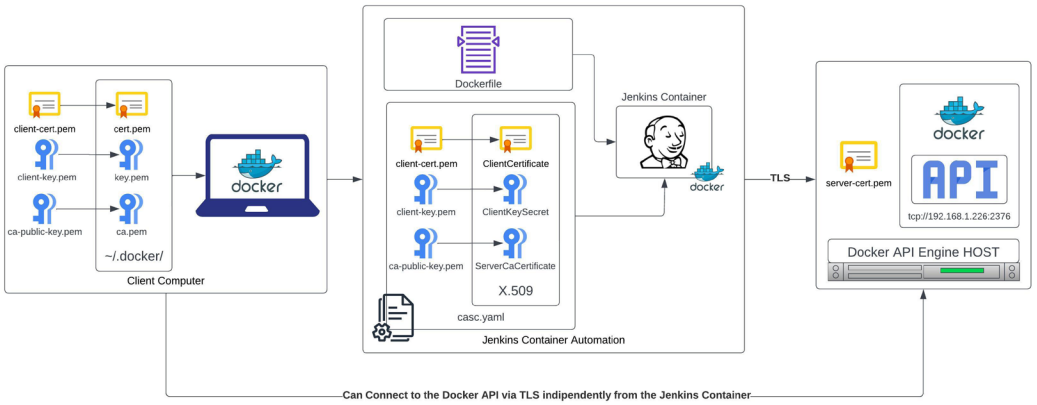

***Figure 3.9**: Jenkins Docker TLS Overview Process*

Normally, the Jenkins installation process would be considered out of scope; a tutorial on this topic would require too many screenshots, and it will not be as beneficial as using coding examples, the process in *Figure 3.9, Jenkins Docker TLS Overview Process*, aims to define a series of steps to obtain a Jenkins server running into a Docker container by leveraging a custom **Dockerfile** that passes key configuration elements through the Jenkins Configuration as a Code plugin, also known as **CasC**, that can remotely execute container jobs by connecting with an external Docker API Engine over TLS secure communication. A summary of the elements needed for this lab are described in *Table 3.9, Secure CI/CD Elements*:

https://t.me/learningnets

Element	Description
jenkins:lts	The latest Jenkins Docker container image
Dockerfile	Build a Jenkins ad-hoc container image
CasC file	Custom YAML file to personalize the Jenkins server upon launch
docker-plugin	Jenkins plugin to execute containerized pipelines
CasC plugin	Jenkins configuration-as-a-code plugin
Job-DSL plugin	Define Jenkins jobs by using declarative code
Docker API	The Docker API Engine as shown in the previous section of this chapter
TLS	Set of keys and certificates to enable secure communication as per *Table 3.8 – Docker Secure by Default Keys*

***Table 3.9**: Secure CI/CD Elements*

The first step is to rebuild the latest Jenkins Docker image to include plugins that are not present in the default container image. Usually, when running Jenkins for the first time, users are asked to go through the installation process, which would include choosing login credentials and the initial plugin installation options. This is where the configuration as a code plugin helps eliminate a series of intermediate steps by eliminating the initial setup wizard and providing a set of parameters that can be consumed beforehand by the system to get as close as possible to a final, ready-to-use Jenkins system. The provided Dockerfile contains comments to help understand the logic behind this approach.

```
1. # syntax=docker/dockerfile:1
2. FROM jenkins/jenkins:lts
3. # Elevate permissions to install additional plugins
4. USER root
5. # Disable interactive installation wizard
6. ENV JAVA_OPTS -Djenkins.install.runSetupWizard=false
7. # set the CasC file var
8. ENV CASC_JENKINS_CONFIG /var/jenkins_home/casc.yaml
9. # Install additional plugins
10. RUN jenkins-plugin-cli --plugins docker-plugin:1.3.0 configuration-
    as-code:latest job-dsl:latest
11. # Copy the casc.yaml file
12. COPY casc.yaml /var/jenkins_home/casc.yaml
```

https://t.me/learningnets

```
13. # Drop back permissions to the regular jenkins user
14. USER jenkins
```

In the same folder where the Dockerfile has been saved, create the **casc.yaml** file that will supply configuration as a code to the Jenkins server. The Configuration as a Code file in this section mainly has three root elements:

- The **credentials** configuration allows the Jenkins server to communicate with a remote Docker API Engine via TLS.
- The **jenkins** configuration for the Jenkins server itself, including the local Docker container system under the **cloud** section.
- A simple script under the **job** definition will be able to execute a shell echo command on a remote container by connecting to the remote Docker API Engine.

All the lines starting with hash (#) sign in the following **casc.yaml** provide a brief comment to help edit the configuration as a code input for the Jenkins server:

```
1. # Generate the X.509 certificate by using a combination of the
2. #  client-cert.pem, client-key.pem, and ca-public-key.pem files
3. credentials:
4.   system:
5.     domainCredentials:
6.     - credentials:
7.       - x509ClientCert:
8.           clientCertificate: |-
9.             -----BEGIN CERTIFICATE-----
10.            # Paste here the content of the client-cert.pem file
11.            # Do not duplicate the BEGIN and END CERTIFICATE lines
12.            # they just show the exact indentation.
13.            -----END CERTIFICATE-----
14.          clientKeySecret: |-
15.            -----BEGIN PRIVATE KEY-----
16.            # Paste here the content of the client-key.pem file
17.            # Do not duplicate the BEGIN and END CERTIFICATE lines
18.            # they just show the exact indentation.
```

https://t.me/learningnets

```
            -----END PRIVATE KEY-----
          serverCaCertificate: |-
            -----BEGIN CERTIFICATE-----
            # Paste here the content of the ca-public-key.pem file
            # Do not duplicate the BEGIN and END CERTIFICATE lines
            # they just show the exact indentation.
            -----END CERTIFICATE-----
          description: "TLS Certificate"
          # ID To be referred into the Cloud section below
          id: "1"
          scope: GLOBAL
jenkins:
  agentProtocols:
  - "JNLP4-connect"
  - "Ping"
  authorizationStrategy:
    loggedInUsersCanDoAnything:
      allowAnonymousRead: false
  systemMessage: "Jenkins custom image with Docker and CasC plugins\
n\n"
  securityRealm:
    local:
      allowsSignup: false
      # since there is no setup process, user credentials
      # are provided on docker run command
      users:
       - id: ${JENKINS_ADMIN_ID}
         password: ${JENKINS_ADMIN_PASSWORD}
  # Docker Configuration
  clouds:
    - docker:
        name: "docker"
```

https://t.me/learningnets

```
50.         dockerApi:
51.           dockerHost:
52.             # As per above x509ClientCert ID
53.             credentialsId: "1"
54.             # Replace this according to the current network configu-
    ration
55.             uri: "tcp://192.168.1.226:2376"
56.         templates:
57.           # Label to be used for the pipeline creation
58.           - labelString: "my-docker-agent"
59.             dockerTemplateBase:
60.               image: "jenkins/inbound-agent"
61.             remoteFs: "/home/jenkins"
62.             connector:
63.               attach:
64.                 user: "jenkins"
65.             instanceCapStr: "2"
66. # Create a test job using the Cloud configuration
67. # through the labelString parameter
68. jobs:
69.   - script: >
70.       job(‹Test Docker Pipeline›) {
71.         label(‹my-docker-agent›).       # Matches the labelStrings
    line 58
72.         steps {
73.             shell(‹echo Hello Docker!›)
74.         }
75.       }
```

From the security standpoint, credentials should be stored to be preserved securely. The **`client-cert.pem`** and **`ca-public-key.pem`** files have been created to be publicly distributable; therefore, it makes sense that the **`clientCertificate`** and **`serverCaCertificate`** parameters are stored in plain text on the Jenkins system, and they can be retrieved by simply reading the **`credentials.xml`** file, while

https://t.me/learningnets

the **client-key.pem** will be encrypted with the Jenkins master key masking the **clientKeySecret** parameter from being retrieved.

Note: As far as this is an acceptable use case for testing purposes, it is not the best option. Storing credentials should be done by adopting a secret mechanism. Jenkins supports many tools that can serve the purpose, such as Docker Secrets via Swarm, Kubernetes Secrets, HashiCorp Vault, Azure Key Vault, AWS Secrets Manager and Parameter Store, and CyberARK; therefore, the current approach is considered an excellent starting point but is not sufficient for a production environment.

The next step is to build the new Jenkins image by invoking the Docker **build** command; the **--no-cache** parameter is useful in the case of multiple builds of the same image. It helps not to bring cached elements that could affect the outcome of the latest build, and the **-t** parameter allows you to tag the new image with a significant name that must be recalled by the Docker **run** command.

```
1. $ docker build --no-cache -t jenkins-docker:tls .
```

The **jenkins-docker:tls** image is now added to the local Docker system and can be listed with a simple **docker images** command. In the beginning, it could be useful not to run the detached mode; it can help in troubleshooting errors. Since the YAML files are very sensitive to indentation, verify that the **casc.yaml** is properly set.

```
1. $ docker run \
2.   --name jenkins \
3.   --rm \                                    # destroy the container
   on exit
4.   -p 8080:8080 \
5.   --env JENKINS_ADMIN_ID=choose-an-user \       # casc.yaml line 44
6.   --env JENKINS_ADMIN_PASSWORD=chose-a-password \      # casc.yaml
   line 45
7.   jenkins-docker:tls                                  # docker
   build tag
```

https://t.me/learningnets

With the Jenkins server finally up and running in a Docker container, open the web interface at **http://localhost:8080** and log in using the environment variables at line five and six of the preceding code.

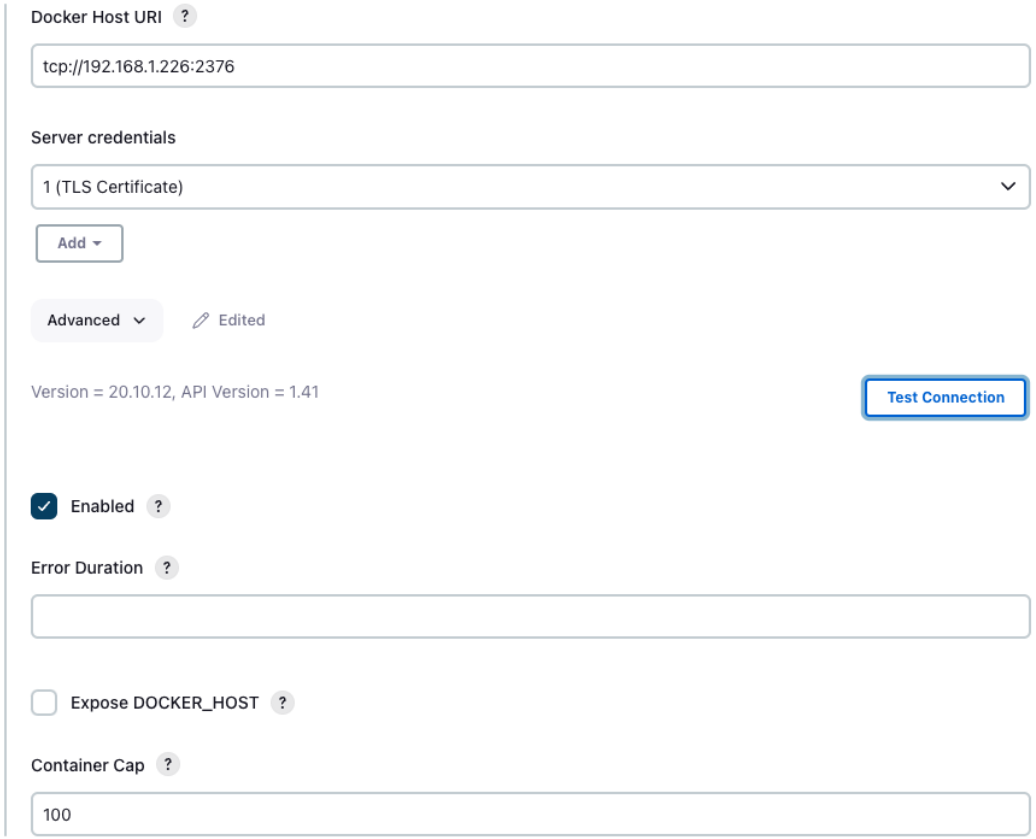

***Figure 3.10**: Jenkins Docker Test TLS Connection*

Before building the job, let's verify that under **Manage Jenkins** | **Manage Nodes and Clouds** | **Configure Clouds** | **Docker Cloud details** | **Test Connection**, the Jenkins server can communicate with the remote Docker API Engine hosted at 192.168.1.226:2376 over TLS secure protocol, as illustrated in *Figure 3.10, Jenkins Docker Test TLS Connection*.

If something went wrong, the message returned will instead be "**Error response from daemon: Client sent an HTTP request to an HTTPS server.**" To be sure to have matched the TLS key with the correct nomenclature provided by the Jenkins system, verify *Table 3.10, Jenkins Docker X.509 Client Certificate*:

https://t.me/learningnets

Key	Jenkins
ca-public-key.pem	serverCACertificate
client-cert.pem	clientCertificate
client-key.pem	clientKeySecret

***Table 3.10**: Jenkins Docker X.509 Client Certificate*

According to **`casc.yaml`** file, a job named **Test Docker Pipeline** should be in place. This job can connect to a remote Docker API Engine over TLS and use the remote Docker system as build agent to spin up on-the-fly remote containers to execute jobs securely. By clicking on **Build Now**, the **Build Queue** will be populated with a **Test Docker Pipeline job**, while on the remote Docker system, by executing **`docker ps`** command we can understand that a container using the jenkins/inbound-agent docker image is running. In this test, everything should be executed quickly considering that the job has just a build step that runs a shell echo command. Upon inspection of the build, the outcome should be as per *Figure 3.11, Jenkins Docker Build over TLS*:

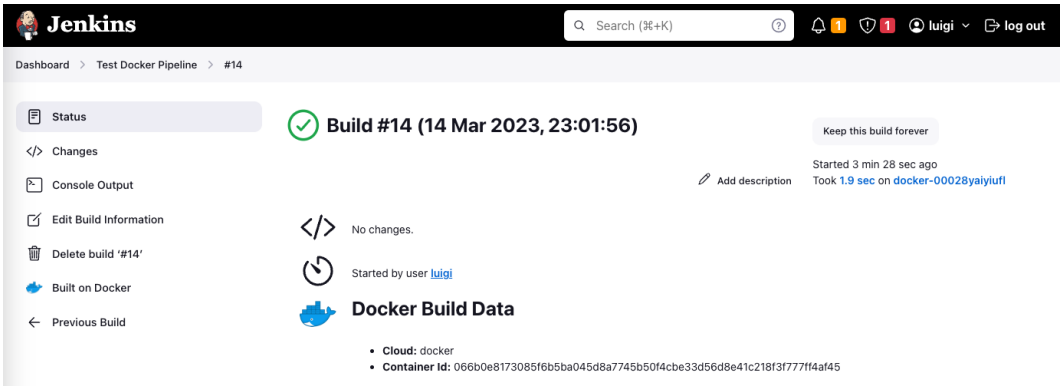

***Figure 3.11**: Jenkins Docker Build over TLS*

Further confirmation can be retrieved by clicking on the **Built on Docker** link, which will return logs similar to what a **docker inspect** command would output, and by clicking on **Console Output**, which will output the shell step defined in the job as per the following example:

```
1. Started by user luigi
2. Running as SYSTEM
3. Building remotely on docker-00028yaiyiufl on docker (my-docker-
   agent) in workspace /home/jenkins/workspace/Test Docker Pipeline
4. [Test Docker Pipeline] $ /bin/sh -xe /tmp/
   jenkins4590919463263968080.sh
```

https://t.me/learningnets

```
5. + echo Hello Docker!
6. Hello Docker!
7. Finished: SUCCESS
```

As the final verification step, the **jenkins/inbound-agent** container image at line sixty of the **casc.yaml** has been downloaded only on the same host where the Docker API Engine is running, while the same image is neither present on the system where Jenkins is running nor inside the Jenkins container itself.

Update life cycle

It is of paramount importance to ensure that the Docker Engine and the host operating system on which Docker is running are updated to reduce the attack surface by eliminating any new known vulnerabilities, some of which can determine a container escape. A similar topic was briefly discussed in *Chapter 1, Containers and Kubernetes Risk Analysis,* in the section related to the Attack Surface. The argument was suggesting utilizing Container-specific OSes like Fedora Core OS, openSUSE Leap Micro, and RancherOS. Container-specific OS are considered "smart operating systems".

The principle behind the update methodology rolled by container-specific OS is simple: the update process is a container itself, which is launched separately from the other containers already running on the system. If an updated package is corrupted or forged, the updating container can be shut down or rolled back.

Intel has created its own container-specific OS distro called **Clear Linux**. It claims to be **stateless**, which means it is running a custom configuration that separates the system configuration from the operating system configuration, so any wrongdoing at the user level won't affect the OS. As part of Intel Clear Containers project, an open-source technology that integrates into Docker and Kubernetes, is optimized for Intel Architecture VT and **Cloud Integrated Advanced Orchestrator** (**CIAO**), and is a complete TLS-based workload scheduler is available at the following address: **https://clearlinux.org/about**.

The most interesting innovation Clear Linux brings to the table is the integration of an automated tool that scans and remediates **Common Vulnerability and Exposure (CVEs**). It uses both the NVD (NIST Vulnerability Database) and MITRE repositories as sources, which aim to keep the attack surface minimal, and it is highly effective from a Zero Trust Model perspective. It also integrates a service that monitors SSH login systematic patterns called **Tallow** as part of the **`openssh-server`** package.

https://t.me/learningnets

Conclusion

In this chapter, we discussed the various aspects of container security, applying the principle of defense-in-depth, looking at containers from any possible angle and applying security principles to make every single process or option as hard as possible to crack for an attacker.

We discussed the basics of containerization and looked at how to apply security to the various steps of the most significant container platforms available in the industry, and we also evaluated the difference between the most prominent containers' runtime and their market application. Further on in the chapter, we looked at Docker in depth and explored the numerous ways of security inter-steps the Docker Engine makes available. Finally, we discussed how to secure Docker API and interface it with CI/CD best practices for security and managed a secure update life cycle.

In the next chapter, we will learn about securing images, registry, scans, and vulnerability management.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

https://t.me/learningnets

https://t.me/learningnets

Chapter 4
Securing Container Images and Registries

Introduction

Container images are the standard delivery format in cloud-native platforms today. Their extremely broad distribution and deployment, either in on-premises or cloud environments, requires reviewing security practices to guarantee container images integrity.

Container image security scanning must be a fundamental part of the container security strategy, even though it should only be part of a larger initiative of security best practices needed to effectively protect the container stack.

Indeed, container registry security, either private or public, must not be overlooked. Uploading, storing and downloading container images from such systems requires a custom set of security best practices to be applied to such repositories.

Structure

In this chapter, we will discuss the following topics:

- Container image hardening
- Building file configuration
- Minimal and distroless image

https://t.me/learningnets

- Scanning and verify images
- Private and Public Registries
- Role-Based Access Control
- Auditability
- Image Control
- Vulnerability Management

Objectives

This chapter aims to provide a set of security best practices for container image creation and life cycle management, private and public container registry security and their indissoluble relationship in actively contributing to the overall security of the container stack. It also aims to provide the best known DevSecOps security approach when considering container images and container registry from the software development life cycle standpoint.

Container image hardening

A container image is a layered package of software, usually derived from a Linux or Windows system distribution, that can be eventually customized. The **image** provides information on how a **container** should be initialized and reviewed, determining what software, applications, or libraries will run and how.

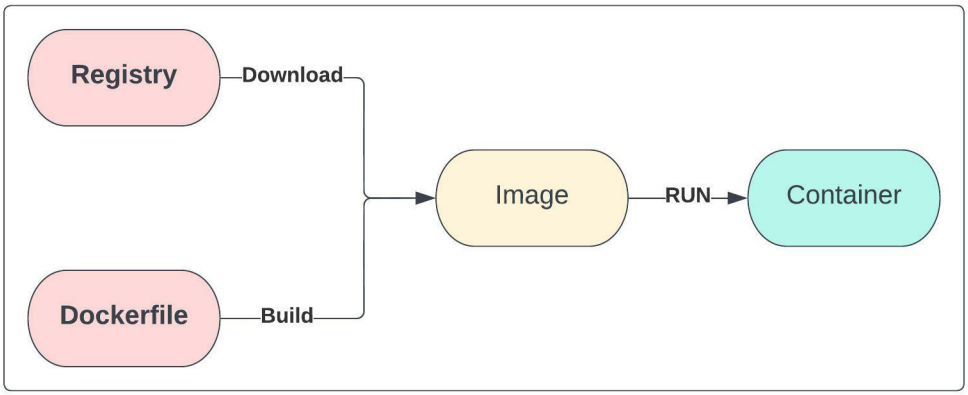

***Figure 4.1**: Images*

Figure 4.1, Images, depicts the two main procedures for obtaining a container image: download a pre-built image from a registry or build a new image from a Dockerfile process. Both approaches pose security challenges.

https://t.me/learningnets

Note: The registry topic will be discussed further in this chapter; refer to the *Private and Public Registries Section*.

Dockerfile is an efficient way to build container images through a declarative text file that contains all the information and commands needed. The Docker build command reads the input from the Dockerfile and create read-only layers, one for each line or directive, combining all the layers in one final image. Usually, a Dockerfile starts from a parent image, an image that has been pre-built and made available locally or in the registry. This is represented by the **FROM** directive. The Dockerfile used in *Chapter 3, Container Stack Security*, in the *Secure CI/CD* section is re-proposed here as an example, excluding the commenting lines:

```
1. # syntax=docker/dockerfile:1
2. FROM jenkins/jenkins:lts
3. USER root
4. ENV JAVA_OPTS -Djenkins.install.runSetupWizard=false
5. ENV CASC_JENKINS_CONFIG /var/jenkins_home/casc.yaml
6. RUN jenkins-plugin-cli --plugins docker-plugin:1.3.0 configuration-
   as-code:latest job-dsl:latest
7. COPY casc.yaml /var/jenkins_home/casc.yaml
8. USER jenkins
```

When a container image is created, Docker uses the image declared in the FROM directive as an input, creating a new writable layer, the only **writable layer**, on top of all the image layers combined via the Dockerfile through the storage driver. *Figure 4.2, Dockerfile Layers*, shows how the container layer stack is built:

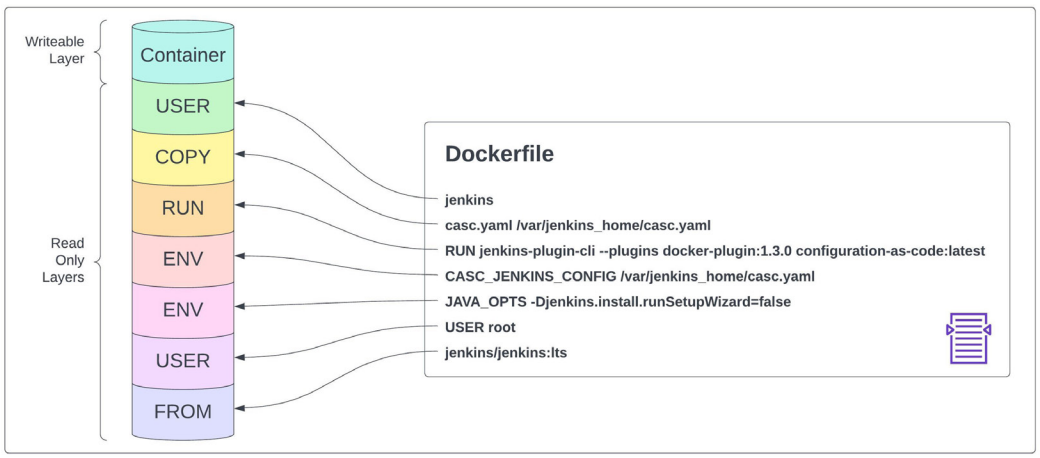

***Figure 4.2**: Dockerfile Layers*

https://t.me/learningnets

The main difference between containers and container images is that **very writable container layer**.

Building file configuration

With regard to secure image building process, one of the most common and the most dangerous vulnerability is secret management when it becomes part of the image building process. A common error when dealing with secrets during application development is the copy and remove (**COPY / RM)** approach. The following example code is based on a simplification of the Jenkins custom image build, but the Jenkins image has been replaced with the Ubuntu image, which is much lighter in terms of disk space:

```
1. # syntax=docker/dockerfile:1
2. FROM ubuntu:22.04
3. COPY the_secret_file /root/the_secret_file
4. RUN rm /root/the_secret_file
```

As explained in *Figure 4.2, Dockerfile Layers*, the build process will layer the three steps above executing the COPY command first, and then it will execute the **RUN** command to remove the **secret** file, but both layers would be added to the final image, as the following output confirms:

```
1. $ docker build –no-cache -t this-is-not-a-safe-image:latest .
2. [+] Building 1.6s (13/13) FINISHED
3. …
4. OMITTED
5. ...
6.  => CACHED [1/3] FROM docker.io/library/ubuntu:22.04          0.0s
7.  => [2/3] COPY the_secret_file /root/the_secret_file          0.0s
8.  => [3/3] RUN rm /root/the_secret_file                        0.2s
9.  => exporting to image                                        0.0s
10. => => exporting layers                                       0.0s
11. => => writing image sha256:5c40b3ccbedc31c15873bf-
    ba264e14514a678c65d785f819c365208fd4afee43                   0.0s
12. => => naming to docker.io/library/this-is-not-a-safe-image:latest
```

https://t.me/learningnets

The result of the building process is a container image that should not present the **secret** file in the final layer; running the container makes it possible to verify that indeed there is no file in the root folder:

```
$ docker run -it 5c40b3ccbedc

root@33034062fb29:/# ls root/

root@33034062fb29:/#
```

To demonstrate that the Docker storage driver is adding up layers into the final image, and that the secret file is still stored inside the newly created image, a simple verification methodology involves extracting a compressed **tar** file of the image.

```
$ docker save this-is-not-a-safe-image -o this-is-not-a-safe-image.tar

$ tar xf this-is-not-a-safe-image.tar
```

By listing the folder in which the **this-is-not-a-safe-image.tar** file has been extracted with the **tar** command, we should find something like the following (note that the output of the **ls** command has been modified slightly to accommodate publishing requirements):

```
$ ls -l

total 140320

drwxr-xr-x  063d9c2e40c64f3012e236e946d55aef47f80cae9584da0c-d0e36417628e77b6

drwxr-xr-x  2d2784d5d10388724011f34336b29c9e834d5318b611d-d1143a4e1af7a80671c

drwxr-xr-x  59679b05a55027190f7d4e06290c9ab5d63ff34eb62be-25f053807eec333acf4

-rw-r—r--   5c40b3ccbedc31c15873bfba264e14514a678c65d785f-819c365208fd4afee43.json

-r—r--r--  manifest.json

-r—r--r--  repositories

-rw-------  this-is-not-a-safe-image.tar
```

The **manifest.json** file is the high-level overview of the structural image creation; it contains the final sha256 digest assigned to the image and the layers digest assigned at each of the command of the Dockerfile. The details of the building operations are stored in the JSON file named as the final Docker image digest. As readers may note, the layers mentioned in the **manifest.json** file have been extracted as well in the

https://t.me/learningnets

form of folders; each folder contains a **layer.tar** compressed file, which snapshots the Linux folders tree of the original container image declared in the **FROM** directive at the time of the correspondent command listed in the Dockerfile.

The Dockerfile contains three commands: **FROM**, **COPY**, and **RUN**; therefore, there are three layer folders extracted. It is safe to assume that one of the extracted **layers.tar** still contains the secret file. Looping inside all the layers can help verify the assumption:

```
$ for secret in */layer.tar; do tar -tf $secret | grep the_secret_file && echo $secret; done
root/.wh.the_secret_file
063d9c2e40c64f3012e236e946d55aef47f80cae9584da0cd0e36417628e77b6/layer.tar
root/the_secret_file
59679b05a55027190f7d4e06290c9ab5d63ff34eb62be25f053807eec333acf4/layer.tar
```

The interesting thing to note in the output is the **.wh.** suffix. In Linux systems, the **.wh.** suffix to a file or folder is the indication that the file or folder has been deleted from an upper layer of the filesystem by the **OverlayFS** mechanism. When a file has been marked for deletion, it is not actually removed from a filesystem; instead, it is replaced by a **without** file to testify that the file is no longer needed. This corresponds to the **RUN** command in the example Dockerfile provided earlier. It is clear, then, that the digest at point 5 corresponds to the **COPY** command of the Dockerfile; this is verifiable by attempting to extract the secret from the **layer.tar** file, which is indeed still stored in the final image.

```
$ tar xf 59679b05a55027190f7d4e06290c9ab5d63ff34eb62be25f053807eec333acf4/layer.tar root/the_secret_file
$ cat root/the_secret_file
my_password
```

Docker has developed a kit that adds other capabilities to the build process, called **Buildkit**. Among other performances improvements, the interesting flag is **--secret**. This flag can safely provide a secret to the Dockerfile at the build time, mounting **tmpfs** and storing the secret in **/run/secrets** temporarily. This feature guarantees that no secrets will be left in the image inadvertently. To enable **Buildkit**, the Dockerfile must have a syntax line commented out at the beginning of the file, with minimal version of 1.2:

```
# syntax=docker/dockerfile:1.2
FROM ubuntu:22.04
```

https://t.me/learningnets

```
3. RUN --mount=type=secret,id=the_secret_file,dst=/root/.aws/credentials
   # Launch any aws command
```

As someone might have noted, the previous code adds the **dst** flag to the mount command for the secret, which means the secret will be stored in a custom location; when the **dst** flag is missing, **Buildkit** will assume that the secret will be in **/run/secrets**. In order to enable Buildkit building, the **DOCKER_BUILDKIT** environment variable must be declared:

```
1. $ DOCKER_BUILDKIT=1 docker build --no-cache \
2.     --secret id=the_secret_file,src=the_secret_file  \
3.     -t safe-image:latest .
```

The **--secret** flag is passed via **build**, and it is indicating the secret **ID** and location (**src**). The output of the **build** command is as follows:

```
1. [+] Building 1.5s (9/9) FINISHED
2.  => [internal] load build definition from Dockerfile           0.0s
3.  => => transferring dockerfile: 160B                           0.0s
4. ...     OMITTED
5.  => CACHED [1/2] FROM docker.io/library/ubuntu:22.04           0.0s
6.  => [2/2] RUN --mount=type=secret,id=the_secret_file,dst=/root/.aws/
   credentials                                                    0.2s
7.  => exporting to image                                         0.0s
8.  => => exporting layers                                        0.0s
9.  => => writing image sha256:6ee02422e4bc8f792f019c06df5a9e63cc0d-
   65c6ea35f32fc0bd7ecbdb51ddd6
```

What has changed with the **Buildkit** approach is that the Docker build command, when interpreting the **Dockerfile**, has acknowledged the presence of the **mount** line, and has, therefore, treated the secret as a flag to be used during the build process (line 6).

This is a secure approach in handling secrets during the building process, because there will be no trace of the secret inside the container after the image has been built. The verification can be executed by running the newly created container and checking the presence of the **credentials** file in the **root/.aws** folder.

```
1. $ docker run -it 6ee02422e4bc8f792f019c06df5a9e63cc0d65c6ea35f32f-
   c0bd7ecbdb51ddd6
2. root@59b4dc46b075:/# ls -la root/.aws/
```

https://t.me/learningnets

```
3. total 8
4. drwxr-xr-x 2 root root 4096 Apr  2 16:21 .
5. drwx------ 1 root root 4096 Apr  2 16:21 ..
```

The verification can also be processed by extracting the tar compressed file as per the previous example with the not safe image and looping inside the **`layers.tar`** files:

```
1. $  docker save safe-image -o safe-image
2. $ tar xf safe-image.tar
3. $ for secret in */layer.tar; do tar -tf $secret | grep creden-
   tials && echo $secret; done
```

The newly built **safe-image**, in contrast with the previous **this-is-not-a-safe-image**, has no inspecting layers therefore it is not possible to retrieve the content of the secret file.

Multi-stage builds

There are cases where the Dockerfile is an extreme complex configuration file, especially when the file contains specific service configurations parameters. It can include the base image specifications, the user configuration, any update or upgrade needed, service configuration parameters, the TLS certificate configuration, and secrets management, etc..

As illustrated in the previous section, any line in a Dockerfile is a new layer in the image, so the number of layers an image contains is directly proportional to the number of instructions a Dockerfile contains. With a complex Dockerfile, it becomes quite difficult to keep the build size as small as possible, resulting in a complex process that can potentially increase the size and attack surface of the image. A large deployment brings into context several intermediate processes, tools and artifacts. Refer to the following figure:

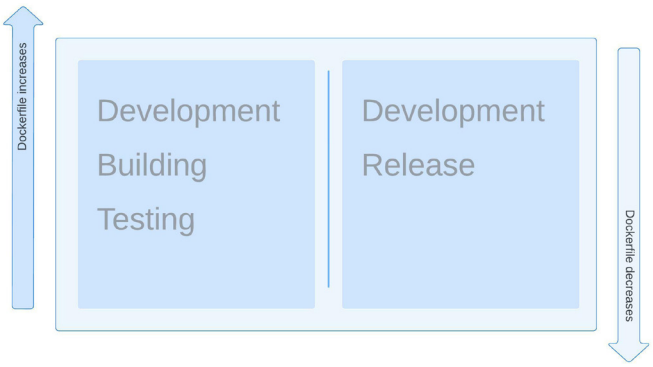

***Figure 4.3**: Dockerfile Weight*

https://t.me/learningnets

Figure 4.3, Dockerfile Weight, shows some of the phases a Dockerfile could be subjected to. Thinking of a Dockerfile from a development process standpoint, where ideally everything that is needed for building the application is considered within the building process, is opposed to a Dockerfile that contains only the application and what is needed to execute it. **Multi-stage builds** are Docker's answer to this specific issue, aiming to optimize Dockerfile itself and creating intermediaries' steps between the initial and the final image, providing better control over the files and the artifacts that a container image will contain.

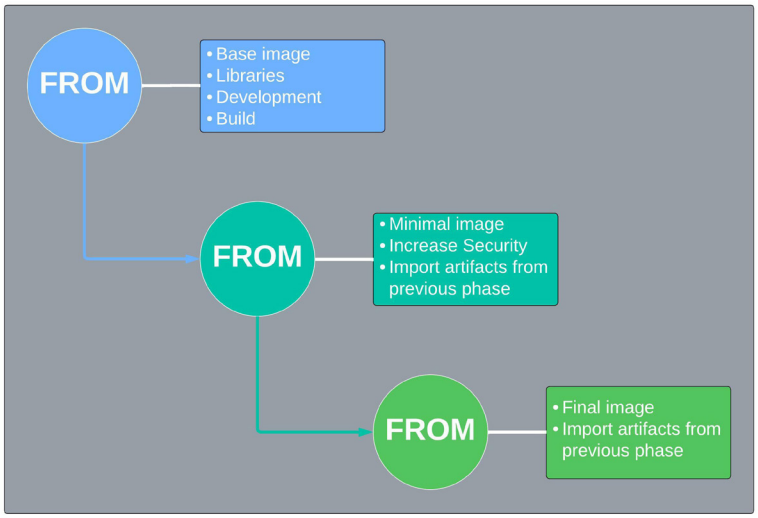

***Figure 4.4**: Nested FROM*

The nested selected and progressive creation methodology helps in reducing the attack surface, as illustrated in *Figure 4.4, Nested FROM*. It is very common to use different Dockerfile, one for each phase of the building process, usually a **development** Dockerfile, then a **build** Dockerfile, and finally a **deployment** script. This approach to the containerization process brings a series of layers as part of the **previous** container image, which are not strictly necessary for the purpose of simply running the final binary file, and it also introduces the concept of **base image**, where a container image is treated as the base for multiple environments or purposes:

```
1. # syntax=docker/dockerfile:1
2. FROM jenkins/jenkins:lts AS base
3. USER root
4. ENV JAVA_OPTS -Djenkins.install.runSetupWizard=false
5. RUN jenkins-plugin-cli --plugins docker-plugin:1.3.0
6. USER jenkins
```

https://t.me/learningnets

```
7.
8.  # Use the base Jenkins image to build the matrix plugin
9.  FROM base AS development
10. USER root
11. RUN jenkins-plugin-cli --plugins configuration-as-code:latest
12. USER jenkins
13.
14. # Use the development image to build the production image
15. FROM development AS production
16. USER root
17. ENV CASC_JENKINS_CONFIG /var/jenkins_home/casc.yaml
18. RUN jenkins-plugin-cli  --plugins job-dsl:latest
19. COPY casc.yaml /var/jenkins_home/casc.yaml
20. USER jenkins
```

The previous example code re-think the Dockerfile used in *Chapter 3, Container Stack Security*, in the *Secure CI/CD* section for a multi-stage purpose, defining three steps: base, development, and production, as multiple **FROM** directives. The building process can be initiated through Buildkit and can be selectively stopped at the desired build stage using the **--target** directive.

```
1. $ DOCKER_BUILDKIT=1 docker build --no-cache \
2.     -t multi-image:development --target development .
```

Each **FROM** directive can use a different base image, and each **FROM** initiates a new phase of the build process, where artifacts can be selectively copied from one phase to another, discharging everything that is not needed for the final phase.

Minimal and distroless images

The layering of the container images makes use of what is known as the **copy-on-write** (**CoW**) strategy, essentially a strategy of copying and sharing layer for maximum efficiency and sizing containment. The principle is that if a file or folder already exists in a lower layer of the image, it will use that very layer; a new layer is only created when that file or folder is modified. Container images should indeed be considered ephemerals, if a change is needed, a new image is generated and the current one is destroyed.

https://t.me/learningnets

Although the copy-on-write strategy helps in reducing disk I/O (Input/Output), consequently reducing the disk space occupied, developers should consider creating their own image from scratch to have full control on its contents. Despite the obvious security advantage of such approach, in the cloud age, where everything is ephemeral, it is hard to imagine developers creating their own machine, stripping out all the not-needed packages and libraries, and create their own image. It's time consuming and does not fit well in a constant high-paced software development life cycle. It also requires sysadmin skills to resolve dependencies in providing a stable environment.

Figure 4.5, Image Evolution, shows the container image evolution path:

- The **base** image is standard image, it can be also referred to as the DVD iso image, containing all the standard packages of a modern Linux distribution.
- The **minimal** image type is a tentative to reduce at the minimum necessary the image size of the standard image.
- The **micro** or **distroless image** aims to provide a smaller footprint in term of size compared to the minimal image, and packaging only the necessary libraries to execute the application binary which it is meant for.

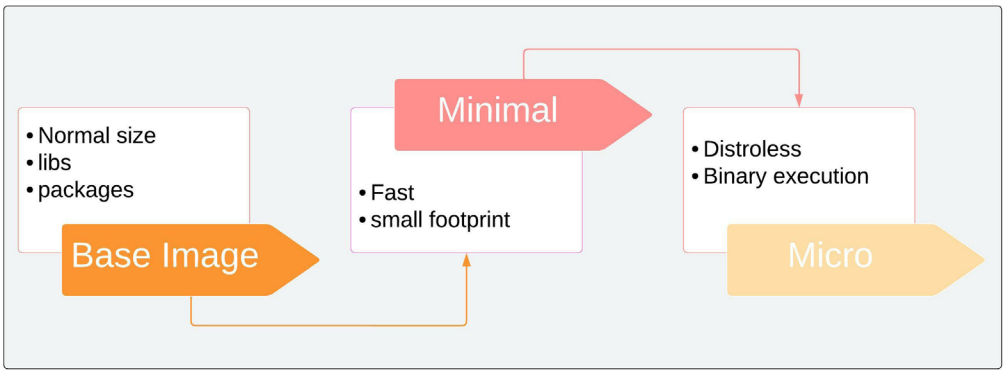

***Figure 4.5**: Image Evolution*

Distroless is, therefore, a language-oriented container system rather than Linux distro flavored. Linux distros, such as RedHat, Debian, Ubuntu, and CentOS, have published ad-hoc container image versions of the respective releases, the so-called **minimal**. Minimal is a specific type of container image designed for automation and deployment at scale, made available through the most common registry channels like DockerHub, Quay or AWS. The **distroless** project is one of the many Google Container Tools projects available, and the difference between **minimal** and **distroless** is quite interesting, and it is at the center of an interesting discussion in the community. RedHat has created a set of container images available without a RedHat subscription, the UBI that stands for **Universal Base Image**, in four different

https://t.me/learningnets

flavors: standard, init, minimal and micro. *Table 4.1, UBI Images*, shows the main differences between the four flavors:

Type	Description
Standard	• init system: all systemd features • yum: standard set of commands • utilities: tar, dmidecode, gzip, and so on
Init	• systemd initialization system • ps and included process commands
Minimal	• microdnf • 32 M image size
Micro	• like distroless images

***Table 4.1**: UBI Images*

In terms of size, minimal and micro are not far away; for instance, the **Minimal Ubuntu** claims to be just 29 MB (Megabyte) for the 18.04 LTS (**Long Term Support**) version, which is remarkable on its own, but the **Google Distroless** based on Debian 11, available at gcr.io/distroless/static-debian11 (you would need access to Google Cloud to verify such a URL), claims to be only 2 MiB (Mebibyte).

In terms of security, the smaller the footprint, the smaller the attack surface. If an attacker gains access to the container, the best practice is to provide as few tools as possible. *Table 4.2, Tools to avoid*, shows a minimal list of tools that should be definitively avoided in a production environment:

Type	Description
Package Managers	yum, apt, zipper
Network tools	curl, ssh, netcat, wget
Shells	bash, zsh, ksh
Compilers	gcc, aocc, ispc, babel, free pascal, javac, Gc, pypy
Debuggers	gdb, xdb, dbx, ghidra, bashdb, strace, xdebug, jpda

***Table 4.2:** Tools to avoid*

A quick digression brings to light the fact that not many know that the **United States National Security Agency** (**NSA**) has made the Ghidra project publicly available.

Note: Ghidra is an open-source, reverse engineering framework, a full high-end software featured analysis tool including assembly, scripting, decompilation, disassembly, and graphing.

https://t.me/learningnets

Among the minimal image types, **Alpine** made a name for itself recently. The Alpine distro is a minimal security-oriented container image designed to compile all binaries running in the user space as **Position Independent Executables** (**PIE**). The PIE are considered the output of hardened build processes, meaning that a PIE binary executable and all its dependencies are executed in random locations of the virtual memory each time the application is executed.

The specific purpose of such a solution is to oppose one of the most common techniques of **stack smashing attack,** such as **stack buffer overflow**, or its more recent and advanced computer security exploit technique: the **Return Oriented Programming** (**ROP**). In software programming, the stack buffer overflow happens when an application writes to a memory address on the application's call stack outside the intended data structure, in other words, when the application writes more data to a buffer than what is actually allocated, generating an **overflow**. The overflow can be generated intentionally as part of the stack smashing attack.

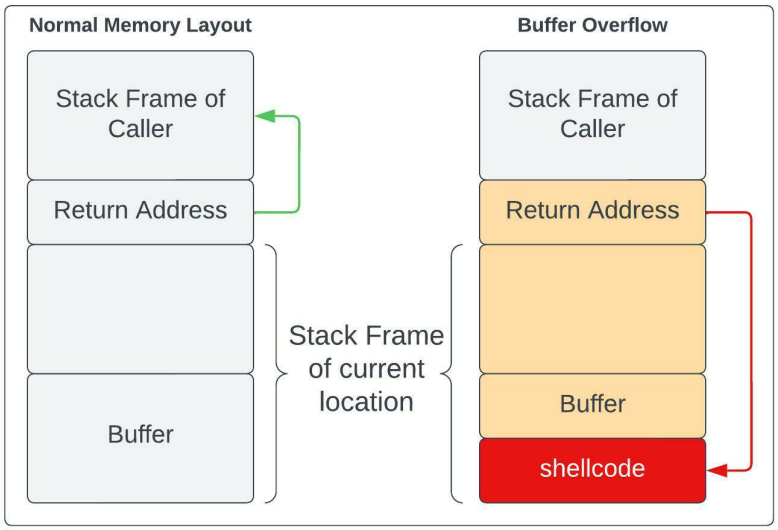

***Figure 4.6**: Stack Buffer Overflow*

In a standard buffer overflow attack, the attacker would write the attack code into the stack and overwrite the return address in a different location of the memory, injecting its payload with new instructions, as shown in *Figure 4.6, Stack Buffer Overflow*. With the ROP attack, the attacker obtains control of the application call stack to inject machine instructions into memory to perform arbitrary and malicious operations for the purpose of invalidating executable space protection systems.

When **Data Execution Prevention** is enabled, the attacker cannot return the payload to another area of the memory because it is marked as non-executable (note that non-executable is different from non-writable), so the attacker uses machine instructions called **gadgets**, such as loops and conditional branches, to build the

https://t.me/learningnets

kind of payload needed for the attack, thanks to valid instructions. **Return Oriented Programming** is one the techniques used in penetration tests. The **Metasploit** team at **Rapid7** explains **ROP** with a quick video at **https://www.rapid7.com/resources/rop-exploit-explained/**, it has also released the Metasploit framework, which has an interesting lab for exploitation and enumeration of Kubernetes clusters. The available Metasploit modules can run via a compromised Docker container or via API, externally to the cluster.

The documentation to run a Kubernetes penetration test is available at **https://docs.metasploit.com/docs/pentesting/metasploit-guide-kubernetes.html**. The following is a quick list of the available Metasploit penetration test modules, in *Table 4.3, Metasploit Modules*:

Type	Description
HTTP + HTTPS	Against port 80 and port 443 encrypted via TLS
Kubernetes	Enumerating: version, auth, namespaces, pods, secrets
MySQL	Against port 3306; Enumerating: version, bruteforcing credentials, dumping database, executing arbitrary queries and SQL queries, reverse shell access
PostgreSQL	Against port 5432; Enumerating: version, bruteforcing credentials, dumping database, capture server, executing arbitrary SQL queries, reverse shell access
SMB	Against port 139 with NetBIOS and 445 without NetBIOS
SSH	Against port 22 with support for tunnelling traffic
WinRM	Against port 5985 for HTTP and 5986 for HTTPS

Table 4.3: *Metasploit Modules*

Distroless containers, despite providing a certain level of security, are not immune to attacks. The focus on the image size could lead to a misinterpretation of what really matters from the security standpoint in any given Linux operating system. Here are a few considerations:

- Attack surface is not measured in megabytes
- Distroless does not mean no operating system

The real attack surface is not just the number of files a container image stores or how many megabytes of disk space the image occupies; rather, it is a combination of several analytic considerations, as *Figure 4.7, Attack Surface* illustrates:

https://t.me/learningnets

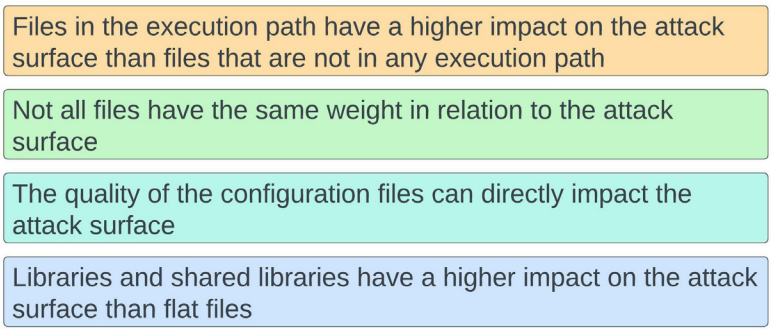

***Figure 4.7**: Attack Surface*

Regardless of how many packages, libraries, and generic software or files can be excluded from a Linux distro, there will always be a **kernel** and a **user space** to handle. Even the distroless project by Google relies on the **Debian** Linux distro for its user space requirements. Furthermore, the application deployed inside the container image would carry additional weight in relation to the specific programming language.

The **Center for Internet Security** (**CIS**) is one of the most recognized **Information Technology** (**IT)** communities worldwide, and it aims to provide guidelines to implement security best-practices. CIS has created and released several free tools for the only purpose of increasing cybersecurity readiness. It is mandatory mentioning version 8 of the CIS Security Controls (previously known as the SANS Top 20), available at **https://www.cisecurity.org/controls/cis-controls-list**; however, this chapter focuses on the suite of the CIS Benchmarks, a wide collection of documented industry security best practices, spanning across securing system configurations, securing network and applications, across several technology categories.

The CIS Benchmarks can be used as standard to map security controls for achieving compliance with regulatory frameworks like **Health Insurance Portability and Accountability Act** (**HIPAA**, **Payment Card Industry Data Security Standard** (**PCI DSS**), **International Organization for Standardization** (**ISO**). From these over 140 CIS Benchmarks, it is worth mentioning the following:

- CIS Hardened Images - **https://www.cisecurity.org/cis-hardened-images**
- CIS Docker - **https://www.cisecurity.org/benchmark/docker**

The **CIS Benchmarks Hardened Images** is a set of security best-practices applied to virtual machines aiming to generate hardened virtual image. Those images are either distro related such as Debian Linux, Ubuntu Linux, CentOS Linux, Amazon Linux, Apple macOS, Microsoft Windows Server or application related such as NGINX, PostgreSQL and so on and so forth, released on a monthly life cycle. The **CIS Benchmarks Docker** is a set of security best-practices applied to the Docker

https://t.me/learningnets

Engine system and to the Container Images and Build File system as part of the hardening process, as shown in *Table 4.4, CIS Container Images Hardening:*

CIS Benchmark Docker Container Images
Use non-root user
Container must use only trusted images
Remove unnecessary packages
Scan and rebuild images to includes patches and updates
Enabled Content Trust
Enable HEALTHCHECK
Do not use update in Dockerfile
Remove setuid and setgid
User COPY rather than ADD
Do not store secrets in Dockerfile
Use only verified packages
Validate artifacts

Table 4.4: *CIS Container Images Hardening*

The CIS Benchmark Docker has been also adopted by the DevSec Project, which aims to create a standard for running secure infrastructure. The idea is to create a common secure hardened layer for operating system and services via automation, essentially, a hardening framework. Readers aiming to enhance their knowledge of security best practices applied via tools like Ansible, Chef or Puppet can visit **https://dev-sec.io**. There is also a **CIS Kubernetes Benchmark**, but this topic will be discussed in *Chapter 7, Kubernetes Hardening*.

Scanning and verifying images

Images built with security vulnerabilities become vulnerable as soon as the container is created. Those vulnerabilities are enabled at the runtime. When integrating the image scanning mechanism into the **Continuous Integration** (**CI)** pipeline, the build should be permitted only if the security requirements are respected as part of the **shift-to-the-left** approach, where vulnerable images should never be pushed to the production environment.

As simple as this topic could appear, image scanning is not perfect science. The market offers various proprietary and open-source image scanners with different levels of coverage: some image scanners check only operating system packages, while some can also scan runtime libraries for programming languages, and others could even provide deeper additional features like binary fingerprinting or file content testing.

https://t.me/learningnets

Table 4.5, Scanning Feature, explains in detail the features that should always be considered to make image scanning as effective as possible:

Type	Description
App	The scanner should be able to verify the app binary language.
Supporting Language	The scanner should be able to scan the app's libraries installed to support the app binary.
Scanning Agent	If the scanner needs an agent, this should be part of the update life cycle within the CI.
Operating System	The scanner should be able to effectively scan the operating system, regardless of the attack surface the image is exposed to.
Acceptable Risk Level	Defines the accepted level of risk with a technical threshold calculated on the severity of the detected vulnerabilities.

***Table 4.5**: Scanning Features*

In this scenario, it is hard to propose a specific tool that could satisfy every possible use case; indeed, there is no such solution, as container image scanning needs may vary from organization to organization, and more often, from team to team.

Historically, the most common open-source code quality platform is **SonarQube** by SonarSource, it can scan code for vulnerabilities in almost thirty programming languages. There are several competitors worth mentioning, such as Veracode, Qualys WAS (Web Application Scanning), Rapid7 InsightAppSec, Tenable Web App Scanning, Snyk Code, and others.

A code quality platform is considered an external system to the container image scanning platform, and it often creates overhead because it is mostly related to the code quality control and a tool that a developer would feel comfortable with. To reduce the gap between developer and operations, and promote DevSecOps best-practice, SonarQube can create a project directly from a source control system like **GitHub**, **GitLab**, **Bitbucket**, and **Azure DevOps**.

In contrast, the source control systems GitHub and GitLab have integrated code quality system to enhance secure code delivering. **GitHub Security** is language-oriented, with support for up to 11 programming languages, along with features like Secret scanning, Dependabot, and Supply Chain Security. On the other hand, **GitLab Application Security** offers several security tools, including **Static Application Security Testing** (**SAST**), **Dynamic Application Security Testing** (**DAST)**, Dependency Scanning (also known as **Software Composition Analysis**), **Infrastructure as Code** (**IaC) Scanning**, Container Scanning, Secret Detection, API

https://t.me/learningnets

Fuzzing, and Coverage Fuzzing. Many of the tools GitLab Security provides need the **GitLab Runner** agent installed as a pre-requisite, which should be considered as additional executable within the secure update life cycle.

The GitLab Security suite covers all the aspects exposed in *Figure 4.7, Attack Surface*, but like many other DevSecOps solutions, it comes at a price in an enterprise or commercial tier. Among the open-source projects, there are several solutions that can be adopted to achieve a very good security posture and fulfil several container image security aspects, like **Clair, XRAY**, **Anchore, Falco**, and the verification system **Notary**. *Figure 4.8, Image Scanners*, shows the main purpose for each one:

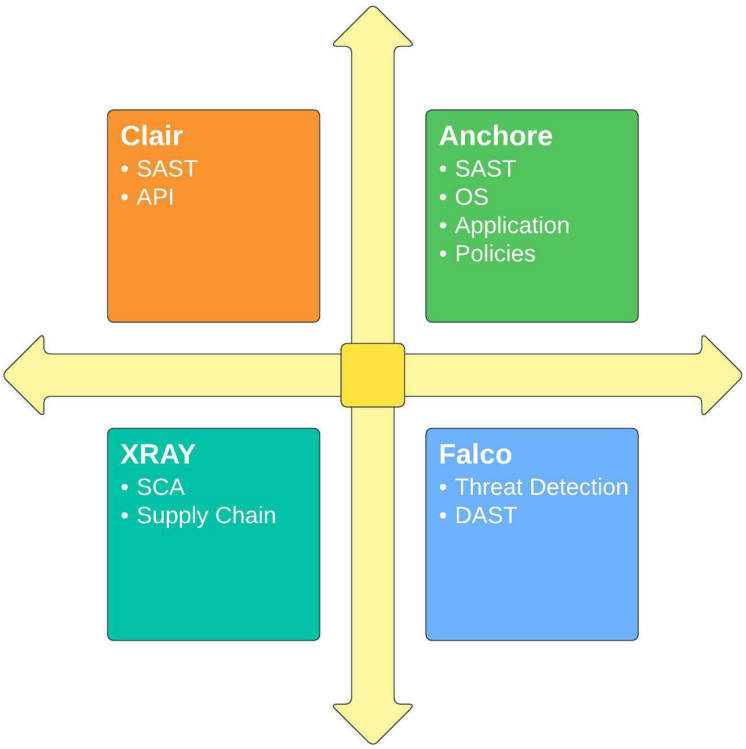

***Figure 4.8**: Image Scanners*

Clair is probably the most famous open-source **Static Application Security Testing (SAST)** tool available online today; it can scan either **Docker** or **Open Container Initiative** (**OCI**) application containers, and it is the scanning engine behind **Quay.io**, the RedHat Container Registry. RedHat OpenShift uses **Clair** for container security natively. **Clair** uses several vulnerability data sources like RedHat Security Data, Ubuntu CVE, and Debian Security Bug Tracker.

JFrog **XRAY** is an open-source artifact and dependencies analysis tool. Its main feature is to provide **Software Composition Analysis** (**SCA**) methodologies, natively integrated with the famous JFrog Artifactory system. This is not a trivial

https://t.me/learningnets

statement; for instance, a supply chain attack was used in late 2020 to compromise the **SolarWinds** software build system. A supply chain attack is a cyberattack aiming to damage one or more organizations by attacking the less secure elements in their supply chain.

The **SolarWinds** cyberattack was perpetrated via a supply chain attack having the target of SolarWinds IT infrastructure, which includes many US federal governments bodies. The case, then rebranded **UNC2452**, brought the Cybersecurity and Infrastructure Security Agency (**CISA)** to release **Sparrow**, a free tool for detecting potential malicious activities in any Microsoft Office 365 or Azure environment, pushing the US government administration to provide new guidelines and regulations. The National Institute of Standards and Technology then released the Software Bill of Materials (SBOM), a new standard for detailing supply chain components relationship as part of the **NIST SP 800-161r1**.

Anchore is a complete ecosystem of security solutions for scanning container images. The open-source portion of the Anchore framework consists of two main projects:

- **Grype** is a powerful tool for vulnerability scanning:
    - It can scan container, container images and filesystems.
    - Language-specific scanning such as Ruby, Golang, PHP, Rust, Java, .Net, Python, and JavaScript.
    - It works with Docker systems, **Open Container Initiative** (**OCI**) systems and **Singularity** image formats.
- **Syft is** a command line interface tool for creating Software Bill of Materials reports directly from container images; it can generate a complete mapping of the application's dependencies in respect of the **NIST SP 800-161r1**.

Note: Singularity is an open-source container platform designed with security in mind for systems running in High Performance Computing (HPC) environments. It provides out-of-the-box features like immutable single-file image format with crypto signatures and encryption and user consistency. At the hardware level works with GPUs accessible hardware, parallel filesystems on clusters and high-speed networks.

Falco was originally created by **Sysdig**, and then it became an incubator for the **Cloud Native Computing Foundation** (**CNCF**). It is essentially a container runtime security tool that can be executed in containers or PODs, with the intent of intercepting kernel events and applying governance policies, in breach of which Falco will generate alerts with a correspondent severity.

Falco is designed to detect Linux syscalls like shell running within a container, containers running in privileged mode or mounting specific path from the host, and non-device written to **/dev**. *Figure 4.9, Image Scan Automation*, illustrates a classic

https://t.me/learningnets

application of a container image scanner applied to a **Continuous Integration Continuous Deployment** (**CI / CD**) pipeline. Most of the tools mentioned so far can be easily integrated in any CI/CD pipeline, but trivy by Aqua Security excels for the ease of integration in highly automated environments.

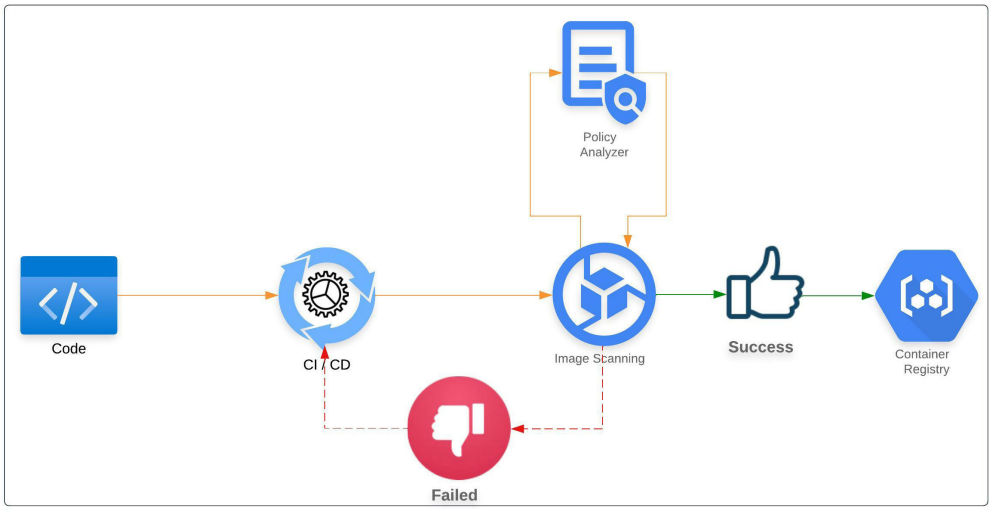

***Figure 4.9**: Image Scan Automation*

Trivy is probably the most comprehensive open-source security scanner today available, it includes features like the following:

- Vulnerability detection for various base images, including **distroless**.
- Docker, **Open Container Initiative** (**OCI**), Podman, containerd container image
- Language-specific scanning packages like Composer, Pipenv, npm, yarn, Bundler, Maven, Go, Cargo, Poetry and NuGet
- SBOM support
- IaC (Infrastructure as Code) scanning, including **Terraform**
- Secret scanning
- Policy and exception framework
- Works with **Open Policy Agent** (**OPA**), **https://www.openpolicyagent.org/**
- Works with Jenkins, GitLab, GitHub Actions, CircleCI, Travis CI, Azure, AWS Code Pipeline, Bitbucket Pipelines, AWS Security Hub
- And of course, it works with pods and Kubernetes

https://t.me/learningnets

- **Istio**, the most popular Kubernetes service mesh, has adopted **trivy** recently

Notary has been donated to the **Cloud Native Computing Foundation** (**CNCF**) in 2017 by Docker. **Notary** is a container verification content system, based on **The Update Framework** (**TUF**) concept, aiming to create a secure software updating system by enabling protection against attackers that could potentially compromise repositories or signing keys. To read more about TUF, visit **https://theupdateframework.io**. The key advantages of the Notary system are detailed in *Table 4.6, Notary Features*:

Type	Description
Survivable Key Compromise	Signing Key Management mechanism
Freshness Guarantees	Timestamping on publishing
Configurable Trust Thresholds	Content with multi-signature mechanism
Signing Delegation	Publishing delegation
Use of Existing Distribution	Trust can be added to existing content
Untrusted Mirrors and Transport	Metadata mirroring

***Table 4.6**: Notary Features*

A deeper approach on the image signing mechanism and its interaction with container registries is explained later in the chapter.

Private and public registries

When all the security requirements related to the container image are in place, it is time to run the building pipeline and push them to the registry. A container registry is essentially a repository or a group of repositories used to store container images. There are two types of registries:

- Private registries are enterprise container image storage system, often providing advanced security features.
- Public registries are publicly available, and they are used most often by professionals or small teams with no resources to create their own registry.

Alongside the three major public cloud provider container registry services, i.e., **Google Container Registry**, **Amazon Elastic Container Registry** (**AWS ECR**), and Microsoft **Azure Container Registry**, **DockerHub** is the most popular public container image registry.

Chapter 03, Container Stack Security, in the section **Secure Connection** has explained how to enable **Transport Layer Security TLS** connection for the Docker Engine system, and how the same secure communication could be applied to the Docker API system when exposed over TCP connections.

https://t.me/learningnets

In a recent study, **TrendMicro** demonstrated how **DockerHub** registry could be exposed to exploit via credentials leaking. The TrendMicro team uploaded to DockerHub two honeypots (honeypot is a controlled vulnerable computer environment), which were the targets of exploitation attempts several times in less than a month, for the purpose of deploying malicious Docker images containing **rootkits**, **Kinsing** malware, credentials stealers, **XMRig** Monero cryptocurrency miners, Docker escape kits and Kubernetes exploit kits. The full article is available at **https://www.trendmicro.com/en_gb/research/22/i/security-breaks-teamtnts-dockerhub-credentials-leak.html**.

The downside of public registries is that there is limited control on the quantity and quality of security requirements that can be used to achieve a sufficient security posture, and the truth is that unless those requirements are somehow enforced by whoever provides the services, they are often overlooked. DockerHub can be re-created locally, in any private network, and can be secured through different advanced methodologies. Because the Docker registry does not accept credentials provided in clear text, it is necessary to create a password file. The suggestion is to create an additional folder named **auth** to store the **htpasswd** file:

```
$ mkdir auth
$ docker run --entrypoint htpasswd httpd:2 -Bbn luigi mypass-
  word > htpasswd
```

To create a container using the htpasswd mechanism, the following Docker command helps in spinning up the local system:

```
$ docker run -d \
  -p 5005:5005 \
  --restart=always \
  --name my-local-registry \
  -v "$(pwd)"/auth:/auth \
  -e REGISTRY_HTTP_ADDR=0.0.0.0:5005 \     # Avoid conflicts on Mac
  -e "REGISTRY_AUTH=htpasswd" \
  -e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Realm" \
  -e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd \
  registry:2
```

https://t.me/learningnets

Then, it is possible to log in to the local registry by simply running the Docker **login** command:

```
1. $ docker login localhost:5005
2. Username: luigi
3. Password:
4. Login Succeeded
```

With the local Docker registry up and running, it is possible to **push** and **pull** container images, as we would do with any container registry.

The obvious security advantage of such solutions, even with basic authentication methodology, is that all the images are stored locally, with greater control on what the registry is handling. The local Docker registry would be empty in the beginning, so it is possible to create a new image via Dockerfile or pull a test image from the public DockerHub and then tag and push it against the local registry:

```
1. $ docker pull alpine
2. $ docker tag alpine localhost:5005/my-local-alpine-image
3. $ docker push localhost:5005/my-local-alpine-image
```

Subsequently, we would work locally with the images by pulling them from the local registry.

```
1. $ docker pull localhost:5005/my-local-alpine-image
```

Non-cloud systems historically in the market as excellent alternatives to DockerHub are **Sonatype Nexus Repository**, RedHat **Quay.io**, **JFrog Container Registry**, and **GitLab Container Registry**. Among those solutions, each of them proposing a free tier or community edition with less or more features, JFrog Container Registry stands out because it can handle both container images and helm charts for Kubernetes. From the security standpoint, JFrog Container Registry helps in automating security within the build pipeline, while GitLab Container Registry has adopted a token methodology to authenticate users against the registry, eventually leveraging **Multi Factor Authentication**. (**MFA**) as well.

More recently, another project promoted by the **Cloud Native Computing Foundation** (**CNCF**) with the intent of providing a trusted registry came to light: **Harbor**. It is an open-source trusted container and helm charts registry that stores, scans and signs cloud native artifacts. The peculiarity of such a system is the added

https://t.me/learningnets

value of the security features that so far seem unique only to this registry, such as identity management, access control, and auditing. The list of security capabilities for Harbor contains **Role Based Access Control RBAC**, LDAP and Active Directory integration, OpenID Connect support, the native integration of **Notary**, discussed in the previous section, a full logs auditing mechanism, and a vulnerability scanning system.

Registry authentication

The deployment of the local registry as it is, works with basic authentication, so it is widely available on the local network and, of course, is exposed to attack. There are three ways to enable authentication security, and each method has an increased level of difficulty and security:

- Add authentication via user, as demonstrated in the previous section
- Use Nginx as authentication proxy
- Add a **Transport Layer Security** (**TLS**) certificate

Basic authentication is often not sufficient in working environments that are not **small office home office** (**soho)** and that may require enterprise authentication mechanism like **Lightweight Directory Access Protocol** (**LDAP)** or **Active Directory** (**AD)** or **Single Sign On** (**SSO**). With the following methodology, a reverse nginx proxy is deployed in front of the Docker registry that implements authentication. For the purpose of this exercise, we will use the key and certificate created in the previous chapter, where we enabled the Docker API Engine secure communication over TLS.

Note: Although a self-signed certificate is a more secure communication mechanism than a basic authentication system based simply on passwords, the TLS certificate should be signed by a Certificate Authority to achieve stronger security posture.

The **auth** folder should already be in place if readers have completed the Docker registry basic authentication explained in the previous section. Additionally, a **data** folder, a **nginx.htpasswd** file, the **nginx.conf** file, **domain.crt** certificate, the **domain.key** key files, and the **docker-compose.yml** file are needed to run the container stack that deploys the registry and the nginx container:

```
1. $ mkdir -p auth data
2. $ docker run --entrypoint htpass-
   wd httpd:2 -Bbn luigi mypassword > auth/nginx.htpasswd    # “mypass-
   word” for the docker login
3. $ cp client-cert.pem auth/domain.crt
4. $ cp client-key.pem auth/domain.key
```

https://t.me/learningnets

```
$ touch auth/nginx.conf
$ touch docker-compose.yaml
```

Add the following code to the **`docker-compose.yaml`** file:

```
version: '3'
services:
  nginx:
    image: nginx:alpine
    ports:
      - 5043:443
    links:
      - registry:registry
    volumes:
      - ./auth:/etc/nginx/conf.d
      - ./auth/nginx.conf:/etc/nginx/nginx.conf:ro
  registry:
    image: registry:2
    environment:
      - REGISTRY_HTTP_ADDR=0.0.0.0:5055
    volumes:
      - ./data:/var/lib/registry
```

Add the following code to the **`nginx.conf`** file; comments are provided inline:

```
events {
    worker_connections  1024;
}

http {
  upstream docker-registry {
    ## Avoid conflicts on Mac
    server registry:5055;
  }
```

https://t.me/learningnets

```
10.
11.   ## Set a variable to help us decide if we need to add
    the ‹Docker-Distribution-Api-Version› header.
12.   map $upstream_http_docker_distribution_api_version $docker_distri-
    bution_api_version {
13.     '' 'registry/2.0';
14.   }
15.
16.   server {
17.     listen 443 ssl;
18.     # Custom name
19.     server_name nginx_registry;
20.     # SSL provided by the certificate in Chapter 3
21.     ssl_certificate /etc/nginx/conf.d/domain.crt;
22.     ssl_certificate_key /etc/nginx/conf.d/domain.key;
23.     ssl_protocols TLSv1.2;
24.     ssl_ciphers 'EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH';
25.     ssl_prefer_server_ciphers on;
26.     ssl_session_cache shared:SSL:10m;
27.     client_max_body_size 0; # disable any lim-
    its to avoid HTTP 413 for large image uploads
28.     chunked_transfer_encoding on;
29.     location /v2/ {
30.       # Do not allow connections from Docker 1.5 and earlier
31.       # Docker pre-1.6.0 did not properly set the user agent on ping
32.       if ($http_user_agent ~ "^(docker\/1\.(3|4|5(?!\.
    [0-9]-dev))|Go ).*$" ) {
33.         return 404;
34.       }
35.       auth_basic "Registry realm";
36.       auth_basic_user_file /etc/nginx/conf.d/nginx.htpasswd;
37.       proxy_pass                          http://docker-registry;
```

https://t.me/learningnets

```
38.       proxy_set_header  Host              $http_
   host; # required docker client
39.       proxy_set_header  X-Real-IP         $remote_
   addr; # pass on real client IP
40.       proxy_set_header  X-Forwarded-For   $proxy_add_x_forwarded_
   for;
41.       proxy_set_header  X-Forwarded-Proto $scheme;
42.       proxy_read_timeout                  900;
43.     }
44.   }
45. }
```

Execute the following to run the container stack:

```
1. $ docker-compose up -d
2. [+] Running 3/3
3.  ⠿ Network chapter_04_default       Created        0.0s
4.  ⠿ Container chapter_04-registry-1  Started.       0.5s
5.  ⠿ Container chapter_04-nginx-1     Started
```

The proxy will authenticate the user on the Docker login command:

```
1. $ docker login localhost:5043
2. Username: luigi
3. Password:
4. Login Succeeded
```

As per the example in the previous section, it is now possible to push and pull images to the registry via nginx proxy:

```
1. $ docker tag alpine localhost:5043/my-local-alpine-image
2. $ docker push localhost:5043/my-local-alpine-image
3. $ docker pull localhost:5043/my-local-alpine-image
```

When a reverse proxy configuration is not needed, an excellent and fast security authentication mechanism can be achieved adding a TLS certificate to the Docker compose file. First step is to copy the certificates into the certs folder:

https://t.me/learningnets

```
$ cp client-cert.pem certs/domain.crt
$ cp client-key.pem certs/domain.key
```

Then, edit the Docker compose file:

```
version: '3'
services:
  registry:
    restart: always
    image: registry:2
    ports:
      - 443:443
    environment:
      REGISTRY_HTTP_ADDR: 0.0.0.0:443
      REGISTRY_HTTP_TLS_CERTIFICATE: /certs/domain.crt
      REGISTRY_HTTP_TLS_KEY: /certs/domain.key
    volumes:
      - ./certs:/certs
```

The push and pull commands do not need Docker login authentication to allow images to be transferred because the Docker registry is listening over port 443.

Role-Based Access Control

Role-Based Access Control (**RBAC**) is an access control methodology that assigns permissions to users based on the role they hold in the organization. This approach guarantees fine-grained control, and it is less prone to error. It is worth mentioning that Docker has, of course, an RBAC system, which was exclusively part of the Docker Enterprise Edition; therefore, this feature is not available in the Docker Engine (recently renamed as Docker Community Edition).

As discussed in *Chapter 03, Container Stack Security*, the Docker Enterprise Edition was sold to Mirantis in the late 2019, so all the related container images or solutions were removed within a year from the acquisition. Also, it is not in the scope of this book to provide edge information that would eventually benefit only a small niche of readers or a single use case for a specific technology; instead, it is more helpful to provide a valid security alternative that could fit the needs of a wider audience.

https://t.me/learningnets

While container orchestrator systems like **Rancher**, **Kubernetes**, **Apache Mesos**, **RedHat OpenShift**, **Hashicorp Nomad**, and basically all the cloud-based container services provide a user management system with role-based access control, self-hosted solutions today are lacking in this aspect. In this scenario, it is worth mentioning two solutions: the first one is **Portainer.io**, a container management platform that offers role-based access control in a very affordable business tier; and the second one is **Harbor**, where the **RBAC** capabilities are directly tied to the container registry solution.

With Kubernetes taking the spotlight where role-based access control is included in the cluster, it is hard to suggest a solution that could fit well container environments only. Adopting cloud solutions can simplify access management, with the role-based access control system implemented as part of the **Identity and Access Management** (**IAM**) service.

Type	Description
Limited Guest	Only pull images permissions, no push
Guest	Read-Only privileges for a specific project
Developer	Read and Write privileges for a project
Maintainer	Elevated permissions like scan images, delete images or helm charts
Project Admin	User management privileges, such as adding or removing members and execute vulnerability scans

***Table 4.7**: Harbor RBACs*

Harbor is limited to container images management. To provide access to images, users must be added to the project that handles those images; then the user can have any of the roles listed in *Table 4.7, Harbor RBACs.*

Note: Harbor has two system-level roles: anonymous for not logged users with read-only access to public projects, and system administrator, with super privileges like list all the projects, elevate user to admin roles, delete users, and set vulnerability scan policies.

Lastly, DockerHub provides a mechanism called **Registry Access Management** (**RAM)**, an option available to organizations with a Docker Business subscription. RAM works toward centralizing access to registries from the Docker Desktop with a whitelisting principle.

Auditability

Docker provides audit logs features at the organizational level for repositories that are part of a Docker Team subscription. Audit is a very important step in securing

https://t.me/learningnets

the container environment. An overview on which features we should consider on auditing Docker containers is provided in this section; a more modern solution will be explored in *Chapter 06, Monitoring Containers and Security.*

The following information is considered as quick runbooks or cheat-sheets; it is helpful in troubleshooting scenarios, or when a quick copy and paste exercise can easily provide useful details as part of a single operation or wider investigation. *Table 4.8, Docker images and containers*, lists the most common auditing commands for images and containers:

Type	Description
docker images --digests image-name	Checking the checksum of the image
docker trust inspect image-name --pretty	Content trust for signatures
docker inspect image-name	Metadata, env var, and secrets on the image
docker inspect container-name	Metadata, env var, and secrets on the container
docker history image-name	History of the image

***Table 4.8**: Docker images and containers*

Table 4.9, Docker volume and networks, describes the most common auditing commands for volumes and networks:

Type	Description
docker volume ls	List docker volumes
docker volume inspect volumeId	Inspect docker volume
docker network ls	List docker networks
docker network inspect networkId	Inspect docker network

***Table 4.9**: Docker volume and networks*

Table 4.10, Docker registry, describes the most common auditing commands for registries:

Type	Description
curl -s http://localhost:5000/v2/_catalog | jq .	Verify that the Docker registry is running
docker run --rm -it localhost:5000/my-image:latest sh	Inspect container

***Table 4.10**: Docker registry*

https://t.me/learningnets

Table 4.11, Docker runtime describes the most common auditing commands for runtime endpoints:

Type	Description
docker system info	Docker daemon configuration check
docker system events	Runtime generated events
sudo cat /lib/systemd/system/docker.service	API exposure check
docker inspect | grep -i '/var/run/'	Verify that Docker socket is mounted

***Table 4.11**: Docker runtime*

Image control

The scope of this section is to apply the logic of **idempotent** instructions to container images. Idempotency refers to the capability of container images to be independent from the system in which they are built, executed, destroyed or updated. In other words, regardless of the system in which the instructions reside, the image output will be always the same.

When a container is created from an image, it cannot be changed, because the underlying image from which the container is derived cannot be changed either. There is not really an update methodology about container, similar to what a software update like **yum update** or **apt update** would be.

In this mini-lab working with image tags, a full image updating logic will be demonstrated, brining full control on the container image environment. Assuming that there are several containers running on the older **nginx:1.22.0** image:

```
1. $ docker images
2. REPOSITORY         TAG       IMAGE ID       CREATED         SIZE
3. aquasec/trivy      latest    9a3534dae91d   11 days ago     192MB
4. nginx              1.22.0    55eae9c5b582   2 weeks ago     134MB
5. nginx              1.23.1    0c404972e130   2 weeks ago     135MB
6. jenkins/jenkins    latest    c32212374bfe   3 weeks ago     453MB
```

Its life cycle is due, so it is time to replace the image on which the containers have been created with the more recent **nginx:1.23.1**. The following Docker command with the "**ancestor**" filter will display the running containers on **nginx:1.22.0** only:

https://t.me/learningnets

```
$ docker ps -a --filter "ancestor=nginx:1.22.0"
CONTAINER ID   IMAGE          STA-
TUS          PORTS                  NAMES
1b24fcd4cf74   nginx:1.22.0   Up 11 minutes   0.0.0.0:8082->80/
tcp   old-nginx3
45c173b8fb83   nginx:1.22.0   Up 12 minutes   0.0.0.0:8081->80/
tcp   old-nginx2
83b0fcc32c06   nginx:1.22.0   Up 12 minutes   0.0.0.0:8080->80/
tcp   old-nginx1
```

It is more efficient to use the previous command as an input for the **docker stop** command:

```
$ docker stop $(docker ps -aq  --filter "ancestor=nginx:1.22.0")
1b24fcd4cf74
45c173b8fb83
83b0fcc32c06
```

Now it is possible to delete the old containers:

```
$ docker rm $(docker ps -aq --filter "ancestor=nginx:1.22.0")
1b24fcd4cf74
45c173b8fb83
83b0fcc32c06
```

And delete the ancestor image:

```
$ docker rmi nginx:1.22.0
Untagged: nginx:1.22.0
Untagged: nginx@sha256:4535aaa94ae5316180fac74c56035921280275d0e-
c54282253e1a95536d62a05
Deleted: sha256:55eae9c5b5821494851315634494cbd272ba050d9d-
9912a6375b142d79f37cdc
Deleted: sha256:b852123f741b38df06e063858df4ee68c4fe9edfa6be71b-
9ca652cfc977cafa7
Deleted: sha256:47e3a69d9ba6cdf7cc8f5bd789195fe4ff24c-
1c28260c2950561c6b4d051dfba
Deleted: sha256:b227ccadd8de1d1130c3561e7c03e5c95b07ddb-
25d4ad7794d783b588fdc506c
```

https://t.me/learningnets

```
8. Deleted: sha256:680b47ce99eea59d8052263d4172dea68aa7ce566cb52a22ec-
   c70e08f64c928e
9. Deleted: sha256:5aef7e48290092f2f2fd68bc8d1fe20b8ac7d82c-
   ba9d174d9f9990a67223a787
```

Finally, creating new containers based on the new image is as follows:

```
1. $ docker run --name new-nginx -d -p 8080:80 nginx:1.23.1
2. 34164dcee11751abc0351493d7d750b7d0bc7ac332ca3030eb321e0285289984
```

Figure 4.10, Image Control, illustrates the image control cycle, but the previous exercise is not efficient in large-scale environments. To solve this problem, there is a better way to apply **Continuous Deployment** to any containerized platform, and it works with private registries and enforcing **Transport Layer Security** (**TLS**) certificate connection: **Watchtower**.

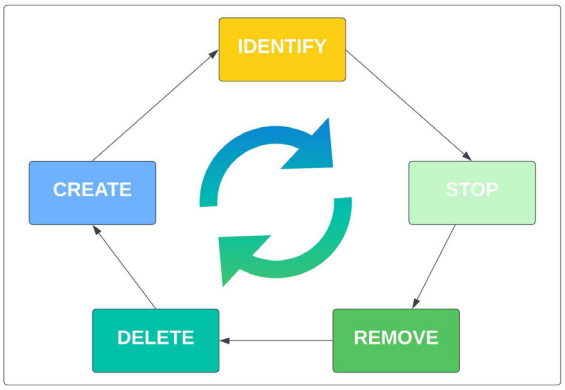

***Figure 4.10**: Image Control*

Once **Watchtower** has been deployed and authenticated against the registry, it will monitor the registry for pushes. When a new image is pushed to the registry, **Watchtower** will pull the new image, shut down the associated containers, and re-deploy those containers with the new image.

Watchtower can read the interdependency between containers like web, API and database, and it can replace them in a logical order to obtain a graceful shutdown and an efficient deployment. The level of customization uses the hooks listed in *Table 4.12, Watchtower hooks*:

https://t.me/learningnets

Type	Description
pre-check	Before it checks whether an update is available
pre-update	After the update is found but before the update is executed
post-update	After the update is completed
post-check	Verify the update has been completed

***Table 4.12**: Watchtower hooks*

It also provides great notification capabilities, such as email notifications, Microsoft Teams and Slack integration.

Vulnerability management

The vulnerability management has been introduced in a few previous sections, such as **Scan and verify images**, where we explained that scanning images is an essential part of the container security life cycle, and the **Private and public registries** as registry software solutions can provide image scanning capabilities. Developing the right procedure to manage **Common Vulnerabilities and Exposures** (**CVEs**) in container images is pivotal toward securing the container platform.

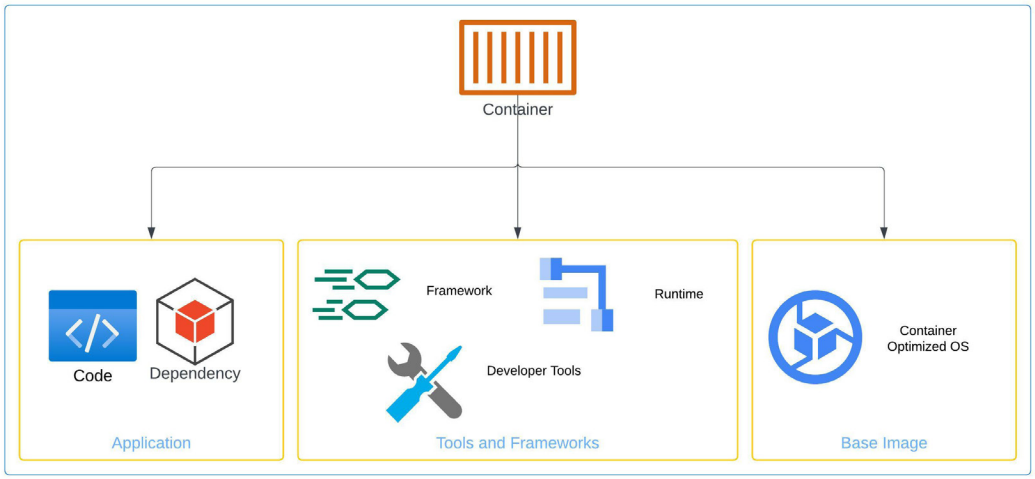

***Figure 4.11**: Container*

The aim of this section is not to provide the umpteenth tool to readers; it would not be possible to treat any solution in depth. Instead, the section aims to look at vulnerability management as a **strategy**. *Figure 4.11, Container*, depicts the complexity each image can achieve, and thus the footprint each vulnerability management would entail.

https://t.me/learningnets

Container lifecycle can be misleading. The maximum age of a container is considered extremely short: hours rather than days.

This is often not true, especially in production customer-facing environments or even worse in highly regulated environments, where any deployment in production goes through a complex and long change control methodology; it is not uncommon to see the same container version running for weeks or months.

Furthermore, scenarios in which the risk of deploying or running flawed systems is accepted by the business due to technology constrains or lack of resources are not common, but they do exist.

There are few strategies that can be used, and the effectiveness of each of these varies upon the deployment environment, but the general rule should be reflected in *Figure 4.12, Deployment Logic*, highlighting the general concept that should be applied to vulnerability management.

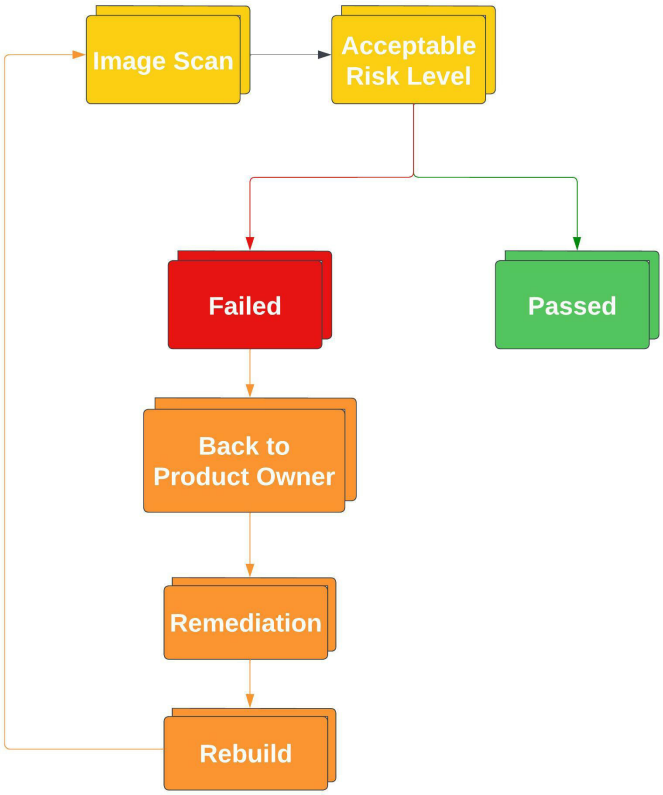

***Figure 4.12**: Deployment Logic*

https://t.me/learningnets

The key element in the logic is the **acceptable risk level** concept recalled into the **Scan and verify images** section earlier in this chapter. When deciding the best vulnerability management strategy, it is important to understand how the **Common Vulnerabilities and Exposures** (**CVE**) scoring works.

The CVE framework is a reference system for known information security vulnerabilities maintained by US National Cybersecurity **Federal Founded Research and Development Center** (**FFRDC**) and operated by The Mitre Corporation. The Mitre maintains the CVE entries at **https://cve.mitre.org**, but they do not include technical information about the risk and, eventually, the fix. Those details are maintained by the **National Institute of Standards and Technology** (**NIST)** in their **National Vulnerability Database** (**NVD)** system browsable at **https://nvd.nist.gov,** which is mirroring the Mitre database.

The NVD implements in their **Common Vulnerability Scoring System CVSS** version 3.0 a metric to classify vulnerabilities, which is based on the combination of factors like attack vector, scope, privileges required, user interaction, impact, and exploitability. The scoring points resulting from the combination of the previous factors are then layered in severity ranges, as shown in *Figure 4.13, NVD Scoring*:

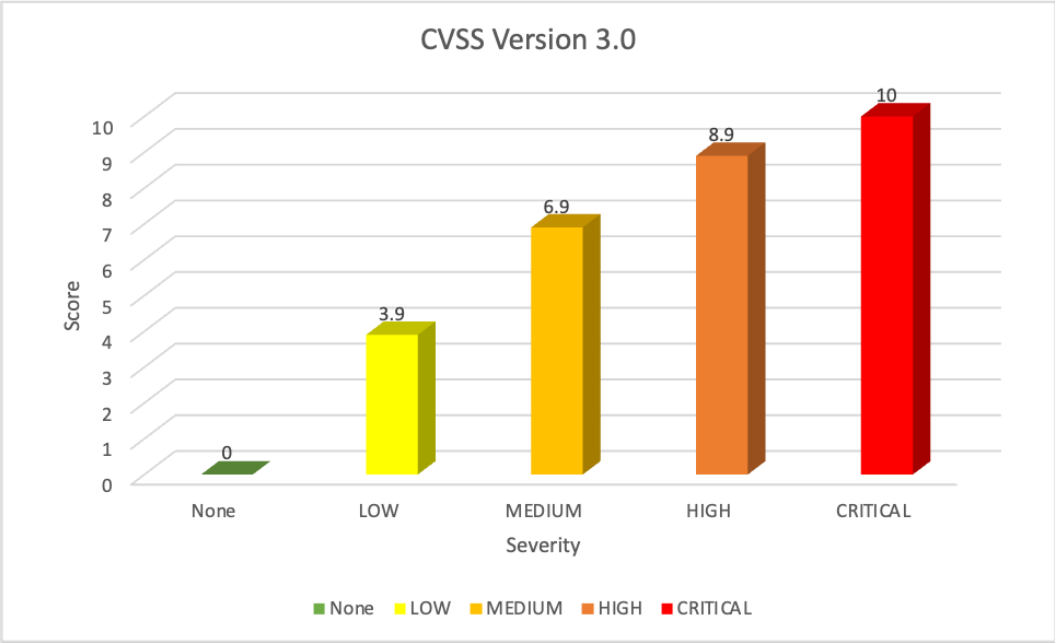

***Figure 4.13**: NVD Scoring*

There are different ways to define an acceptable risk level; the preferred one would be zero CVE, but other considerations could affect this decision, such as legacy system, firewalling mechanism, access control, network exposure, business considerations, market presence, and geo-location.

https://t.me/learningnets

The options in *Figure 4.14, Acceptable Risk Level*, are the most common ones, but in relation to the specific business goals, there could be other combinations to consider, such as remediations for all the scoring levels within a certain time frame and in relation to compliance goals.

This threshold is extremely important and must be evaluated accurately. Let's examine in detail *Figure 4.14, Acceptable Risk Level*, with the most common options available:

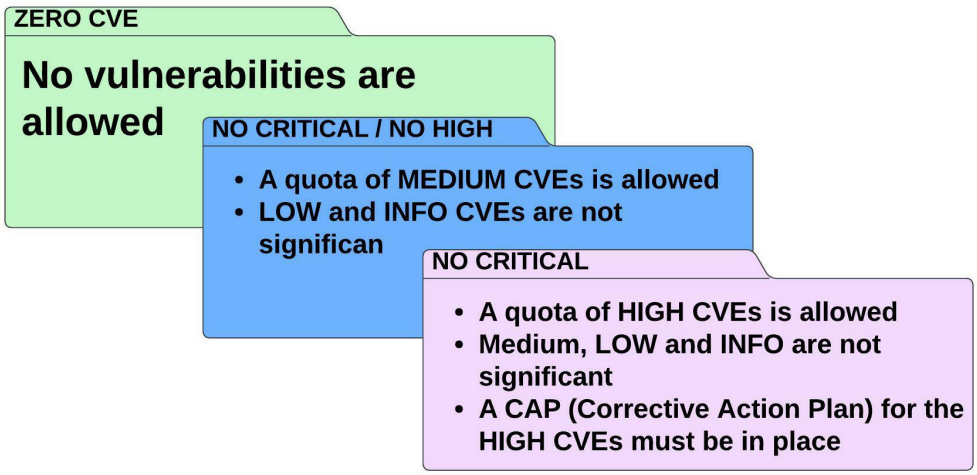

***Figure 4.14**: Acceptable Risk Level*

The previous strategies are not mutually exclusive; they can be combined in different ways to achieve zero trust. Remember that the zero trust model means that devices, systems or components are never trusted by default, even if previously verified.

In relation to the Acceptable Risk Level's threshold, the vulnerability management strategies adopted are summarized in *Table 4.13, Container Vulnerability Management Strategies*, which details the most common strategies for applying security best practices to container platforms:

Description	Description
After build	Images are scanned after build and before push.
Registry scan	Images are scanned after push and before pull.
Staging Strategy	Images are scanned in a staging registry.
Runtime scan	Container are scanned during execution.
Combined Strategy	Managed strategy combination.

***Table 4.13**: Container Vulnerability Management Strategies*

https://t.me/learningnets

The **After Build** scan approach is arguably the simplest and immediate. It offers immediate feedback after the image is built and helps keep the registry tidy. Of course, the build can happen on a local machine with tools like Anchore, Trivy or Clair, or integrated in **Continuous Integration** (**CI**) pipelines and using cloud systems like Azure DevOps or AWS CodePipeline, where the same tools can be leveraged in the earliest of the shift-to-the-left approach possible.

Figure 4.15, After Build, illustrates the location of the scanner immediately after the image has been generated by the **CI** pipeline. If the scanned image meets the requirements of the **Acceptable Risk Level**, it gets pushed to the registry; if it does not, the build fails and an alert is generated for the developer to review the code. There is no deployment at this stage. Refer to the following figure:

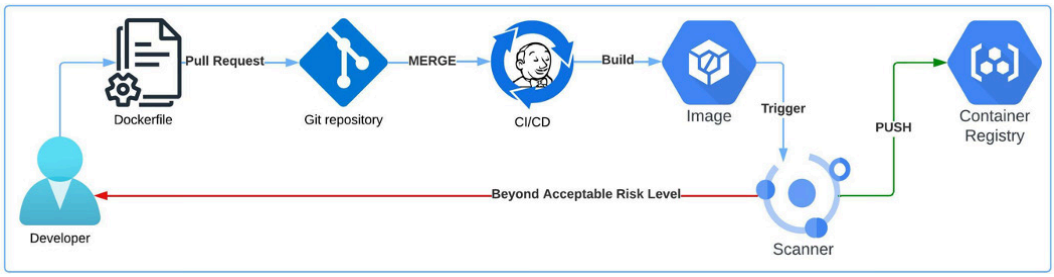

***Figure 4.15**: After Build*

In the **Registry Scan** approach, the container image scanner is placed inside the **container registry** system, so the scan happens after the **image** has been built and pushed to the registry. The container registry has no knowledge of what type of software that image contains or if the image has any vulnerability; it accepts the push and then starts the scan, as illustrated in *Figure 4.16, Registry Scan*:

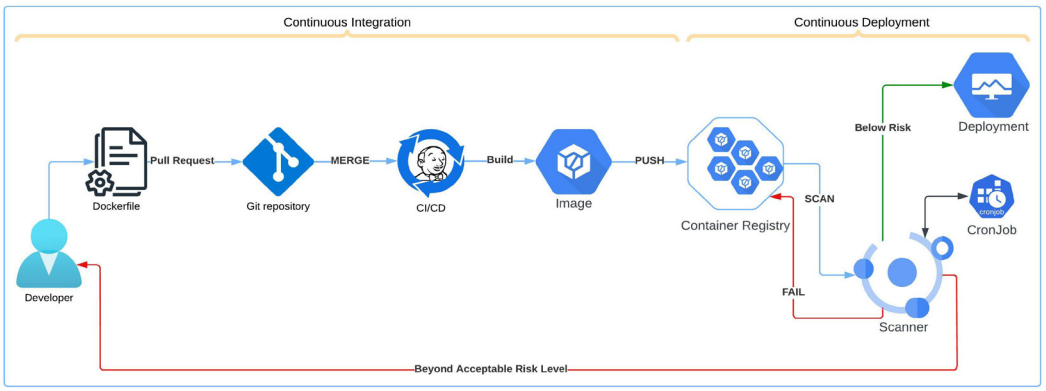

***Figure 4.16**: Registry Scan*

If the container image has a lower level of vulnerability, as defined in the Acceptable Risk Level threshold, the image is deployed, and it becomes a container. If the threshold is exceeded, the scanner marks the image as failed and an alert is triggered

https://t.me/learningnets

for the developer to review the code, then the image is not deployed. The **Staging Registry** strategy involves having two container registries, the staging one is dedicated to analyzing the image at an earlier stage in development cycle, and it ensures that no vulnerable images reach the **production registry**, as shown in *Figure 4.17, Staging Registry*:

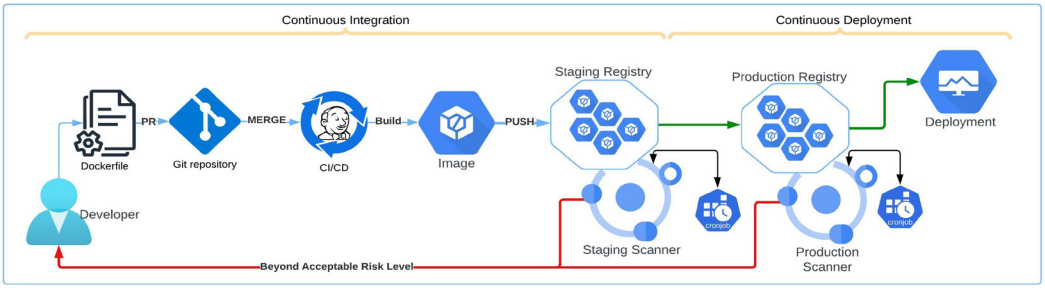

***Figure 4.17**: Staging Registry*

The advantage of this approach is the reduction of the clutter in production, but it implies overhead in managing the two separate systems and pipelines. It does make sense in certain highly regulated environments, such as biotechnology or science-related areas where compliance imposes strictly security control measures.

The **Runtime Scan** strategy goes beyond all the illustrated container registry scanning approaches, so it is scanning the container itself rather than the container images from which the container is created. Take a look at *Figure 4.18*:

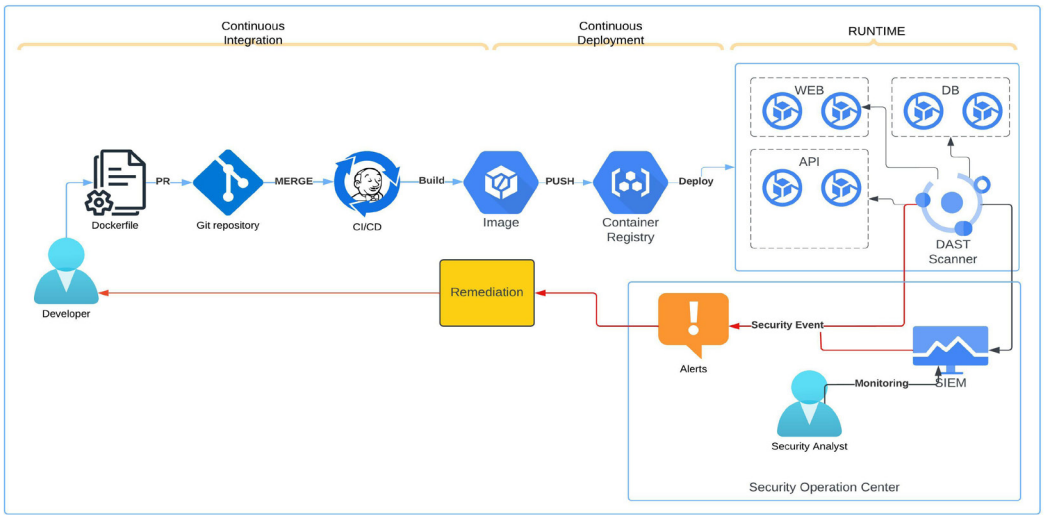

***Figure 4.18**: Staging Registry*

It is essentially an application of the **Dynamic Application Security Testing** (**DAST**) methodology that we will discuss in the next chapter, because it aims to scan the running application, and therefore, is outside the domain of the CI/CD pipeline.

https://t.me/learningnets

DAST uses a behavioral approach and is quite different from the methodologies we have seen so far. It looks at the running application from the "outside", trying to exploit its weakness, and it is limited to test only running executables.

Finally, a combination of all the previously mentioned approaches can provide greater security posture, applying vulnerability management alongside all the steps of the application life cycle, as illustrated in *Figure 4.19, Combined Strategy*:

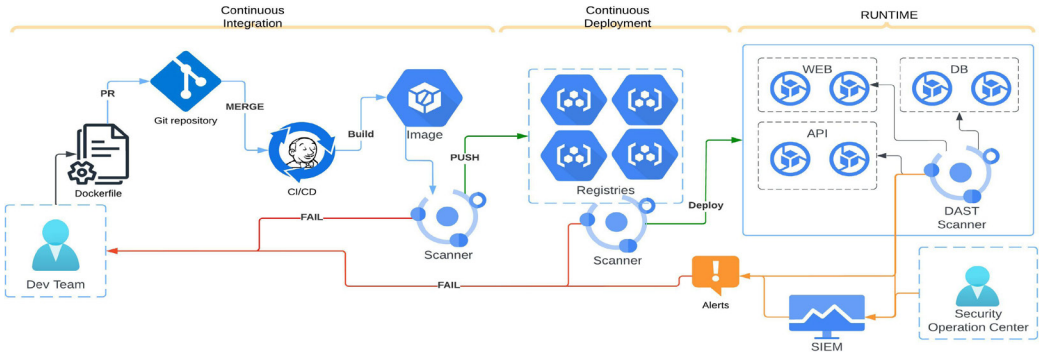

***Figure 4.19**: Combined Strategy*

Lastly, in Kubernetes, we can use the Kubernetes Admission Controller, to leverage external scanners and define policies that would admit a container to be deployed after meeting certain criteria. This topic will be discussed in *Chapter 8, Kubernetes Orchestration Security*.

Conclusion

In this chapter, we discussed the various aspects of container image security and build file configuration security, and we looked at how to scan and verify images. Then, we analyzed how to secure a container registry and explored the communication between the registry itself and external tools.

We also discussed how to control, audit and scan images, with arguments about the various vulnerability management strategies.

In the next chapter, we will learn about container application security.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

https://t.me/learningnets

Chapter 5
Application Container Security

Introduction

Containers, also known as containerized applications, are the essence of a microservices model, where each service is virtualized and exposed on a single running container instance. Readers with many years of experience in the field, and system administrators with memories on when the Linux, Apache, MySQL and PHP (**LAMP**) stack approach was not even virtualized (see *Figure 5.1, LAMP*) and all

https://t.me/learningnets

the services in the stack were running on the same physical hardware, the jump to containerized applications is significantly innovative.

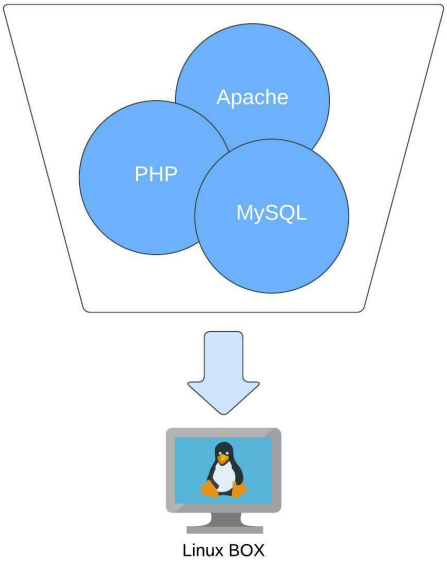

***Figure 5.1**: LAMP*

The microservices architecture, one of the **service-oriented architecture** (**SOA**) applications of software engineering, is an architectural pattern that defines an application as a collection of independent services, communicating between them through a network protocol as opposed to a monolithic application where there are no moving parts. And if a single part of the application is failing, the application functionality is compromised. It is worth recalling that *Figure 1.1, Virtual Machines and Containers Structure*, in *Chapter 1*, Containers and Kubernetes Risk Analysis, with a few adjustments in relation to the concept illustrated in this chapter, becomes *Figure 5.2, Microservices Architecture*:

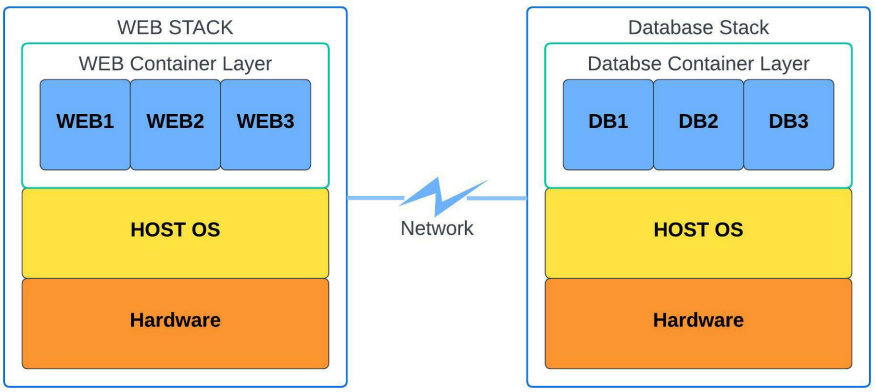

***Figure 5.2**: Microservices Architecture*

https://t.me/learningnets

The advantages of this model are immediate: separation of duties, parallel developing, independent deployment, and monitoring accuracy, but containerized applications expose some security challenges that we partially discussed in the previous chapter.

Containerized applications run isolated systems, each with a specific, and notably, sole purpose. Those are considered as the ensemble of the application itself, including all its dependencies, system libraries, executables and configuration files, as shown in *Figure 4.11, Container*, in the previous chapter.

Structure

In this chapter, we will discuss the following topics:

- Application Container Security
- Threat Intelligence
- CI/CD Security Integration
- Shift Left
- Remediation
- Manage Privileges
- Penetration Testing
- Third-Party Components

Objectives

This chapter aims to provide a set of security best practices for containerized applications, discussing security methodologies for containers while also introducing concepts around threat intelligence, shift-to-the-left, and penetration testing. We will also discuss how these methodologies and their implementation in complex continuous integration and continuous deployment pipelines, affect positively a full containerized secure software development life cycle.

Application Container Security

Before getting deep into analyzing the various aspects of a comprehensive application container security workflow and their relationship with any single step of the **Software Development Life Cycle** (**SDLC**), there is a framework that has recently become popular in the community that provides a quick and efficient breakdown set of security controls and requirements, the **OWASP CSVS**, which stands for Container Security Verification Standard.

https://t.me/learningnets

Readers familiar with such methodologies for container security would also be familiar with frameworks like **National Institute of Standard and Technology** (**NIST**) Application Container Security Guide publication NIST.SP.800.190 or **Center for Internet Security** (**CIS**) Docker Benchmark.

The **NIST Application Container Security Guide** is a concept guide that focuses on abstraction of virtualization and containerization. It highlights, with a risk-based approach, the security best practices, the remediations or countermeasures needed to reach a sufficient level of container security posture, but it does not offer any specific technical suggestion.

The **CIS Docker Benchmark** is a prescriptive guide providing security controls and requirements to elevate the security posture of the container platform, a very technical document that is considered one of the industry references today but could result in a very challenging implementation due to its over two hundred security recommendations.

The **OWASP Container Security Verification Standard** aims to provide a middle ground approach with a clear security standard framework for container platforms, with an easy-to-go approach to verify the security posture related to container solutions. This framework proposes a new approach with three layers: a **Level 1 Basic Security** that can be applied to all the container platforms, a **Level 2 Advanced Security** meant for sensitive data container system in need of additional security protection, and a **Level 3 High Security** container project handling medical data like **Personal Health Information** (**PHI**) or high value transactions. Readers interested in enhancing their knowledge about the solutions mentioned earlier can refer to the following addresses:

- OWASP CSVS **https://owasp.org/www-project-container-security-verification-standard/migrated_content**
- CIS Docker Benchmark **https://www.cisecurity.org/benchmark/docker**
- NIST SP 800-190 **https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-190.pdf**

In the previous chapter, we acknowledged, with reference to *Figure 4.11, Container*, the three main components of a container application, which are as follows:

- The application code itself and its dependencies
- The framework or runtime necessary to successfully execute the application
- The operating system on which the two previous points are ingested

As we discovered in the previous chapter, there are different ways to enhance security posture in a container platform, and each of the three points identified earlier has a specific methodology that is summarized in *Figure 5.3, Microservices Model*:

https://t.me/learningnets

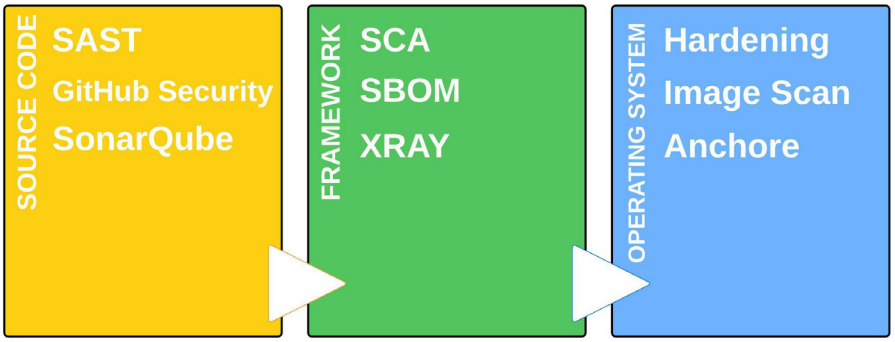

***Figure 5.3**: Microservices Model*

There are two ways to protect a running container, both of which were introduced in the previous chapter: an **external method**, using Nginx Reverse Proxy, and an **internal method**, using a container runtime testing system. In the previous chapter, we looked at Nginx as a security authentication mechanism, specifically in conjunction with a container registry, but Nginx can be used also, and mainly, as a reverse proxy system to work as the first line of defense against running containers. The benefits that Nginx Reverse Proxy add in value as security posture to the container platform are as follows:

- Façade Routing and HTTPS
- TLS Offloading
- Authentication and Authorization Offloading

Table 5.1, Reverse Proxy lists the main features:

Type	Description
URLs restrictions	URLs Access Restrictions
Intercept Response Headers	Upstream servers' headers interception
Control Request Methods	Control the request methods
Control Domain Level Access	Define what [*].domain can be accessed and by who
Façade Routing	URLs Layer exposed
URLs rewrite	Fix broken URIs
API Version Control	Versioning Control

***Table 5.1**: Reverse Proxy*

https://t.me/learningnets

Let us visually clarify the intended architecture of this solution in *Figure 5.4, NGINX Reverse Proxy:*

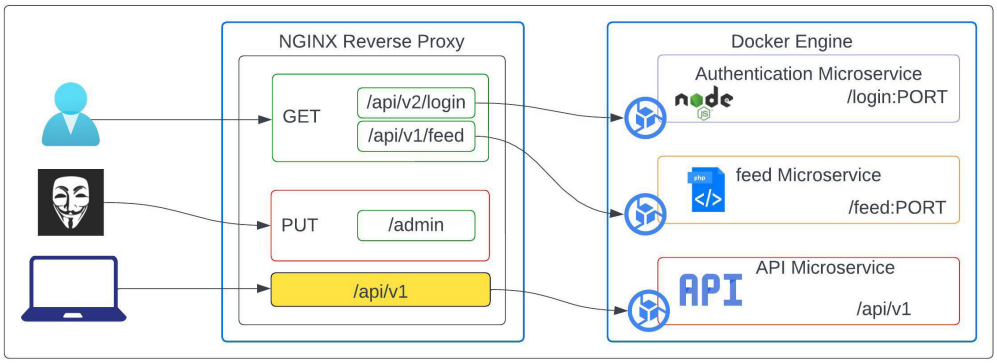

***Figure 5.4**: NGINX Reverse Proxy*

In the preceding example, the network traffic is coming from the left, hitting the **Nginx Reverse Proxy** only. There are two different domain names: **api.mydomain.com** running on port **80** with three different **URIs** (**Uniform Resource Identifier**), and **api2.mydomain.com** running on the same port but with its own unique URI. Those two domains are configured on the NIGNX reverse proxy system to talk uniquely with the container services running in the container runtime.

The **Nginx Reverse Proxy** is blocking all the malicious activities, such as the **PUT** attempt on the **/admin** URI, but is allowing all the legit network connections. A sample of the Dockerfile snippet for the login service is in the following code:

```
1. login:
2.     image: node:slim
3.     container_name: login
4.     depends_on:
5.       - nginx-proxy
6.       - letsencrypt
7.     environment:
8.       - VIRTUAL_HOST=api.mydomain.com
9.     networks:
10.      - my-reverse_proxy-net
```

Note: The preceding code is only an example.

https://t.me/learningnets

You would have noted the **letsencrypt** directive in line six. **Let us Encrypt** is a non-profit **Certificate Authority** (**CA**) maintained by the **Internet Security Research Group** (**ISRG**) that issues X.509 certificates for **Transport Layer Security** (**TLS**) encryption for free, with the goal of securing websites using HTTPS. The non-profit organization has many major sponsors, including but not limited to Facebook, Google Chrome, Mozilla Foundation, Cisco Systems and AWS. *Table 5.2, TLS Offloading*, illustrates the complexity of managing TLS:

Type	Description
TLS Protocols	v1.1, v1.2, v1.3
Cyphers	RSA, PSK, ECDH, and so on
Sessions	How to handle TLS sessions
OCSP	Online Certificate Status Protocol (revoking process)
Key Management	How to manage key pairs
Vulnerabilities	How to manage vulnerabilities
Patching	How to implement a patching cycle
Performance degradation	Overhead created by the TLS system

***Table 5.2**: TLS Offloading*

The implementation of the **Let's Encrypt** solution would help offload all the managing key points expressed in the previous table by automating the TLS delivery mechanism through container deployment. Let us look at *Figure 5.4* again, with the addition of the letsencrypt directive in *Figure 5.5, NGINX Let us Encrypt.*

Deploying the Nginx Reverse Proxy through a container implementing the **letsencrypt** directive allows us to pull a certificate from the **Let's Encrypt** Certificate Authority. The reverse proxy uses a system called **certbot** to request certificates from

https://t.me/learningnets

Let's Encrypt, then it stores those certificates and keys in memory or on disk, or loaded on demand with a mechanism called **lazy load**.

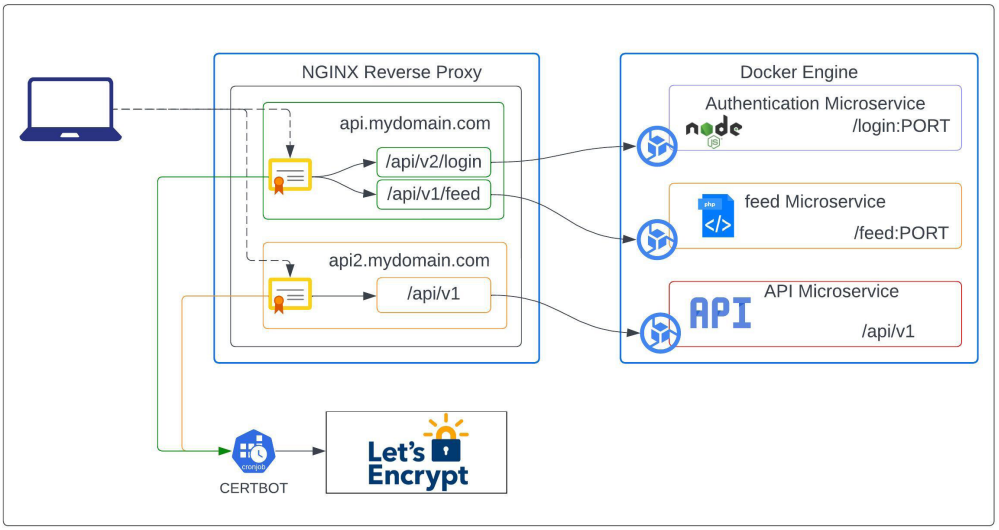

***Figure 5.5**: NGINX Let us Encrypt*

Nginx Plus, an enterprise version of the famous Nginx system, also offers a **Key Value Database** that can be implemented in this scenario to handle certificates and keys, adding one more layer of security.

It is suggested to deepen the Nginx knowledge, as this is also one of the most utilized Kubernetes Ingress Controller. You can find more information at **https://www.nginx.com/products/nginx/**. Besides its native connection with container registries, Nginx can be used for more complex authentication and authorization scenarios, such as the ones in *Table 5.3, Authentication and Authorization*:

Type	Description
IDP Support	Identity Provider integration (OIDC)
MFA	Multi Factor Authentication support
Authentication	Credential Validation offloading
Policy Enforcement	Authorization enforcement
Interception	Block unauthenticated requests

***Table 5.3**: Authentication and Authorization*

Nginx can load a **JSON WEB Key** file, essentially a secret file in JSON format returned upon client request, into a specific domain to serve as a decryption key to

https://t.me/learningnets

decode the header string and payload into a JSON structure and build variables for the contents of all the keys and values, as illustrated in *Figure 5.6*, *JSON WEB KEY*.

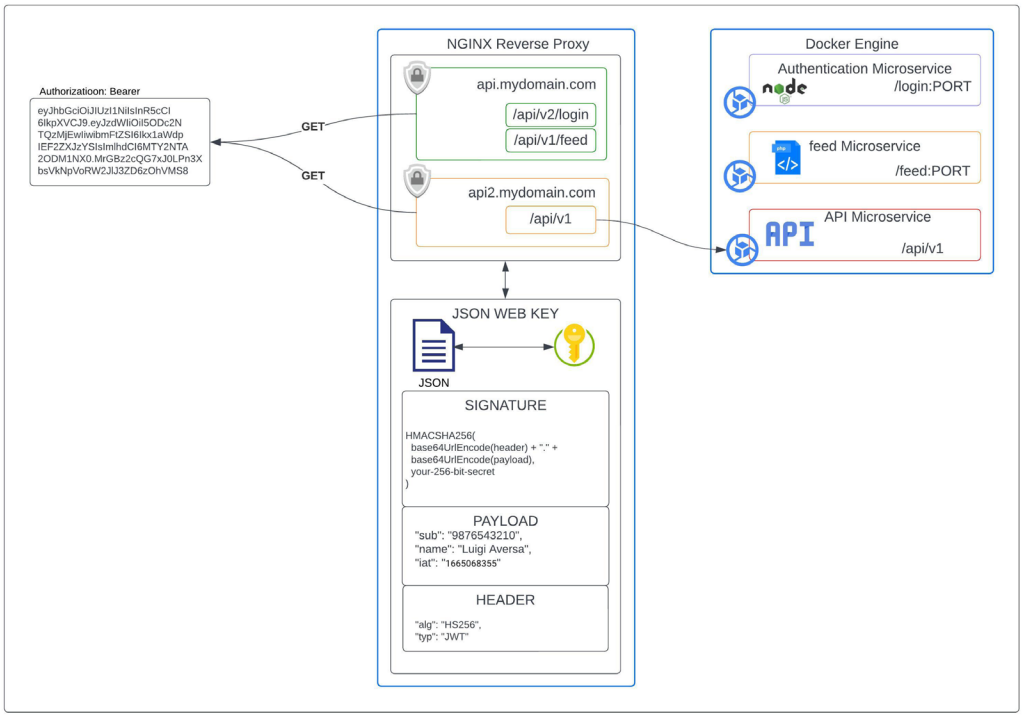

***Figure 5.6**: JSON WEB Key*

In all the three cases illustrated so far, Nginx Proxy Server can be deployed as a container to route the traffic to another container deployment via network communication, as demonstrated in *Chapter 4, Securing Container Images and Registries*, in the *Private and Public Registries* section. Eventually, it can be deployed with the sidecar approach where essentially, a new Nginx Proxy Server is deployed for each container.

Note: A couple of useful resources: for JWT encoding, visiting https://jwt.io is helpful, while for the "iat" timestamp (as part of the payload in the previous figure), which stands for "issued at", this tool is excellent: https://www.timestamp-converter.com.

Among the internal testing methodologies, the **Dynamic Application Security Testing** (DAST) system is likely the most widely known, but there are two more testing approaches less know to the public such as **Interactive Application Security Testing** (IAST) and **Runtime Application Self-Protection** (RASP). *Figure 5.7*,

https://t.me/learningnets

DAST vs IAST vs RASP, illustrates the main difference between the three testing methodologies:

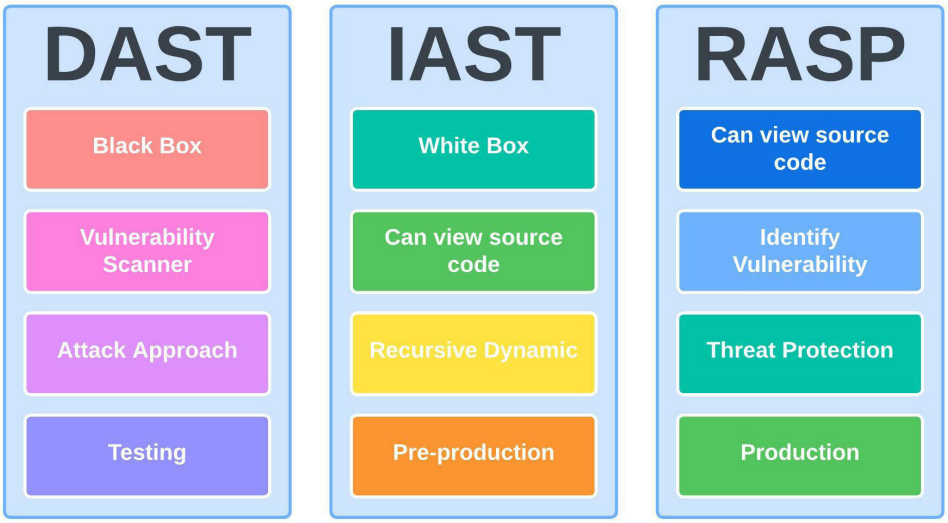

***Figure 5.7**: DAST vs IAST vs RASP*

A **DAST** system comes to help when the container is created and the application is up and running. It is a kind of "black box testing" because it doesn't have access to the application code and therefore, has no knowledge of software internals. The main fields of applications are as follows:

- Detect exploitable vulnerabilities in a running application.
- Detect issues in HTTPS traffic, API system, sessions, JavaScript, authentication, and so on.
- Simulate XSS (Cross Site Scripting), Code Injection, CSRF (Cross Site Request Forgery) attacks, and so on.

Unfortunately, due to its very nature, a DAST system scan could result in unexpected side effects like crashing the application or generating false positive alerts. Because it scan the application from outside, it can detect the vulnerability, but it cannot tell the portion of the code that generates that vulnerability. The best case scenario for DAST applications is likely to be a testing environment.

There are many commercial solutions that can apply the DAST methodology, such as **Veracode Dynamic Analysis**, **Rapid7 InsightAppSec**, and **Detectify Deep Scan**, which is a cloud-based DAST tool created by a team of ethical hackers. On the open-source side, **ZAP** (**Zed Attack Proxy**) stands out as it has a passive mode called the **Baseline Scan** that doesn't attack the application. This solution has been adopted by GitLab as part of their DevSecOps CI/CD pipeline.

https://t.me/learningnets

An **IAST** is a modern approach for testing application security; it is a hybrid methodology that combines the best **SAST** and **DAST** tools. An IAST tool can scan the application during the development phase in a white-box approach, and during the application execution runtime using a black-box approach, by means of intercepting actual user inputs and actions. Refer to the following figure:

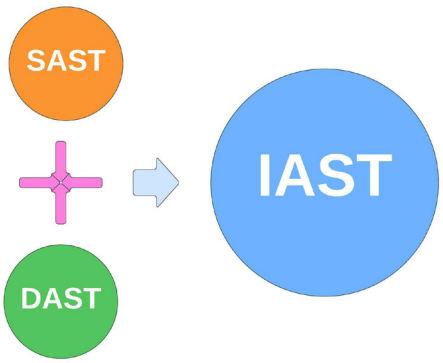

***Figure 5.8**: IAST*

Figure 5.8, IAST, shows the IAST approach as a combination of SAST and DAST, combining the best of the two worlds in one methodology. A code analyzer tool (SAST) can look for security issues in the source code, but it can't detect an SQL injection in that code, and vice versa: a DAST tool can look for **Cross Site Scripting** (**XSS**), but it can't verify which part of the code is generating that vulnerability. This is when the benefit of the IAST approach comes into scope: in its ability to analyze and explore the surface of the application, and also in pinpointing the vulnerability to its source code.

This brings many advantages, such as identifying vulnerabilities early in the deployment phase (shift-to-the-left approach), allowing for faster fix and remediation processes, reducing false positives and also creating a consistent, homogeneous development life cycle, enabling better control on both the coding experience and the runtime environment.

There are two types of IAST, as illustrated in Figure *5.9, IAST types*: active and passive. An active IAST combines application security testing with vulnerability scanning via an agent installed on the host machine (it could be the container from

https://t.me/learningnets

where the application is launched), while a passive IAST has no application security testing methodology.

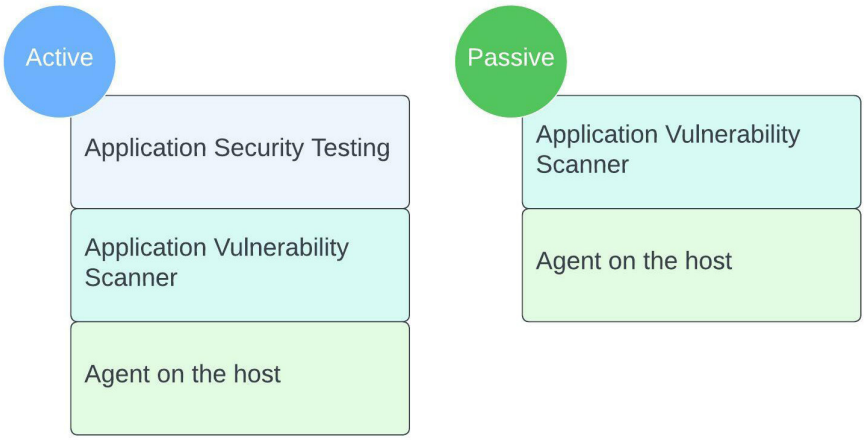

***Figure 5.9**: IAST types*

There are two open-source projects and one commercial solution in this field worth highlighting: Hvid Security, Vega and W3AF:

- **Hvid Security** was one of the first security tools enabling DevSecOps best practices combining source code inspection and application security testing. It was initially published as an open-source project and then acquired by **Datadog** at the beginning of 2022 as part of the Datadog Cloud Security Platform.
- **Vega** is a web application security testing platform written in Java developed by Subgraph. It runs in two modes: as automated scanner and as intercepting proxy. When running in the **automated scanner** mode, it crawls the application, collecting information like the Google spider but with the difference that Vega will try to run the appropriate software module to inject code into any vulnerability it discovers. A module is essentially an attack vector (XSS, SQL Injection, and so on). When running as intercepting proxy, Vega is analyzing the browser interaction with the application, intercepting requests and responses, and it can override those requests and induce anomaly behavior. Interestingly, Vega can also simulate HTTPS communications generating dynamic certificates to simulate real-world attack conditions. Visit the following web address for more info **https://subgraph.com/vega**.
- **W3AF** is an application security tool focused on attack and auditing. Its approach is very similar to what Vega can do, with two major differences: it has an extensive vulnerability type library, over 200 (refer to *Table 5.4, W3AF Vulns Type*, for an overview of the most significant ones), and it has

https://t.me/learningnets

been developed in Python providing, greater flexibility and easy integration. Refer to the following table:

Type	Description
Audit	Blind SQL injection, Buffer overflow, LDAP injection, MX injection, Insecure SSL version, eval() input injection, Shell shock, Rosetta Flash, and so on.
WebSocket	Insecure WebSocket Origin filter, Open WebSocket, Origin restricted WebSocket, Websockets CSRF.
Crawl	dwsync.xml file found, phpinfo() file found, PHP register_globals, Google hack database match, Cross-domain allow ACL, Potential web backdoor, robots.txt file, Identified WordPress user, and so on.
Grep	US Social Security Number disclosure, Parameter has SQL sentence, NTLM authentication, Cookie without HttpOnly, Secure flag missing in HTTPS cookie, Click-Jacking, Private IP disclosure, Oracle application server, .NET ViewState encryption is disabled, Insecure password form access over HTTP.
Infrastructure	HTTP traceroute, Apache Server version, Virtual host identified, Internal hostname in HTML link, PHP Egg, DAV methods enabled, Reverse proxy identified, HTTP load balancer detected, MS15-034, JetLeak.
Brute force	Guessable credentials.
Attack	DAV Misconfiguration, Arbitrary file upload, OS Commanding code execution, Code execution via remote file inclusion, Arbitrary file read, Eval() code execution.

***Table 5.4**: W3AF Vulns Type*

RASP is considered one of the most advanced solutions. It analyzes the application traffic and behavior at the runtime to detect and prevent cyber threats. It can review the application source code even if the executable is compiled, it analyzes vulnerabilities and weakness, and proactively terminate user sessions and issue alerts if a drift from an expected behavior is detected. Even if RASP seems to be the perfect solution, it is not meant for replacing SAST, DAST or IAST. They all act at different levels of the development life cycle, with some overlapping features; take a look at *Figure 5.7, DAST vs IAST vs RASP*.

Due to its very nature, RASP is the perfect tool for legacy applications or for applications with a reduced update life cycle. Although a RASP system seems pretty like a Web Application Firewall (WAF) field application, they are different in essence: a WAF system analyzes the network traffic before the traffic hits the application

https://t.me/learningnets

(perimeter inspection), while a RASP system analyzes the network traffic and eventually blocks malicious activities in relation to the application's behavior.

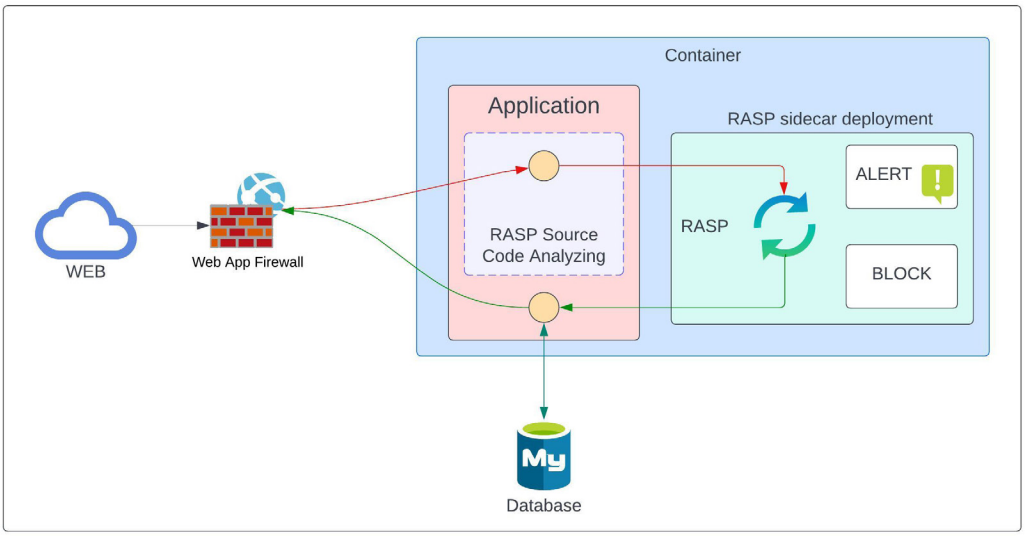

***Figure 5.10**: RASP*

Figure 5.10, RASP, depicts the network communication flow, where it is visible that the WAF is on the network perimeter rather than on the application perimeter. The RASP system analyzes the application's behavior and decides whether to block a specific request, generate an alert or allow the traffic and generate a response based on that.

There are several security tools using the RASP approach, such as Datadog Real-Time Container Monitoring, Contrast Security Protect, Imperva, Dynatrace Application Security, Sysdig Falco, Rapid7 Threat Command, Veracode Analytics and CrowdStrike Falcon.

Threat intelligence

Since RASP has knowledge of the application's runtime environment, it can be tailored to the application's specific requirements. While systems like WAF or **Intrusion Protection System** (**IPS**) are traditionally related to network infrastructure security and are, therefore, used to monitor suspicious activities via network traffic or user sessions, their utilization is limited to the network perimeter.

RASP monitors the application from the inside, moving security inside the container but in an intelligent way because it evaluates real-time application traffic within the context of its expected behavior. It is very effective in cloud environments as field

https://t.me/learningnets

application of cloud security, where traditional security countermeasures are not technologically equipped to deliver security in the cloud.

RASP is application specific, so the sensors installed as part of the code deployment must meet the programming language environment to work properly, but thanks to its deep embedding in the application, it provides a series of advantages:

- A RASP system makes decisions based on the context in which the application resides because it has knowledge of the application architecture.
- It protects from various attack vectors, including OWASP's Top Ten.
- No configuration files, no policies, no rules, and no maintenance is expected, just as simple as installing the RASP sensor.
- **Prevents Cross-site scripting** (**XSS**) and SQL Injection attacks.
- Helps with Zero Day and **Denial of Service / Distributed Denial of Service** (**DoS / DDoS**), attacks.

Let us produce a real-world scenario for an interesting tool developed by Baidu: **OpenRASP**. OpenRASP integrates a protection engine inside the application; the following SQL query has no malicious payload:

```
1. SELECT * FROM mytable WHERE ID = '65324' and begin_time >= '2022-09-
   30' ORDER by end_time
```

However, take a look at the following request body:

```
1. orderBydata=end_time+desc&query=ID+%3d+'65324'+and+begin_
   time+%3C%3D+'2022-09-30'
```

It can change the logic of the SQL statement, so OpenRASP would detect it as a potential malicious activity.

Java is a widely used programming language, and its adaptation as web system is not new. **Java Server Pages** (**JSP**) is a server-side scripting system based on the famous Java technology created by Sun Microsystems in the late nineteens. This solution was created to generate **Hyper Text Markup Language** (**HTML**), **Extensible Markup Language** (**XML**), and other type of pages dynamically, conceptually working like PHP or ASP but using the Java programming language as the back end.

A bi-directional encrypted network traffic over HTTPS would effectively bypass WAF and IDS, even with the following payload, where a **Behinder** remote web shell is requested from the server:

```
1. java.lang.ProcessBuilder.Start
2. ...
```

https://t.me/learningnets

```
3. net.rebyond.behinder.paylod.java.Cmd.RunCMD

4. net.rebyond.behinder.paylod.java.Cmd.equals
```

When the attackers break through the web application firewall and intrusion detection system, they can upload the backdoor., The **Endpoint Detection and Response** (**EDR**) would overlook the threat since the communication is encrypted, while OpenRASP, acting at the application layer would have been able to detect the threat.

Note: Behinder is a very popular implant web server capable of providing very powerful tools to attackers, such as memory-only web shells, and native support with tools like Meterpreter and Cobalt Strike.

That is how Chinese hackers were able to compromise some Confluence servers from the famous Atlassian system in early June 2022, leveraging initially a zero-day vulnerability, impacting the Java-based content management system. This incident is what was coded as CVE-2022-26134; you can find more information at **https://confluence.atlassian.com/doc/confluence-security-advisory-2022-06-02-1130377146.html**.

CI/CD Security integration

Modern software development life cycle processes are managed using **CI/CD** (**continuous integration / continuous delivery**) tools, for the purpose of completely automating the release process. Security testing should be integrated fully inside the continuous integration / continuous delivery pipelines, from planning to testing and deployment.

The application security methodologies we have discussed so far have their specific application fields and therefore, can help test the application at a specific point in time throughout the CI/CD pipeline. However, achieving overall visibility in any of those specific phases can be challenging, resulting in a lack of knowledge or tool managing overhead, especially in a smaller team.

ASTO is a new category of the application security field introduced by Gartner in 2017, and it stands for Application Security Testing Orchestration.

The purpose of an ASTO tool is to integrate security tools across all the phases of the entire software development life cycle, as shown in *Figure 5.11, SSDLC* (Secure Software Development Life Cycle), to fully enable DevSecOps processes.

As an orchestrator, an ASTO tool should be able to interact with any single security tool at any given point in time of the SDLC via simple API calls, unlocking the potential of a comprehensive **Secure Software Development Life Cycle** (**SSDLC**).

https://t.me/learningnets

Readers should think of an ASTO system as what Kubernetes is to containers or what Rancher is to Kubernetes: a tool that can manage other tools.

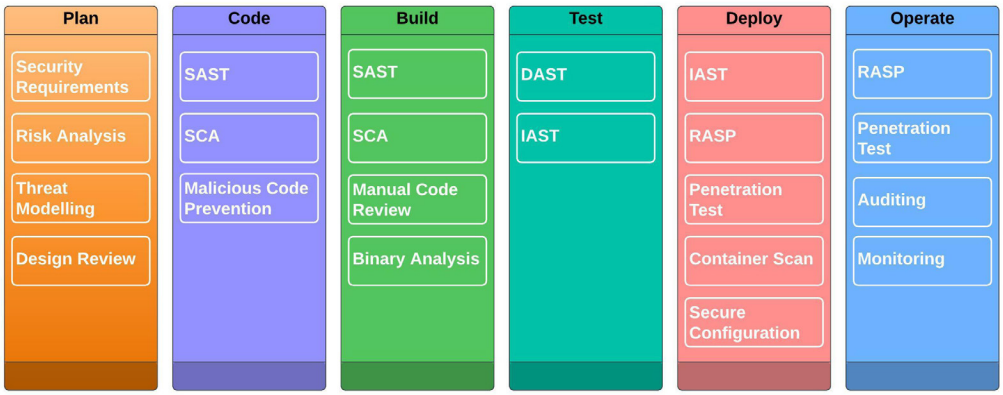

***Figure 5.11**: SSDLC*

ASTO tools are relatively new, but the market has proposed interesting solutions over the last few years, such as Synopsys Intelligent Orchestration, GitLab DevSecOps, Checkmarx Application Security Platform, and Snyk Application Security. But among all of these, **Kondukto.io** stands out for the clear approach in applying security principle, tools and best practices at the right stage of the pipeline. Unfortunately, none of the mentioned platforms have an open-source or community edition, but Snyk Application Security has a free tier up to 100 container tests per month.

Shift left

As discussed in the previous section, modern software development life cycle processes are managed using CI/CD tools, and each specific phase has security mechanisms to help improve the security of the entire process. The SDLC is secure when each critical phase of the development process meets the security requirements, in this case, we can talk about SSDLC.

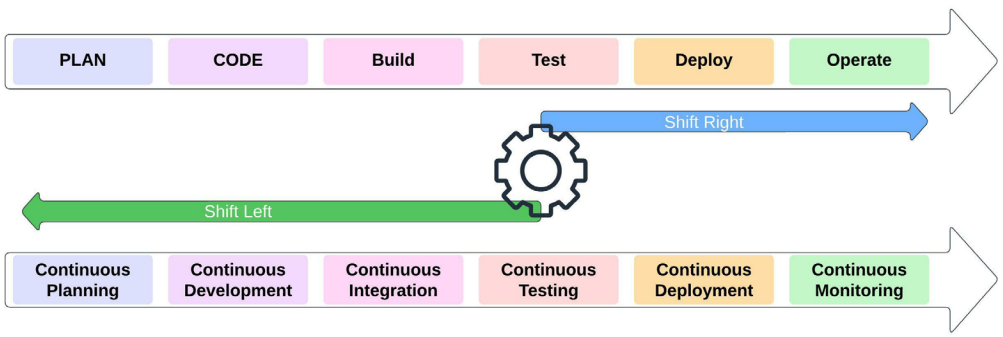

***Figure 5.12**: SDLC*

https://t.me/learningnets

Readers should consider SDLC as a process that starts from a specific point in time in a specific phase, planning, and extending in time from left to right whenever a new phase is reached. This is visually expressed in Fig*ure 5.10, SSDLC*, and in *Figure 5.12, SDLC*. Being able to apply security at an earlier stage of the software development life cycle (shift left) means being able to elevate the security of the overall development process and increase the security posture of the application. The purpose of shifting left is not only to move the security testing at an earlier point in time within the software development life cycle but also to plan to apply security testing from an earlier phase throughout the entire SDLC.

A less known Agile or DevOps methodology is **Shift Right**, which applies testing and monitoring procedures in production to help DevOps uncover unexpected malicious scenarios or activities that may not have been detected within previous phases of the SSDLC.

When applying both methodologies, especially in containerized environments, the results could be an overload of vulnerability detections that could add friction to the process and slow down the development life cycle. The common way to address this issue is to define the DevSecOps maturity level within the organization and understand the balance between development and security.

Remediation

In early 2019, a security misconfiguration in an AWS cloud account led to the loss of sensitive information from the Capital One Financial Corporation, also known as the Capital One Hack. The hack involved over 100 thousand **Social Security Numbers (SSNs)** and 100 million people's credit card information. No details were disclosed about the specific vulnerability exploited by the attacker, but all the investigation information gathered by the FBI was leading to a cloud misconfiguration in the Capital One AWS IAM service for WAF role. Readers curious about the case can read more about it at **https://www.justice.gov/usao-wdwa/press-release/file/1188626/download.**

A **Cloud Security Posture Management** (**CSPM**) solution detects misconfigurations in cloud environments that could expose the infrastructure of the containerized application to potential risks and attack vectors. This kind of solution can recommend remediations or automatically apply security best practices based on the organization's policies or well-known framework security standards.

One of the tools we discussed in the previous chapter, Trivy from Aqua Security, has recently added the CSPM capability to its scan system specifically for AWS, becoming the Aqua Wave security solution. Aqua Cloud Security Posture Management is based on three interesting features:

- Multi-Cloud Visibility

https://t.me/learningnets

- Rapid Remediation
- Cloud Native Security

The technology used by Trivy is based on another tool that Aqua Security acquired in 2019: CloudSploit. The open-source version could be running as a Docker container simply cloning the mentioned git reop and running Docker build. *Table 5.5, CloudSploit Overview*, provides a recap of the various features of CloudSploit:

Type	Description
Cloud	AWS, Microsoft Azure, Google Cloud Platform, Oracle Cloud
Compliance	HIPAA, PCI, CIS Benchmarks
Output	CSV, JSON, XML
Plugin	A set of security controls or requirements required to analyze a specific environment or system
Remediation	A set of actions needed to perform remediations in consideration of the scanning findings

***Table 5.5**: CloudSploit Overview*

The most famous commercial alternative to CloudSploit currently in the market is CrowdStrike CSPM, with its container security and runtime protection, containerized application protection from the build time to the runtime following the SDLC, breach prevention, automate response and behavioral profile with drift detection.

Managing privileges

Secrets are privileged credentials used to perform authentication when privileged users need to access sensitive container applications or data. There are several types of secrets, but let us summarize them in *Table 5.6, Secrets*:

Type	Description
User credentials	Username and password combination
Database strings	Secret used to establish a connection between the application and the database
Keys	Cryptographic keys establish secure communication
Cloud Service Credentials	Authentication requested to access cloud resources or data
API keys	Secret requested to identify an API source
Tokens	API access request for users

***Table 5.6**: Secrets*

https://t.me/learningnets

Secrets must never be stored inside a container; it is critical to adopt a system that manages secrets and provides them to the container when requested. Container orchestrator systems like Kubernetes often have a built-in secret management system, but it also possible to adopt specific cloud provider secret management, like AWS Secret Manager and Azure Vault, or plug either in the cloud or on premises open-source solution, such as Hashicorp Vault.

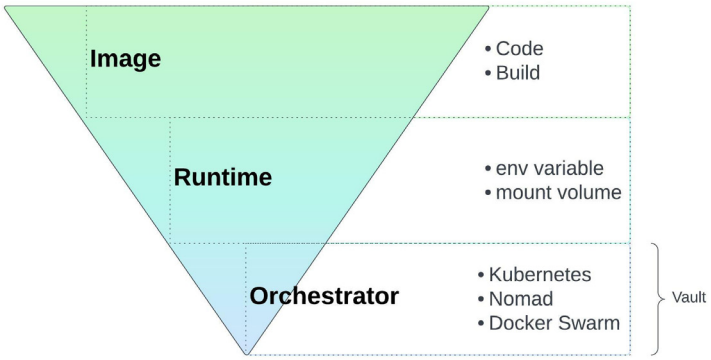

***Figure 5.13**: Secrets*

Figure 5.13, Secrets, illustrates the three methodologies to pass a secret to container, each with a progressive decreased risk exposure.

When outlining the features that a secret management should have, few basic handling characteristics should be addressed:

- Encryption at rest and in transit, decryption on the fly, only in memory
- Restrict access only to container that need to retrieve the secret
- Secret rotation and secret revocation
- Auditing

In April 2021, hackers were able to gain unauthorized access to CodeCov Bash Uploader script, thanks to an error in its Docker image creation process that allowed the attackers to extract the credentials needed. This address, **https://about.codecov.io/security-update**, contains the full incident report. At the top of the reverse pyramid in *Figure 5.12*, there is the container image where secrets could be stored either in the application source code or as a layer of the image via Docker Build; those secrets are, therefore, available to everyone who has access to the code or can inspect the image.

When the secret is stored at the container runtime level, anyone who can **exec** into the container also has the ability to read the secret, and it also can be exposed in **/proc**. Container filesystems are temporary volumes, which means their ephemeral

https://t.me/learningnets

life has a pre-determined end; still, it is not the right approach to apply security best practices.

Vault provides encryption out-of-the-box, and the secret gets ingested into the container application as a file via exchange of token, meaning there is control over the secret ingestion and eventually, auditing. If the secret changes, for example on rotation needs, the token associated with the secret changes, so Vault revokes the previous file and ingests the new one.

The main difference in the **Docker Swarm** secret management system is that while **Vault** can be used with any container orchestrator, Docker secret management works only with Swarm; that said, it doesn't not provide an easy secret rotation mechanism.

HashiCorp Vault is the most known access management tool due to the famous cousin Terraform system from the same company, but there are alternatives out there worth mentioning, such as Delinea Secret Server, CyberArk Privileged Access Management, BeyondTrust DevOps Secrets Safe and Symantec Privileged Access Management.

Penetration testing

Penetration testing, also known as pen test, is a simulated and authorized attack on a system aiming to evaluate the security posture and identify weaknesses that could be exploited by a hacker. There are typically three kind of penetration tests: white box, grey box and black box.

In a white box penetration test, all the information related to the application or system is provided to the tester in advance; the tester acquires knowledge about the application business logic, the underlying infrastructure, the programming language and deployment mechanism. In a black box penetration test, the tester has the minimum level of information necessary to start the attack procedure, such as the URL endpoint or the hosting platform. This approach is the closest to a real-world attack simulation scenario. A grey box penetration test is a middle way between the previous two approaches, and the level of information provided varies according to the agreement between the tester and the organization.

Penetration tests are considered one of the cybersecurity exercises belonging traditionally to the **Red Team**. In cybersecurity, a red team aims to identify, test and assess the vulnerabilities of a given system or application; in other words, Red Team is synonymous of attackers.

Attack frameworks are often used as base model for threat modelling, a process by which threats can be identified, enumerated and prioritized. The most common ones are as follows:

https://t.me/learningnets

- **Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation or privilege (STRID)** created by Microsoft in the late '90s
- **Process Attack Simulation Threat Analysis (PASTA)**
- **Visual Agile Simple Threat (VAST)**
- **Linkability, Identifiability, Non-repudiation, Detectability, Disclosure of Information, Unawareness, Non-compliance (LINDDUN)** a privacy threat model
- **Confidentiality, Integrity, and Availability (CIA)**

Threat modelling is traditionally considered an element of defense and therefore, belonging to the **Blue Team**. The OWASP project has an open-source, very interesting threat modelling tool based on STRIDE, LINDDUN and CIA aiming to provide threat modelling diagrams called **OWASP Threat Dragon**. There are several open-source attack systems that the red team can use to conduct simulated attacks, such as **Mitre Caldera**, **Red Canary Atomic Red**, **Uber Metta** and **Endgame Red Team Automation.** Most of these can be used with penetration tools like **Metasploit**, **nmap**, **Wireshark**, and **ZAP**.

The Mitre has also defined several other attack frameworks, such as for Windows, macOS, Linux, Cloud, Mobile, and Network, but it has defined an attack framework for containers in their **Mitre ATT&CK Container Matrix**, which is also being used as the attack framework for container orchestrator like Kubernetes, as shown in *Table 5.7, Mitre Containers Matrix*. You can visit **https://attack.mitre.org/matrices/enterprise/containers** for more information.

Type	Description
Initial Access	• Exploit Public-Facing Application • External Remote Services • Valid Accounts
Execution	• Container Administration Command • Deploy Container • Scheduled Task/Job • User Execution
Persistence	• External Remote Services • Implant Internal Image • Scheduled Task/Job • Valid Accounts

https://t.me/learningnets

Privilege Escalation	• Escape to Host • Exploitation for Privilege Escalation • Scheduled Task/Job • Valid Accounts
Defense Evasion	• Build Image on Host • Deploy Container • Impair Defenses • Indicator Removal on Host • Masquerading • User Alternate Authentication Material • Valid Accounts
Credential Access	• Brute Force • Steal Application Access Token • Unsecured Credentials
Discovery	• Container and Resource discovery • Network Service discovery • Permission Groups discovery
Lateral Movement	• Use Alternate Authentication Material
Impact	• Endpoint Denial of Service • Network Denial of Service • Resource Hijacking

***Table 5.7**: Mitre Containers Matrix*

Among all the tools mentioned in the preceding table that use the Mitre Attack framework for Containers, **OWASP ZAP** (**Zed Attack Proxy**) is likely the most widely known open-source resource. The goal of the **ZAP** system in penetration mode is to break into the target system and start a denial-of-service attack. **ZAP** works out the penetration test in three phases: explore, attack and report.

The first phase includes exploring attempts to understand and learn about the target system and gathering information like software type, endpoint, programming language, patching level; the second phase is executed trying to break into the system using known vulnerabilities; in the last phase, a report highlighting results of the penetration test is created, including exploitable vulnerabilities and severity.

ZAP is a very flexible system, as it can be integrated into CI/CD systems, it provides a command line interface for easy automation processes and an API interface.

https://t.me/learningnets

Third-party components

Package managers and modules are one of the most important parts of any programming language. Many of today's languages, like Golang, Python, Java or NodeJS, can install additional resources to enhance and streamline the development process. It is not trivial that many of the routine or functions that are common to many already have a pre-determined set of instructions free to everyone aiming to let developers focus on the customer code they are working on.

Then, downloading packages for that purpose significantly increases the risk associated with enhancing the coding experience. Attackers have found new attack vectors by inducing developers in downloading misspelled packages; this type of attack on third-party components is known as **dependency confusion**.

There are other packages aiming not to cover any of the application's internal needs, such as in relation to a specific function; therefore, the developers would not want to integrate those inside the application because it would make coding heavier, but they can be used externally as plug-ins or external modules, like logging for example.

This was the case the popular Java logging framework **Log4j. Log4j** is one of the most popular background components running in many modern web applications. Its only purpose is to log the application behavior. **Log4j** was one of the most dangerous vulnerabilities discovered; NIST applied to CVE-2021-44228 a severity of Critical scored with 10 out of 10. **Log4j** had a massive impact on the industry at the end of 2021 and beginning of 2022, because was deployed in private organizations but also in some US government agencies.

Third-party components, also known as supply chain security, are hard to spot, especially in complex environments where multiple teams are working on different parts of the same application and can affect all the layers of the software development life cycle. The Log4j incident brought to the attention of the industry the importance of having defined Software Bill of Materials (SBOM) as a key element of the SDLC. SBOM is a list of libraries, modules and components that are required to build or execute any given software and the structured supply chain relationship between them. Key elements of the security guardrails to protect against supply chain attacks are as follows:

- Implement only trusted dependencies
- Scan open-source software
- Patch regularly

Docker is working on a **docker sbom** directive aiming to provide a list of packages related to a specific application for container. It can potentially enlist all the components that the application needs or were used to build it. At the container image level, this also includes the operating system and language-specific packages.

https://t.me/learningnets

Note: The Docker sbom command is still experimental.

The following is an example:

```
$ docker sbom neo4j:4.4.4
Syft v0.42.0
 ✓ Loaded image
 ✓ Parsed image
 ✓ Cataloged packages      [348 packages]

NAME                        VERSION                    TYPE
...
bsdutils                    1:2.36.1-8+deb11u1         deb
ca-certificates             20220614                   deb
...
log4j-api                   2.17.0                     java-archive
log4j-core                  2.17.0                     java-archive
...
```

The **docker sbom** command is based on **Syft**, a tool introduced in *Chapter 4, Securing Container Images and Registries,* developed by **Anchore**. Syft is a flexible tool that can create a map of the libraries, dependencies and packages of an application, starting from the container image in standards as Docker Images, **OCI** (**Open Container Initiative**) and Singularity. It can scan within the container image for the packages listed in *Table 5.8, Syft*:

Description	Description
Image scanning	dpkgdb, apkdb, dotnet-deps, portage, ruby-gemspec, python-package, php-composer-installed Cataloger, rpmdb, javascript-package, java, go-module-binary, alpmdb
Directory scanning	alpmdb, apkdb, conan , cocoapods, dpkgdb, hackage, rpmdb, python-index, , dotnet-deps, python-package, php-composer-lock, javascript-lock, java, java-pom, go-module-binary, portage , go-mod-file, rust-cargo-lock, dartlang-lock, ruby-gemfile

***Table 5.8**: Syft*

https://t.me/learningnets

It supports various programming languages like C, C++, .Net, Go, Objective-C, Java, JavaScript, Haskell, PHP, Python, Ruby, Rust, Swift on several systems, like Alpine, Debian, RedHat, and Jenkins for CI/CD integration.

Conclusion

In this chapter, we discussed the various aspects of container application security, including specific examples of how to protect application containers at runtime. We also discussed threat intelligence methodologies, and continuous integration and continuous deployment implementation in container application automation contexts, defining them as important parts of the secure software development life cycle when shifting the security either on the left or on the right of the cycle.

In the last part of this chapter, we deepened our knowledge on remediation and privilege management, exploring container attack framework and an application's third-party components.

In the next chapter, we will learn about monitoring container and security.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

https://t.me/learningnets

Chapter 6
Secure Container Monitoring

Introduction

Security information monitoring is the process of collecting and analyzing information aiming to detect potential security threats. In **Information Technology** (**IT**), monitoring refers to networking traffic, and from the security perspective, traditionally, monitoring takes two main aspects:

- East-West traffic
- North-South traffic

East-West traffic unfolds typically as horizontal network traffic, a network communication between two or more physical or virtual servers (for example, a web server connecting to a database); routers, firewalls, or more general components of a data center or cloud environment are all examples of east-west traffic network communication. For instance, a communication between data centers could be also considered "east-west traffic" (redundancy), and communication between availability zones to use a more modern concept in relation to cloud systems. North-South traffic is typically referred as vertical network traffic or any communication between devices of a data center and an external system, which is physically located outside the data center's (or cloud) boundaries. A clear example is an external client requesting connection to a system running within the data center or cloud system:

https://t.me/learningnets

- The network traffic generated by the client that is entering the data center or cloud system from outside through a perimeter device like a router or firewall, is denominated to as **southbound traffic**.
- The network traffic generated by a system running within the data center or cloud system moving outward, in response to a client request, is denominated to as **northbound traffic**.

For visual understanding of these concepts, see *Figure 6.1, Network Traffic*:

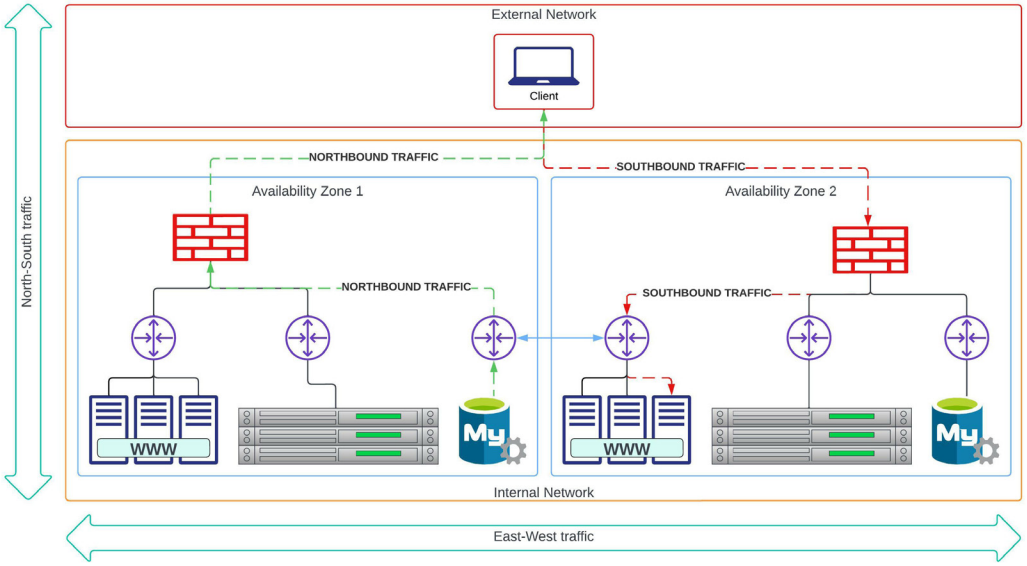

Figure 6.1: *Network Traffic*

There is a tendency to trust east-west traffic because it is generated within the boundaries of the internal network or the known network perimeter, but due to the extensive usage of virtualized and containerized systems in modern applications, east-west traffic has increased exponentially. The traffic increase is an obstacle to the capability of any monitoring system to provide insightful metrics and logs, even more, when these are collected for security purposes.

Structure

In this chapter, we will discuss the following topics:

- Container activity
    - Docker Engine Monitoring
    - Containers Monitoring

https://t.me/learningnets

    - Host Monitoring
    - Application Monitoring
- Workload observability
- Anomaly detection
- Externalize logs
- Alerting
- Topology visualization

Objectives

Monitoring containers requires insights and visibility at multiple layers of the container stack. This chapter aims to provide security best practices for monitoring containerized applications, container runtime and host servers.

Container activity

The same physical server running Docker could have a dozen containers running on top of the host server, generating much more virtual traffic than the only physical network interface card installed on the bare metal would. Among the many monitoring systems in the market, such as Datadog, Dynatrace, SolarWinds Server and Application Monitor, Splunk, Sysdig, New Relic, **Elasticsearch Logstash and Kibana** (**ELK**), **Prometheus** is the only one that has been graduated by the Cloud Native Computing Foundation. Refer to the following figure:

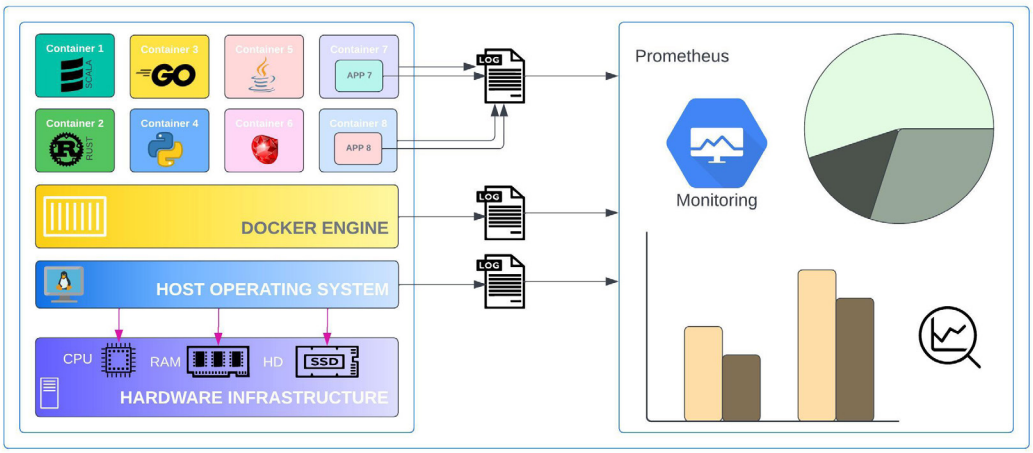

Figure 6.2: *Container Stack Monitoring*

https://t.me/learningnets

Container observability is essential to maintaining a healthy running environment and enhancing security best practices, including all the components of the container stack, as illustrated in *Figure 6.2, Container Stack Monitoring*: Docker engine, containers, host server, and applications running on top of the containers.

Docker engine monitoring

The **dockerd** daemon can be configured to enable metric ingestion into Prometheus; it needs a Docker Swarm running under the hood and reconfiguring the Docker daemon by adding the **--metrics-addr** parameter. The Docker Swarm configuration can be retrieved from *Chapter 3, Container Stack Security*, the *Network Security* section, and there is no need to reset the Docker environment, as the parameter is not adding conflicts to the dockerd daemon executability. To modify the dockerd daemon in a **systemd-based** environment, this is likely to be the most common approach; edit the **/etc/systemd/system/docker.service.d/override.conf** file to look as follows:

```
1. [Service]
2. ExecStart=
3. ExecStart=/usr/bin/dockerd -H unix:///var/run/docker.
   sock -H fd:// -H tcp://0.0.0.0:2376 --tlsverify --tlscacert=/
   home/luigi/ca-public-key.pem --tlscert=/home/luigi/server-
   cert.pem --tlskey=/home/luigi/server-key.pem --metrics-
   addr=192.168.1.226:9323
```

Reload **systemctl** and restart the Docker service with the following commands:

```
1. $ sudo systemctl daemon-reload
2. $ sudo systemctl restart docker.service
```

On the manager node, create the **prometheus.yml** file and add the following code:

```
1. # Prometheus config
2. global:
3.   scrape_interval:     15s
4.   evaluation_interval: 15s
```

https://t.me/learningnets

```
  external_labels:
      monitor: prometheus-monitor-docker'
rule_files:
  # - "first.rules"         # No rules for this example
  # - "second.rules"        # No rules for this example
scrape_configs:
  - job_name: 'prometheus'  # Establish reachability for
  Prometheus system
    static_configs:
      - targets: ['192.168.1.226:9090']
  - job_name: 'docker'   # Connects the target to the dockerd daemon
    static_configs:
      - targets: ['192.168.1.226:9323']
```

To spin up the Prometheus system as a service on the swarm manager node, run the following:

```
$ sudo docker service create \
--replicas 1 \
--name my-prometheus \
--mount    type=bind,source=/home/luigi/prometheus.yml,destination=/
etc/prometheus/prometheus.yml \
--publish published=9090,target=9090,protocol=tcp \
prom/prometheus
```

Prometheus is available at the published port 9090 of the IP address associated with the **Docker** swarm manager node. This is verifiable by browsing the Prometheus server at the server address defined in the **override.conf** file at **http://192.168.1.226:9090/targets**, where the endpoint of the Docker target shall be signed with an **up** green

https://t.me/learningnets

state. *Figure 6.3, Docker Engine Graph*, depicts an example of Docker engine network activity using the **`engine_daemon_network_actions_seconds_count`** metric:

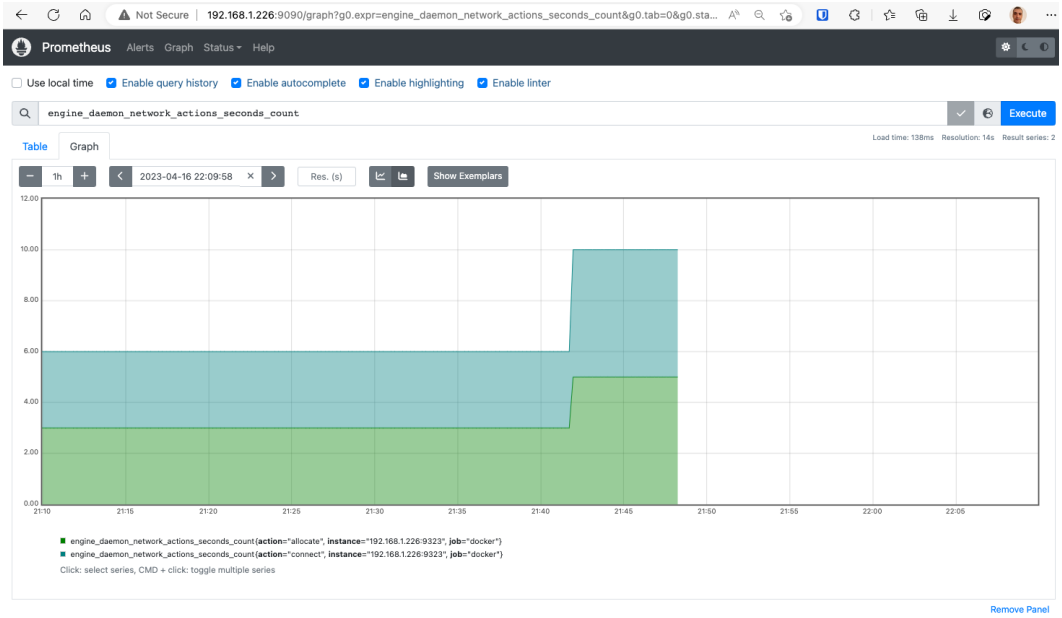

Figure 6.3: *Docker Engine Graph*

The output graph could be flattened out if the Docker host has no container network activity; it is possible to spin up a few containers to generate some network traffic to populate the specific metric, such as the **`ping_service`**.

```
1. $ sudo docker service ls

2. $ sudo docker service rm hm3vsuhz0wzz
```

To terminate the Prometheus instance, list the running Docker services, and then invoke the **rm** command as illustrated in the preceding two commands. All the browsable metrics ingested by the Docker engine into Prometheus have **`engine_`** as the suffix.

Containers monitoring

A Docker Swarm service is not the only way to spin up a containerized Prometheus system and expose the various metrics related to the host machine and the containers running on top of it. That said, it can help redistribute the CPU load. A very popular utility that works with Prometheus out-of-the-box is **Container Advisor** (**cAdvisor**), a utility created by Google to monitor containers, aiming to help analyze and expose resource utilization and performance data. Container Advisor works in a plugin kind of approach, conceptually similar to how Grafana works with Prometheus. To

https://t.me/learningnets

scrape metrics that can be consumed by cAdvisor, Prometheus needs the following **`scrape_configs`** parameter in the **`prometheus.yml`** file:

```
scrape_configs:
- job_name: container_advisor
  scrape_interval: 5s
  static_configs:
  - targets:
    - cadvisor:8080
```

Create a **`docker-compose.yml`** file to spin up the Prometheus system, including the cAdvisor system exposed on the default metric port 8080, and the Redis data structure system to queue up the metrics for delivery, as per the following code:

```
version: '3.2'
services:
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    ports:
    - 9090:9090
    command:
    - --config.file=/etc/prometheus/prometheus.yml
    volumes:
    - ./prometheus.yml:/etc/prometheus/prometheus.yml:ro
    depends_on:
    - cadvisor
  cadvisor:
    image: gcr.io/cadvisor/cadvisor:latest
    container_name: cadvisor
    ports:
    - 8080:8080
    volumes:
    - /:/rootfs:ro
```

https://t.me/learningnets

```
    - /var/run:/var/run:rw
    - /sys:/sys:ro
    - /var/lib/docker/:/var/lib/docker:ro
    - /var/run/docker.sock:/var/run/docker.sock:rw
  depends_on:
    - redis
redis:
  image: redis:latest
  container_name: redis
  ports:
    - 6379:6379
```

The cAdvisor web user interface is available at the **`192.168.1.226:8080`** address. To query container metrics, Prometheus provides the graph expression browser at **`192.168.1.226:9090/graph`**, and to explore specific container stats and graphs, the web interface is available at **`192.168.1.226/docker/<container_id>`**, as illustrated in *Figure 6.4, Container Metrics*. The IP address 192.168.1.226 has been utilized as part of the exercise in this book but can, of course, vary according to the specific platform configuration.

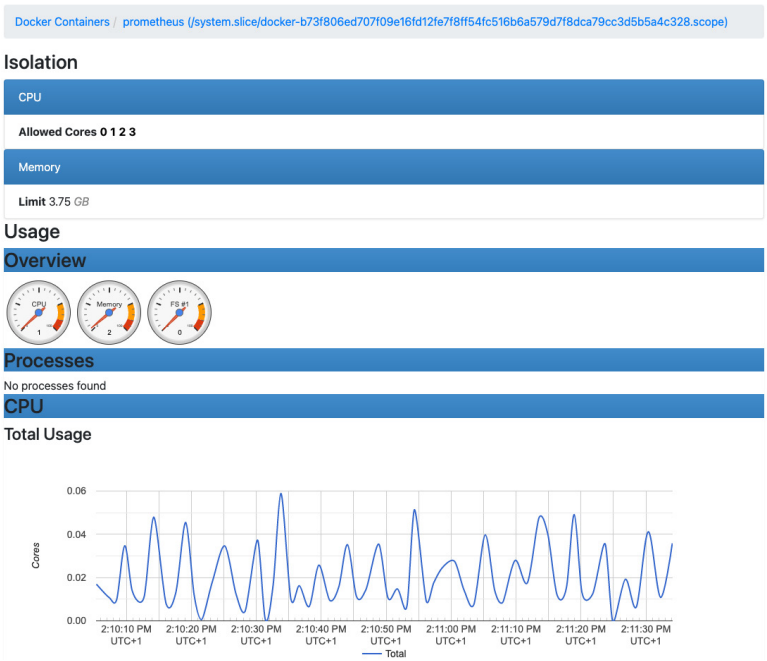

Figure 6.4: *Container Metrics*

https://t.me/learningnets

Interestingly, from the security standpoint, **cAdvisor** can supply metrics related to the **Control Groups**, a feature of the Linux kernel that limits and isolates the hardware resources of the host server that are virtualized into the container stack, as observed in *Chapter 2, Hardware and Host OS Security*, and in *Chapter 3, Container Stack Security*:

Type	Description
rate(container_cpu_usage_seconds_total{name="cadvisor"}[1m])	Cgroups CPU usage
container_memory_usage_bytes{name="cadvisor "}	Cgroups RAM usage
rate(container_network_transmit_bytes_total[1m])	Bytes transmitted
rate(container_network_receive_bytes_total[1m])	Bytes received

***Table 6.1**: cAdvisor Expressions*

Table 6.1, cAdvisor Expressions, lists some of the expressions available for this use case. All the browsable metrics detected by the cAdvisor system Prometheus have **`container_`** as the suffix.

Host monitoring

By host monitoring, this section refers to the capability of the Prometheus system to collect a certain type of hardware-related metrics in addition to what the Linux kernel can provide. To achieve host monitoring, Prometheus provides an additional system called **Node Exporter**. Although a container version of the node exporter exists, due to the very nature of the system, a containerized version is not recommended from the security standpoint because it would need low-level machine access to scrape the needed metrics, which would entail elevating the Docker engine access to the underlying operating system. Node exporter can be downloaded, along with various other systems, such as **`mysqld_exporter`** and **`memchached_exporter`**, from **`https://prometheus.io/download`**. At the time of writing this book, node exporter is on version 1.5.0. To install Node Exporter, download the latest version available with **wget**; the file is compressed, so extract the necessary files and launch it as the following instructions suggest:

```
1. $ tar xvfz node_exporter-1.5.0.linux-amd64.tar.gz
2. $ cd node_exporter-1.5.0.linux-amd64
3. $ ./node_exporter
```

Once **`node_exporter`** is up and running, we can redeploy Prometheus. For this task, we are going to modify the **`prometheus.yml`** and **`docker-compose.yml`** files used in the previous section to include the systems and metrics of the Docker engine, the cAdvisor container monitoring, and the node exporter in one single deployment

https://t.me/learningnets

process. The full **prometheus.yml** file includes all the **scrape_configs** sections mentioned:

```
# Prometheus config
global:
  scrape_interval:     5s
  evaluation_interval: 5s
  external_labels:
      monitor: 'prometheus-monitor-docker'
rule_files:
  # - "first.rules"         # No rules for this example
  # - "second.rules"        # No rules for this example
scrape_configs:
  - job_name: 'prometheus'  # Establish reachability to Prometheus system
    static_configs:
      - targets: ['192.168.1.226:9090']
  - job_
  name: 'docker'      # Connects the target to the dockerd daemon
    static_configs:
      - targets: ['192.168.1.226:9323']
  - job_name: node     # Connects the target on the node exporter
    static_configs:
      - targets: ['192.168.1.226:9100']
  - job_name: cadvisor
    static_configs:         # Connects the target to cAdvisor
      - targets:
          - cadvisor:8080
```

For the deployment of the Prometheus monitoring stack, the **docker-compose.yml** file is not going to change with respect to the one used in the previous section, **Containers Monitoring**; by running a simple **sudo docker-compose up** command, Prometheus will pick up the new configuration. In the example in *Figure 6.5, Node Exporter Metrics*, the disk writes metric **node_disk_writes_completed_total** collected on the host is showing the total KB (Kilobytes) of data written to the disk in the last two hours.

https://t.me/learningnets

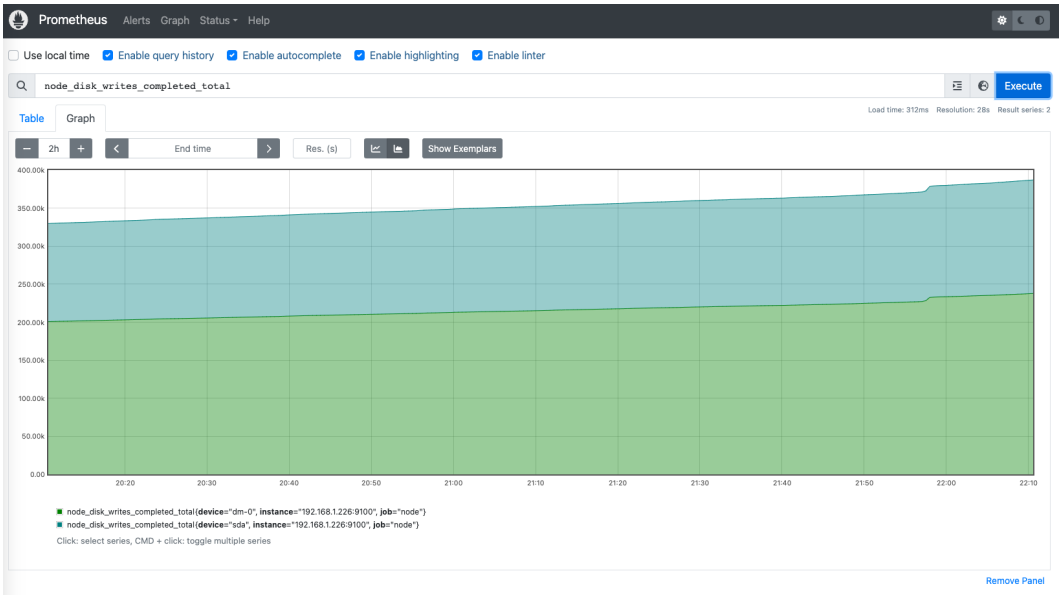

Figure 6.5: *Node Exporter Metrics*

All the browsable metrics ingested by the node exporter into Prometheus have **node_** as the suffix. There is a very rich section of exporters or integration available with Prometheus; some are officially maintained by the Prometheus organization, while others are contributed by third parties, such as AWS for the CloudWatch exporter or Microsoft for the Azure Monitor exporter, and Kubernetes itself. Refer to **https://prometheus.io/docs/instrumenting/exporters/**.

Application monitoring

As per security best practices, it is of utmost importance to gain visibility inside the application itself. This is the last component of the container stack monitoring explained in this section. If the metrics collected exclude the application environment, the visibility inside the container stack is only partial from the monitoring standpoint. Prometheus has various plugins or modules that can be integrated with the main system to collect logs; among these, to monitor applications, Prometheus suggests adding the **client libraries** that expose a service's internal metrics through an HTTP endpoint.

The programming languages officially supported by Prometheus via the client libraries are Go, Java or Scala, Python, Ruby and Rust. There are a number of unofficial client libraries that can be integrated into the application code to expose the internal metrics; among those, it is worth mentioning C or C++, .NET, C#, Node.js, R, Perl and PHP. There is a third option that can be useful if no client libraries are available or if the application code does not expect internal dependencies to be

https://t.me/learningnets

implemented: the **exposition format**. It is based on two main types, the **Text-based format** and the OpenMetrics Text Format, with the latter being Prometheus effort to standardize metric ingestion for the non-client libraries implementation method.

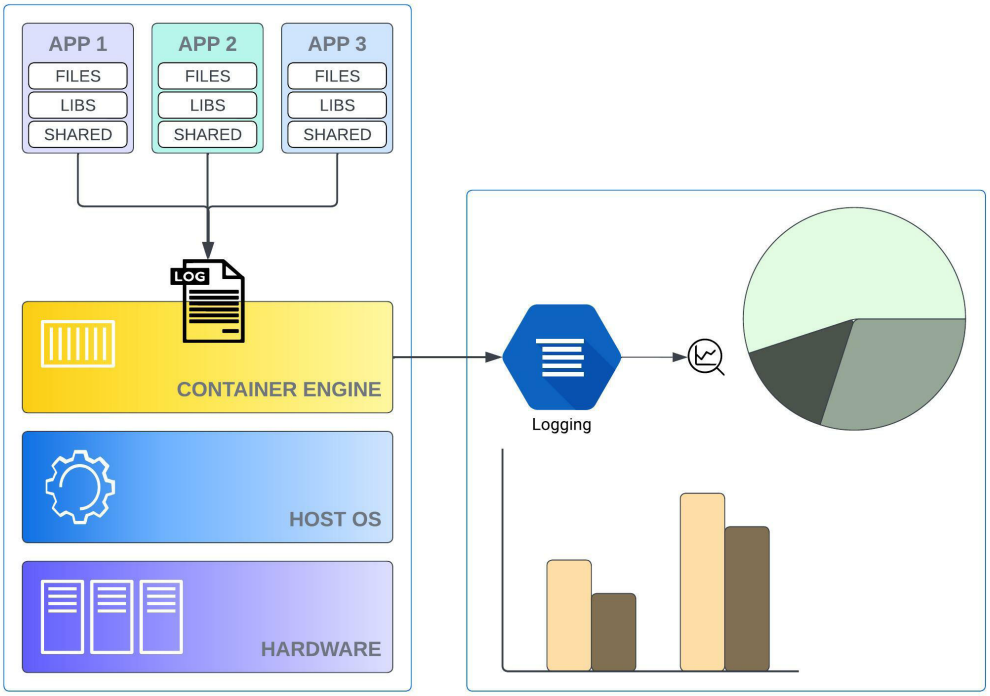

***Figure 6.6**: Application Monitoring*

Figure 6.6, Application Monitoring, refers to enabling logs with respect to the specific programming language the application was built upon, and the ephemeral nature of the container system poses some challenges in tracking metrics and process information. Monitoring systems must be capable of understanding how containers share resources with the operating system through the container engine to be able to monitor the application and the whole container stack effectively. That said, the application needs to be programmed to interact with the container engine to ingest logs that can be consumed further down the line by the monitoring tools.

In order to provide a verifiable methodology on how to expose an application endpoint metric to be collected by Prometheus, the following example will be based on the Python client library. The Prometheus Python client library is distributed as a **pip** (Python Packaging) package, simplifying the installation on systems where Python is already installed. The following procedure illustrates how to install the library on a Linux Ubuntu system, but the installation process is pretty similar on any other nix-based system, including macOS.

https://t.me/learningnets

```
1. $ python3 --version                    # Verify Python is installed
2. Python 3.10.6
3. $ sudo apt install python3-pip         # PIP Installation on Ubuntu
4. $ pip3 --version
5. pip 22.0.2 from /usr/lib/python3/dist-packages/pip (python 3.10)
6. $ pip3 install prometheus-client       # Python Library Installation
7. Collecting prometheus-client
8.   Downloading prometheus_client-0.16.0-py3-none-any.whl (122 kB)
9. Installing collected packages: prometheus-client
10. Successfully installed prometheus-client-0.16.0
```

Upon successful installation of the Python client library, it is possible to create an example Python file that generates a simple request function and exposes the application internal endpoint on HTTP. The metric defined in the following Python script is identified by the **`REQUEST_TIME`** function at point 5, which will ingest **`request_processing_seconds`** metrics into the Prometheus monitoring system:

```
1. from prometheus_client import start_http_server, Summary
2. import random
3. import time
4. # Metric initialization to track time and requests.
5. REQUEST_TIME    =    Summary('request_processing_seconds',    'Requests
   processing time')
6. # Populate function with metrics.
7. @REQUEST_TIME.time()
8. def process_request(t):
9.     """Prometheus Python Client Library example function."""
10.    time.sleep(t)
11. if __name__ == '__main__':
12.    start_http_server(9191)     # HTTP Endpoint exposed on port 9191
13.    while True:                 # Requests generation
14.        process_request(random.random())
```

https://t.me/learningnets

The preceding Python code can be executed either interactively in the Python console or saved in a file and executed via the **python** command. For simplicity, save the code in the **metrics.py** file and run the Python functions with the following command:

```
$ python3 metrics.py
```

By connecting to the Python endpoint at **http://192.168.1.226:9191** declared on line 12 of the preceding code, the script should generate something like the following output:

```
# HELP python_gc_objects_collected_total Objects collected during gc
# TYPE python_gc_objects_collected_total counter
python_gc_objects_collected_total{generation="0"} 233.0
# TLDR OMITTED. . .
# HELP process_virtual_memory_bytes Virtual memory size in bytes.
# TYPE process_virtual_memory_bytes gauge
process_virtual_memory_bytes 1.8151424e+08
# HELP process_resident_memory_bytes Resident memory size in bytes.
# TYPE process_resident_memory_bytes gauge
process_resident_memory_bytes 2.154496e+07
# TLDR OMITTED. . .
# HELP process_cpu_seconds_
   total Total user and system CPU time spent in seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 0.24000000000000002
# TLDR OMITTED. . .
# HELP request_processing_seconds Time spent processing request
# TYPE request_processing_seconds summary
request_processing_seconds_count 11.0
request_processing_seconds_sum 5.801927725995483
# HELP request_processing_seconds_
   created Time spent processing request
# TYPE request_processing_seconds_created gauge
request_processing_seconds_created 1.6819079937132046e+09
```

https://t.me/learningnets

The **request_processing_seconds** metric output of the Python script execution matches the function requirement declared in the **metrics.py** file at point 5. To ingest the metrics created by the Python script into Prometheus, it is necessary to amend the **prometheus.yml** file, adding the Python target into the **scrape_configs** section. Therefore, the file used in the previous section, *Host Monitoring*, will become as follows:

```
1. # Prometheus config
2. global:
3.   scrape_interval:     5s
4.   evaluation_interval: 5s
5.   external_labels:
6.       monitor: 'prometheus-monitor-docker'
7. rule_files:
8.   # - "first.rules"          # No rules for this example
9.   # - "second.rules"         # No rules for this example
10. scrape_configs:
11.    - job_name: 'prometheus'    # Establish reachability to Pro-
   metheus system
12.    static_configs:
13.      - targets: ['192.168.1.226:9090']
14.  - job_name: 'docker'     # Connects the target to the dockerd daemon
15.    static_configs:
16.      - targets: ['192.168.1.226:9323']
17.  - job_name: node         # Connects the target on the node exporter
18.    static_configs:
19.      - targets: ['192.168.1.226:9100']
20.  - job_name: cadvisor
21.    static_configs:         # Connects the target to cAdvisor
22.      - targets:
23.          - cadvisor:8080
24.  - job_name: python         # Connects the target to python endpoint
25.    static_configs:
26.      - targets: ['192.168.1.226:9191']
```

https://t.me/learningnets

The last step of the application monitoring is to redeploy the Prometheus stack using the updated **`prometheus.yml`** file. The Docker compose file does not need adjustment as the **`metrics.py`** file is running externally to the Docker stack.

```
1. $ sudo docker-compose up
```

By connecting to the Prometheus server at **http://192.168.1.226:9090**, it is possible to verify that **`request_processing_seconds`** has been ingested into the monitoring system, as illustrated in *Figure 6.7, Python Client Library*:

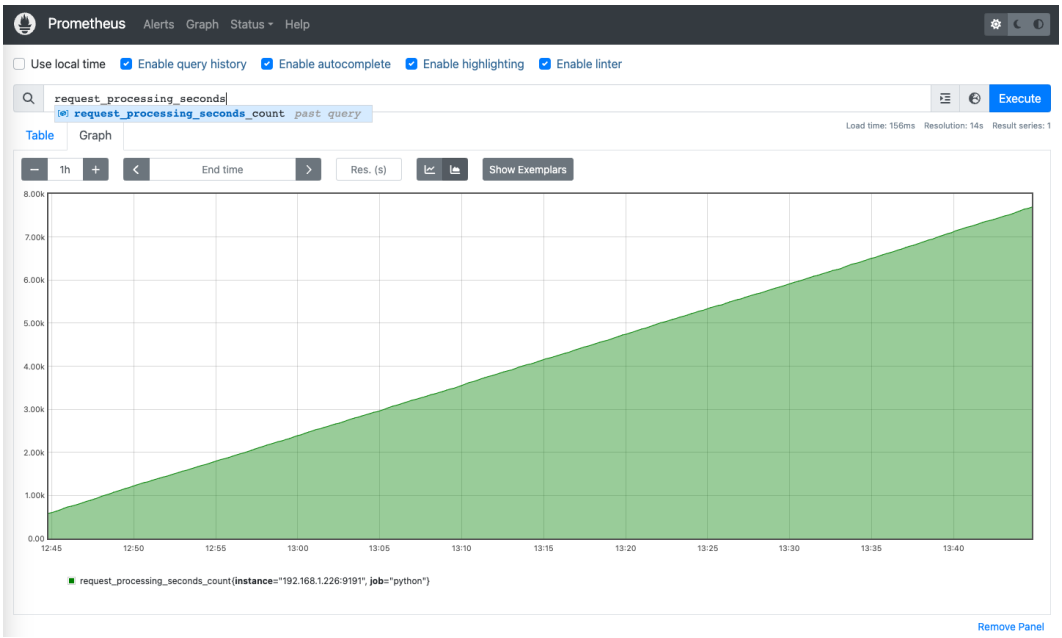

***Figure 6.7**: Application Monitoring*

In this section, we achieved monitoring of the full container stack. The data ingestion from any application or container into a centralized system or repository for post-analysis is also referred to as a **Security Information Management** (**SIM**) tool.

Workload observability

In computing, a workload is any application that runs on a computer, and that computer's capability of ingesting input and processing an output. There is no single set of criteria to classify workloads, but traditionally, they are divided into three main groups: **static and dynamic**, **transactional or batch**, and **analytical**. The static and dynamic group is also the most ancient definition of workload, when compute resources were divided into the following:

https://t.me/learningnets

- Computer systems like operating systems, email systems, hypervisors or any system central to the business need is considered a static workload with none or little change overtime.
- Point in time application like test environments or scheduled jobs are as per their own nature ephemeral, and therefore, are considered dynamic.

The second group is a classic of the mainframe era, but still actual in the financial industry:

- Systems or operating systems built upon the use of real-time kernel adopted by banking, insurance firms, stock exchange markets are considered transactional.
- On-demand data processing in high volumes, such as billing systems, are considered batch workloads.

The third group is likely the most recent type of workload:

- Analytical workloads analyze huge quantity of data, the most appropriate field of application is advanced medical research or biotechnological, and it also the basic concept behind big data and machine learning.

The adoption of the container stack has dramatically increased the density of any type of mentionable workload, but it also adds complexity in securing and monitoring microservices, software as a service application, and serverless compute programs, as shown in *Figure 6.8, Workload Density*:

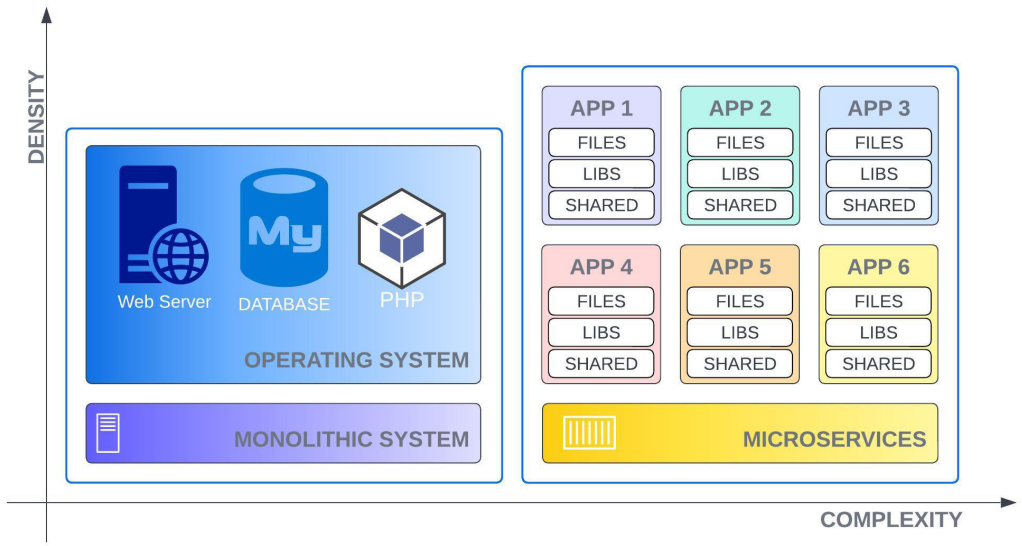

***Figure 6.8**: Workload Density*

https://t.me/learningnets

Container security is, therefore, achieved by applying a layered approach to the applications, containers, the container runtime or the host operating system, because due to the very nature of the container stack, hardware resources used by the applications are virtualized and containerized into containers through the runtime interface, as illustrated in *Figure 6.9, Hardware Abstraction*:

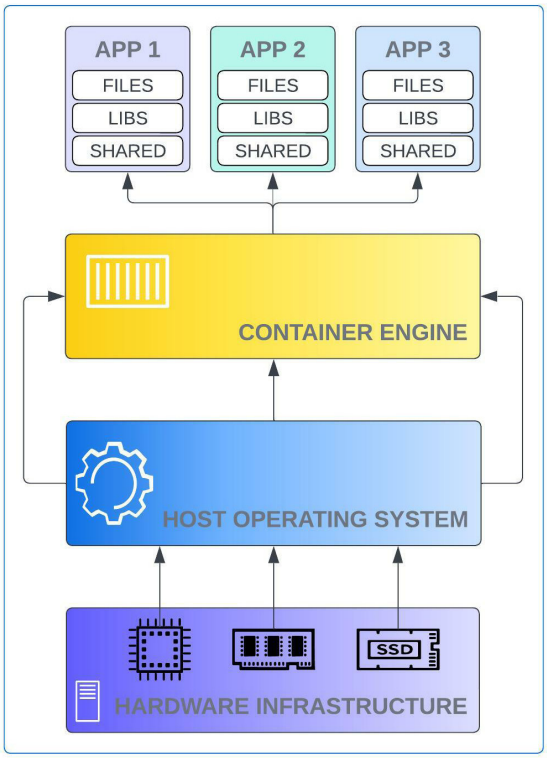

***Figure 6.9**: Hardware Abstraction*

The challenge is, then, to transform the logs and data collected in applying the methodologies exposed in the previous chapter into meaningful information. **Bitdefender**, a Romanian-founded cybersecurity company, has recently come to the market with a new product called **GravityZone Security for Containers**. This feature expands the **GravityZone EDR** (Endpoint Detection and Response) **Cloud** offer, which was initially designed to provide endpoint security solution with insight threat data collection and real-time monitoring.

The peculiarity of the **GravityZone Security for Containers** solution is the implementation of workload security capabilities, and eXtended Detection and Response (**XDR**). The main difference between **EDR** and **XDR** is that while the **EDR** solution is focused solely on endpoint protection, the **XDR** solution aggregates data from all the endpoint agents, providing a higher level of threat intelligence and response mechanism, as highlighted in *Figure 6.10, EDR and XDR*:

https://t.me/learningnets

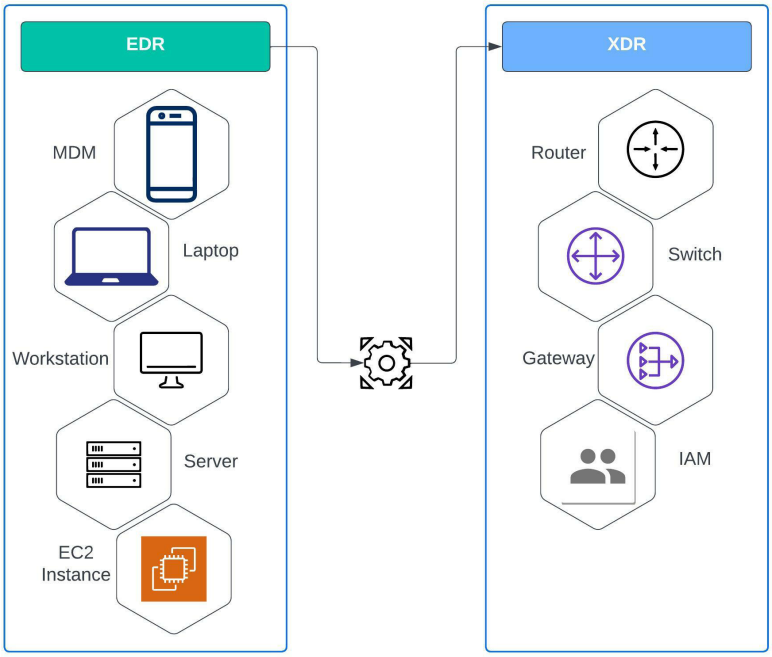

***Figure 6.10**: EDR and XDR*

Bitdefender achieved this by mapping into **GravityZone Security for Containers** the full **Mitre ATT&CK Container Matrix** framework. This framework was discussed in *Chapter 5, Application Container Security*, in the **Penetration Testing** section. To review Mitre framework's matrix, refer to *Table 5.7, Mitre Containers Matrix*.

What **Bitdefender** has achieved is also a field application of what **Gartner** defined as the **Cloud Workload Protection Platform** (**CWPP**). A **CWPP** solution protects the cloud infrastructure workload against security threats, and it should be able to cover and detect several types of workloads either running on-premises or in the cloud, such as containers, databases (including NoSQL), APIs, serverless, and Kubernetes, systems that are traditionally difficult to inspect with a classic monitoring tool. In this context, it feels natural to introduce **Wazuh**. Wazuh has been defined as the open-source security platform that can unify the **XDR** system and a **Security Information and Event Management** (**SIEM**) endpoint protection and cloud workloads in one monitoring tool. *Table 6.2, Wazuh Features*, illustrates some of the more interesting features:

https://t.me/learningnets

Type	Description
Intrusion detection	Scans for malware, anomalies, hidden files, syscalls, stuck processes, and rootkits
Log data analysis	Collects data from the application logs and the operating system; can also ingest data from network devices
FIM (File Integrity Monitor)	Filesystem monitoring, file permission and attributes
Vulnerability Detection	Software inventory data against CVE databases
Configuration Assessment	System settings and application configurations monitoring against security policies
Incident Response	Threat response automation
Regulatory Compliance	Provides security controls to meet industry standards, such as PCI-DSS, HIPAA, GDPR
Cloud Security	API level integration to monitor cloud infrastructure, such as AWS, Azure, and GCP
Container Security	Provides visibility into containers, monitoring volumes, images, running containers, and network settings

***Table 6.2**: Wazuh Features*

It is worth mentioning Wazuh automation capabilities with the full adoption of deploying methodologies via Ansible, Chez, Puppet, and Salt, but also AWS CloudFormation, Docker containers, and Kubernetes.

Anomaly detection

There's no doubt that containers have entirely reshaped the deployment and operational model for the detection and response mechanism. The huge set of metadata produced by container workloads exponentially increases the number of metrics ingested into monitoring tools, and defining rules to detect potential threats is challenging. **Anomaly detection** is one such approach, and it consists of creating a baseline of expected behavior for a container, and then measuring the metrics generated against that baseline. When a drift from an expected behavior is detected, the activity that generated that drift is considered anomalous and must be investigated to fulfil the response to the event. Traditional IT (Information Technology) infrastructure is often based on VMs (Virtual Machines), refer to *Figure 6.11, VMs*, and it provides less overhead and is easier to protect because it implies security via system isolation. However, anomaly detection as a security model is limited in this case because each application runs on a full operating system, so it is hard to distinguish between the metrics related to either the application or the HOST OS. Refer to the following figure:

https://t.me/learningnets

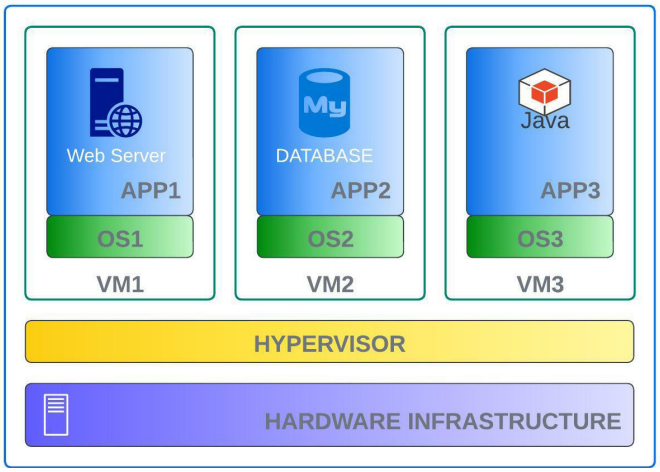

***Figure 6.11**: VMs*

In other cases, to reduce the traffic generated by the quantity of containerized operating systems deployed, multiple applications are deployed on a single host. Although this option could inspire an alternative solution, it would require more time and expertise to configure install multiple applications on the same host. On the other hand, containers are usually running a single application, which often means one process only to monitor. This form factor upholds the anomaly detection efficacy. Here are a few approaches to handle anomaly detection for containers:

- Write anomaly detection policies based on the monitoring approach used in the **Container Activity** section earlier in this chapter.
- Implement a Machine Learning solution in the cloud.
- Integrate Machine Learning system with Prometheus.

The first option leaves the door open to greater flexibility, but it needs a lot of preparation to put down a template that works effectively. Cloud providers have developed integrated system to consume cloud resources that can correlate data metrics produced by their container services, and generate metrics to detect anomalies that can be ingested in machine learning systems.

Microsoft Azure has deployed a container service to use their real-time **Anomaly Detector API** as part of the **Azure Cognitive Service**. The Anomaly Detector can be used offline from the Azure cloud service, it can be downloaded locally but it still needs to be billed via the Azure subscription service. The service is intended to verify the metrics collected by Azure resources deployed either onto **Azure Container Instances**, **Azure Kubernetes Service** or any **Kubernetes** cluster deployed via **Azure Stack**. The installation is straightforward, and it uses Docker commands:

https://t.me/learningnets

```
1. $ docker pull mcr.microsoft.com/azure-cognitive-services/anomaly-
   detector:latest
```

To run the Anomaly Detector system, run the following:

```
1. $ docker run --rm -it -p 5000:5000 --memory 4g --cpus 1 \
2. mcr.microsoft.com/azure-cognitive-services/decision/anomaly-
   detector:latest \
3. Eula=accept \
4. Billing={ENDPOINT_URI} \
5. ApiKey={API_KEY}
```

The preceding Docker command has three additional Microsoft parameters, as explained in *Table 6.3, Azure Cognitive Service*:

Type	Description
Eula	Accept Microsoft End User License Agreement
Billing	This endpoint is utilized to collect billing information
ApiKey	The API Key of the Azure Cognitive Service

***Table 6.3**: Azure Cognitive Service*

Amazon ECS Anomaly Detector is an integration of **AWS EventBridge**, a serverless event bus, with **AWS ECS** (**Elastic Container Service**), storing the EventBridge rules into **CloudWatch** logs. This workflow also leverages the **AWS SNS** and **AWS Lambda** functions for task automation and notification. The concept behind the combination of the preceding services is to use the "near" real-time EventBridge delivery stream to capture all the ECS events at a very granular level. Surfing between the huge number of events generated by an ECS cluster can be challenging and time-consuming, reducing the visibility on the services or tasks that misbehave.

AWS ECS logs four types of events:

- Container instance state change
- Task state change
- Service action
- Service deployment state change

Those events are grouped into three different categories: **info**, **warning** and **error**. AWS ECS Anomaly Detector has a set of predefined rules in EventBridge, as per *Table 6.4, ECS Anomaly Detector rules*:

https://t.me/learningnets

Description	Description
ECS_AD_ServiceActivity	All ECS events and ECS task state
ECS_AD_ServiceAction_Error	Only error and warning
ECS_AD_StoppedTask_Detector	Failed tasks detection
ECS_AD_DeploymentStateChange_Event	ECS deployment state change
ECS_AD_UpdateService_CTEvent	UpdateService API calls via CloudTrail

***Table 6.4**: ECS Anomaly Detector rules*

All the events in the previous table are logged by AWS CloudWatch into the log group called **`/aws/events/ECS_ANOMALY_DETECTOR`**. *Figure 6.12, ECS Anomaly Detector*, shows a visual architecture of the Amazon solution:

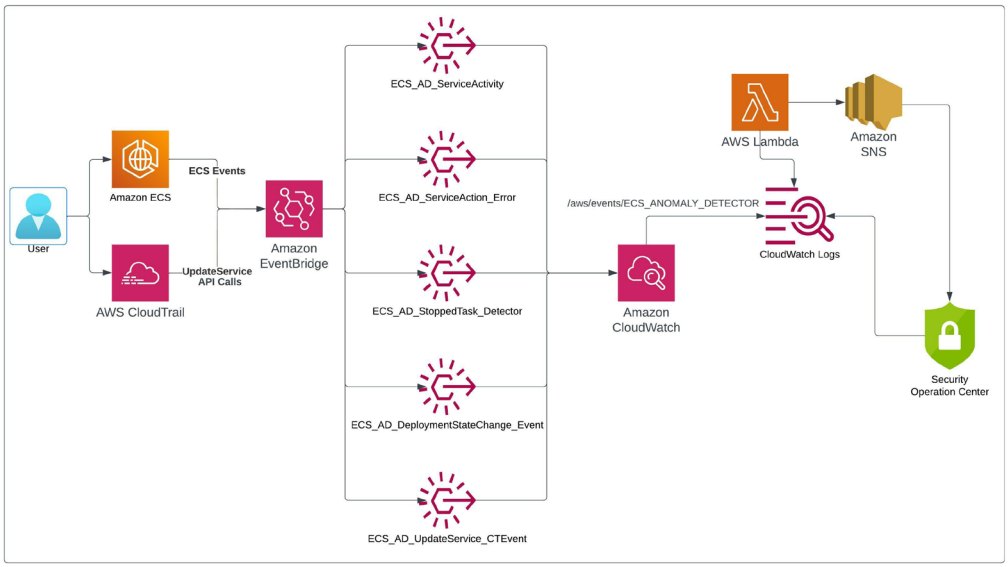

***Figure 6.12**: ECS Anomaly Detector*

The **Artificial Intelligence Center of Excellence** (**AICOE**), a division of RedHat Emerging Technologies, has developed a Machine Learning model to monitor containerized application running on OpenShift through Prometheus. **Prometheus** stores metrics in a time series database. These metrics can present anomalies, i.e., values that are not in line with the expected behavior of the applications or containers; however, due to the quantity of metrics normally ingested, it is challenging to identify those "drifts" within the noise, so the detection is often based on experience or knowledge. With an Artificial Intelligence-based approach, it is possible to train machine learning in consideration of historic metrics aiming to perform valuable "guesses" on what the metrics would likely be in the future, essentially, a prediction model. The most important components of this model are as follows:

https://t.me/learningnets

- Prometheus/Grafana
- Machine Learning Model

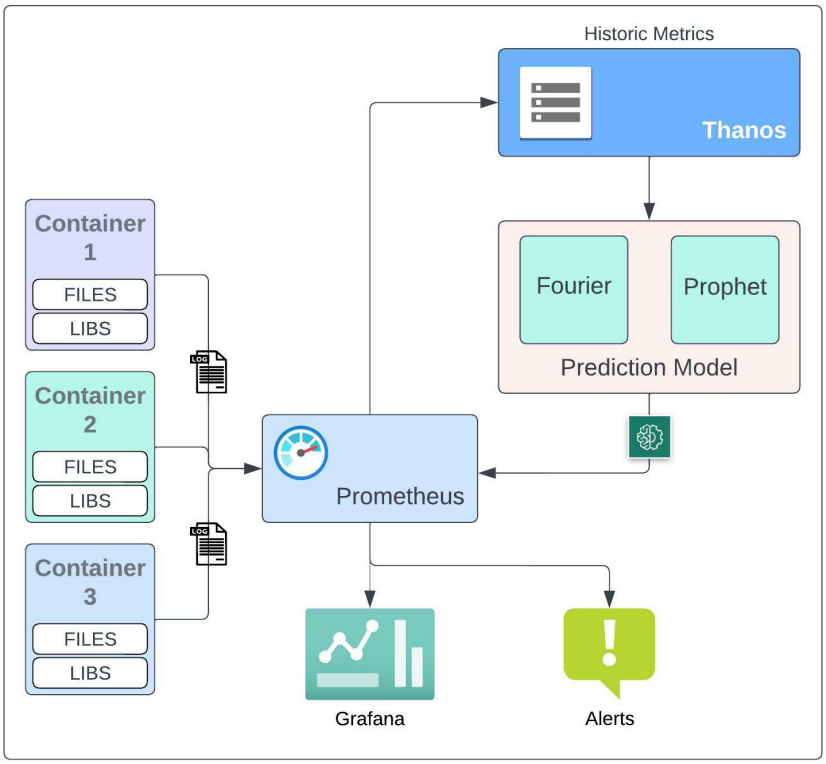

***Figure 6.13**: Prometheus Prediction Model*

The machine learning model in *Figure 6.13, Prometheus Prediction Model*, has three main parts: **Thanos**, **Fourier** and **Prophet**:

- **Thanos** is a **Cloud Native Computing Foundation** (**CNCF**) incubating project that aims to scale the Prometheus querying mechanism of metrics across multiple Prometheus server installations and clusters, with long-term storage metrics capabilities and big data sampling.
- **Fourier**, or "Fourier transform", is a methodical operator that transforms a function in another function; in this case, it helps map metrics from the Thanos system to a frequency domain.
- **Prophet** is a machine learning model developed by **Facebook** that aims to provide automated forecasting process at scale. Prophet will provide a predict metric between an upper and lower value for any given time, in consideration of historic data and modelling the metrics, including non-linear trends.

https://t.me/learningnets

Usually, when looking at metrics, it is common to browse the left part of the graph, looking for something that has already happened. In this scenario, Artificial Intelligence (AI) uses historical patterns to look at the future. The event logging approach described in this section is the foundation of what is called **Security Event Management** (**SEM**) in information security. An SEM tool is the capability of a given system to define an event from the information logged.

Externalise logs

Containers are ephemeral in nature, so any trace of their existence disappears with their termination. Logs are an important part of any diagnostic or troubleshooting technology and must be preserved to ensure visibility. Many security compliance frameworks have specific security logging and monitoring requirements, such as access to logs should be controlled and limited, and the need for log integrity, backup logs to a central server, and a minimum retention period.

Situations in which security breaches have been discovered only several weeks, sometime months after the original attack starting date, are not uncommon. That happens because the attacker has been able to cover their tracks by manipulating the logs: "you do not know what you cannot see". It is a security best practice to not store the logs locally, which refers to keeping the logs on the same machine on which containers are running, and also to keeping the logs in the same network. The goal of this approach is to ensure that if an application or a system is compromised, the logging system is out of the attacker's reach.

As explained in the *Container Activity* section of this chapter, Docker provides logging drivers that can be used to redirect logs to an external logging server. The simplest way to externalize logs is to enable the syslog logging driver. The code to enable the syslog driver is as simple as follows, and it should include the TLS configuration to enable secure communication:

```
{
  "log-driver": "syslog",
  "log-opts": {
    "syslog-address": "udp://1.2.3.4:1111",
    "syslog-tls-ca-cert": "/etc/ca-certificates/custom/ca-key.pem",
    "syslog-tls-cert": "/etc/ca-certificates/custom/server-cert.pem",
    "syslog-tls-key": "/etc/ca-certificates/custom/client-key.pem"
  }
}
```

https://t.me/learningnets

The **National Institute of Standards and Technology** (**NIST**) has published with its Special Publication 800-92, **Guide to Computer Security Log Management**, a series of recommendations and security best practices for managing and securing log data. The **NIST SP 800-92** aims to address the security log management challenges:

- Ensure log integrity during storage and transmission
- Log review
- Log retention

The NIST SP 800-92 highlights the concept of log centralization, the **SIEM**. A SIEM tool has one or more log servers communicating and ingesting information, featuring a combination of a **SIM** (Security Information Management) tool, as described at the beginning of this chapter, and a **SEM** (Security Event Management) tool, as shown in *Figure 6.14, SIEM*:

***Figure 6.14**: SIEM*

A SIEM software is essentially a log aggregator that combines the SIM capability to collect data and the SEM feature to retrieve events from the information collected. Many solutions in the market can satisfy the SIEM requirements today, such as Splunk, DataDog, ELK (Elasticsearch Logstash Kibana), Graylog, Wazuh and SolarWinds. But a SIEM solution has potential to do more from the security perspective; it is not only considered a log and event management tool, but it should also be able to recognize malicious activities and security threats. *Table 6.5, SIEM features*, aims to highlight the most common modern SIEM capabilities:

Type	Description
Single Pane of Glass	A unified dashboard used by the SOC team to provide insights on threat intelligence, anomaly detection, and in-depth visualization
Analytic	Machine learning analytics to gain insight on vast amounts of data
Threat Detection	EDR, Anomaly Detection and Incident Response

https://t.me/learningnets

Forensic	Security events in-depth analysis
Compliance	Security compliance frameworks reporting features

***Table 6.5**: SIEM features*

SIEM tools have become critical components in modern information security structures. AntiVirus/AntiMalware (**AV/AM**) solutions, firewalls, or advanced endpoint protection systems in general are not sufficient anymore against zero-day attacks, where the exploited vulnerabilities are not known to the defense mechanism yet.

Alerting

Alerts are a direct consequence of the logging system. Logs are ingested into the logging server, and the monitoring tool is verifying that the data is not showing anomalies. When an anomaly is detected, the monitoring tool sends out an alert. Today alert mechanisms are multiple, such as email, text message, SLACK or TEAMS, PagerDuty notifications, but all of these can be configured in the monitoring tool. The SIEM system discussed in the previous section, for instance, is the "natural" security system to define criteria in to identify a potential threat and, therefore, to raise an alert.

It is essential to define alert rules based on the severity of the finding. A severity-based alerting process helps reduce noise, providing a marked difference between a high-priority alert and a low-priority alert. These rules are at the core of any incident response procedure. SIEM Alerting is, therefore, the capability of a SIEM system to provide intelligent alerts based on an event-driven monitoring system.

Security Orchestration Automation Response (**SOAR**) technology is the next level of a SIEM solution. A SOAR system brings security in aggregation of monitoring tools, helping coordinate and automate tasks within a single system. Gartner has defined a SOAR system that is designed to satisfy three software capabilities:

- Security Operations Automation
- Threat and Vulnerability
- Security Incident Response

Typically, a SOAR software ingests alerts from systems that a SIEM tool would not cover, such as vulnerability scan software, IoT devices, and cloud security alerts (for example, AWS GuardDuty), and responds to these alerts with automated playbooks with the intent of automatically applying a fix to clear the alert. In consideration of the huge amounts of information collectable from a container stack, and even more from container orchestrator like Kubernetes, surfing the noise of data in search of

https://t.me/learningnets

meaningful events can be daunting. That's why the orchestration and automation aspects of a SOAR system are extremely useful to a Security Operations Center team.

Topology visualization

Despite the due diligence, sometimes it is hard to understand why some containers are not behaving as expected, and monitoring the huge amount of data can be very time-consuming. In this scenario, having a visual representation of the network communication in a container stack could come in handy to gain immediate access to valuable information.

ThreatMapper by **Deepfence** is a Cloud Native threat hunter visualization tool; refer to *Figure 6.15, ThreatMapper*, with the interesting feature of hunting for what Deepfence defines the "hidden threats", ranking them on exploit-risk based criteria.

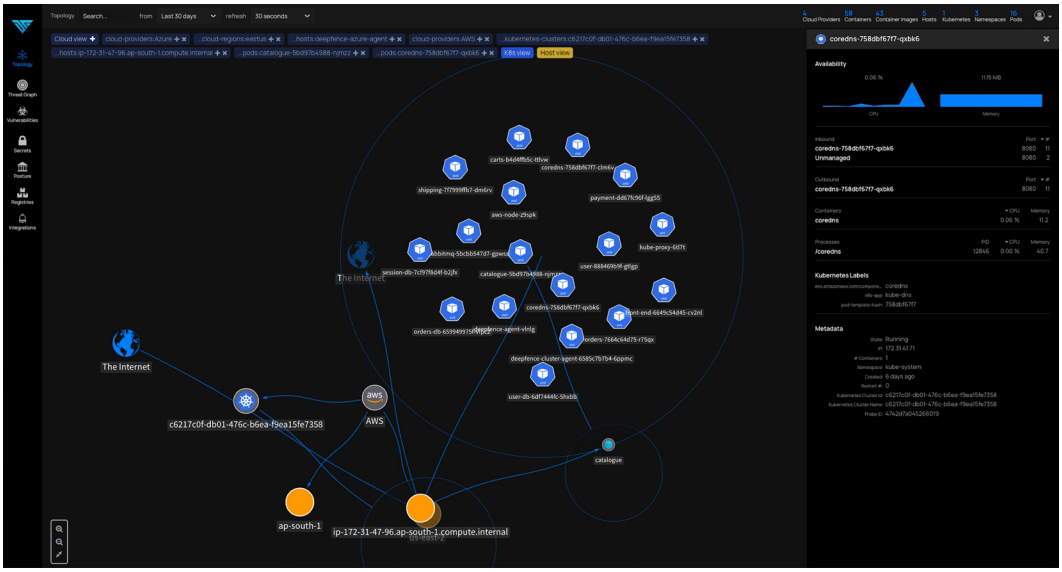

***Figure 6.15**: ThreatMapper*

In relation to containers, ThreatMapper works on the following logic: discover, find threats and actionable information. ThreatMapper inspects infrastructure using the following:

- **CloudScanner**, which, as the name suggests, scans cloud providers via API calls to gather configuration information, and being a risk-based system, compares the acquired information against compliance benchmarks to determine whether a deviation has occurred.
- **Sensor Agents**, which is an agent-based tool that can be installed on the most common platforms, such as Docker, Kubernetes, Amazon ECS, AWS

https://t.me/learningnets

Fargate, and Virtual Machines or Bare-Metal systems, which will report to the ThreatMapper Management Console.

The core system used by ThreatMapper to plot the containers interconnection is called **Scope** by **Waeveworks**. Scope has been specifically designed to map containers and applications running into the target infrastructure to monitor and control microservices, as shown in *Figure 6.16*, *Scope*:

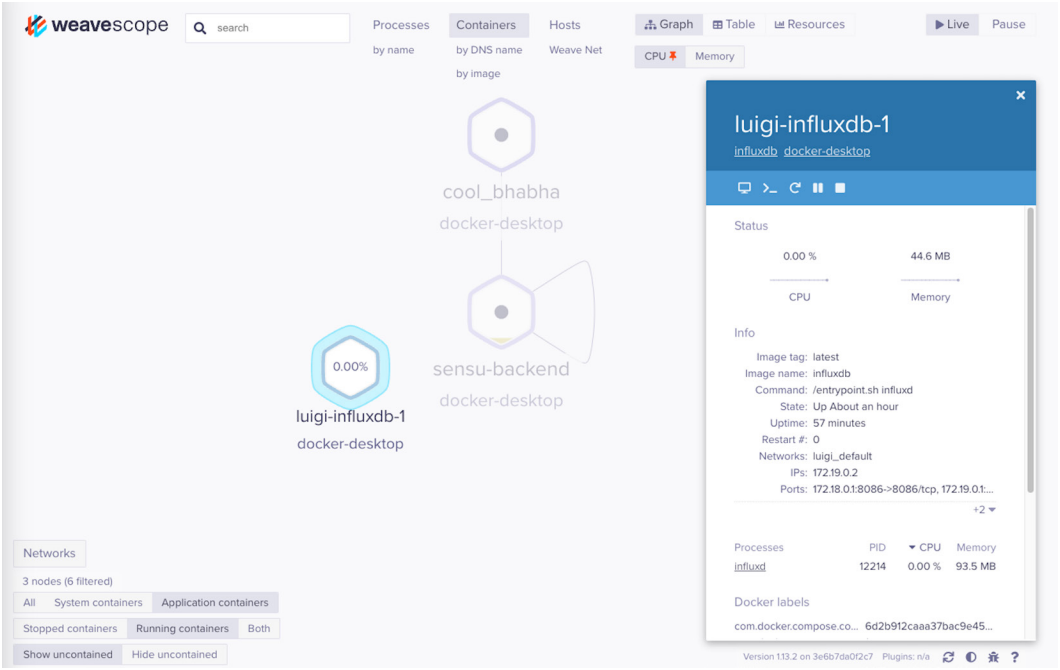

***Figure 6.16**: Scope*

The main characteristics of **Scope** are as follows:

- Understanding container infrastructure in real time
- Metrics and metadata visualization
- Containers interaction, such as start/stop, pause/reboot, and live console

Scope supports a plugin mechanism to extend the capabilities of the system. Among those already available, it is worth noting **Scope Traffic Control** and **Scope Volume Count**. The Scope Traffic Control plugin allows us to change the container's network parameters, and the Scope Volume Count plugin inspects Docker for the total number of volumes mounted for each container.

https://t.me/learningnets

Conclusion

In this chapter, we discussed the various aspects of container monitoring and security, starting from the basic definition of network traffic from the security perspective and moving on to enabling logging at the different layers of the container stack with practice examples on the container runtime and the container itself. We discussed and defined the various types of container workloads and how to observe these with respect to monitoring tools that can cover either EDR or XDR.

We then moved on to define what an "anomaly" is and how to detect it in cloud monitoring platforms that can correlate data intelligence mechanisms and how to use AI (Artificial Intelligence) to apply a prediction model. We also explained why logs should be preserved and how they can be used by different solutions, such as SIM, SEM and SIEM, for different purposes.

Finally, we looked at how to produce intelligent alerting and the capability of a SOAR system to leverage automation to better respond to an incident from the SOC (Security Operations Center) standpoint, and how visualizing the topology of the container stack that can provide great insights into the container network traffic and workload security.

In the next chapter, we will learn about Kubernetes Hardening.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

https://t.me/learningnets

Chapter 7
Kubernetes Hardening

Introduction

Kubernetes, commonly known as K8s, is an open-source orchestrator system for automating, deployment, scaling containers and management of applications running as microservices originally developed by Google to address the need of containerized applications and microservices management at scale.

Originally designed to work natively with Docker through an interface called **Dockershim**, as discussed in *Chapter 3, Container Stack Security*, from release v1.25, Kubernetes does not install the Docker connector by default, in favor of a more "open approach" toward other projects like **containerd** and **CRI-O**, container engines supported by the Cloud Native Computing Foundation. The Dockershim connector is still available, and it will be available for the foreseeable future.

Kubernetes has introduced greater flexibility in containerized platforms when compared to traditional monolithic software infrastructure, but it has also introduced greater complexity in securely managing microservices and the underlying virtualized infrastructure. Kubernetes' common sources of compromise, according to the **Cybersecurity and Infrastructure Security Agency** (**CISA**), department of the

https://t.me/learningnets

National Security Agency (**NSA**), are highlighted in *Table 7.1, Kubernetes Compromise Sources*:

Type	Description
Insider threats	Users, administrators or cloud providers
Supply chain	Container deployment life cycle
Malicious threat actors	Kubernetes' components misconfigurations and vulnerabilities

***Table 7.1**: Kubernetes Compromise Sources*

Getting started with architectural concepts will bring more clarity to the security approach taken in this chapter, along with a high-level overview of the Kubernetes platform and the potential security issues a system of such complexity may introduce. With reference to *Table 7.1, Kubernetes Compromise Sources*, *Table 7.2, Kubernetes Threats Reference Matrix*, illustrates the association between the threats addressed by the CISA Kubernetes hardening guide and the Kubernetes components we will discuss in this chapter and in *Chapter 8, Kubernetes Orchestration Security*:

Type	Description
Insider threats	• User access • Administrators • Infrastructure or Cloud Provider
Supply chain	• Infrastructure or HOST • Container runtime • Application layer and third-party dependencies
Malicious threat actors	• Control plane (apiserver, etcd, scheduler, controller manager) • Worker nodes (kubelet, kube-proxy) • Containers

***Table 7.2**: Kubernetes Threats Reference Matrix*

Note: Some of the aspects of *Table 7.2, Kubernetes Threats Reference Matrix*, have been discussed previously in this book; readers would appreciate that due to the extensiveness of the topics, this cannot be covered in a single section. The next chapter will provide a summary of the various topics with a quick reference to the sections of the book.

https://t.me/learningnets

Structure

In this chapter, we will discuss the following topics:

- Architecture
- Control plane hardening
- Worker node hardening
- Securing network communication
- Securing container runtime interface
- POD security
- POD escaping
- Hardening tools
- Updating life cycle

Objectives

This chapter aims to familiarize you with the security best practices for securing the Kubernetes platform, including the main components of its complex architecture. This hardening procedure aims to provide insights useful in real-world scenarios, where security best practices can be leveraged to achieve a better security posture.

Architecture

The main aim of Kubernetes is to make the complexity of managing containers disappearing by providing a simple alternative mechanism based on an internal API system. Kubernetes is a **client-server** architecture system with all the limitations and security concerns of a such an old model. In Kubernetes, the client-server model is applied to the platform itself, where the communication between the client and the server is serving the purpose of "running and managing" the underlying infrastructure rather than the containerized application, also known as **workloads**. In this scenario, the key components for the Kubernetes client-server model are the

https://t.me/learningnets

worker nodes (or **Data Plane**) acting as the client and the **control plane** acting as the server, as shown in *Figure 7.1, Kubernetes Architecture*.

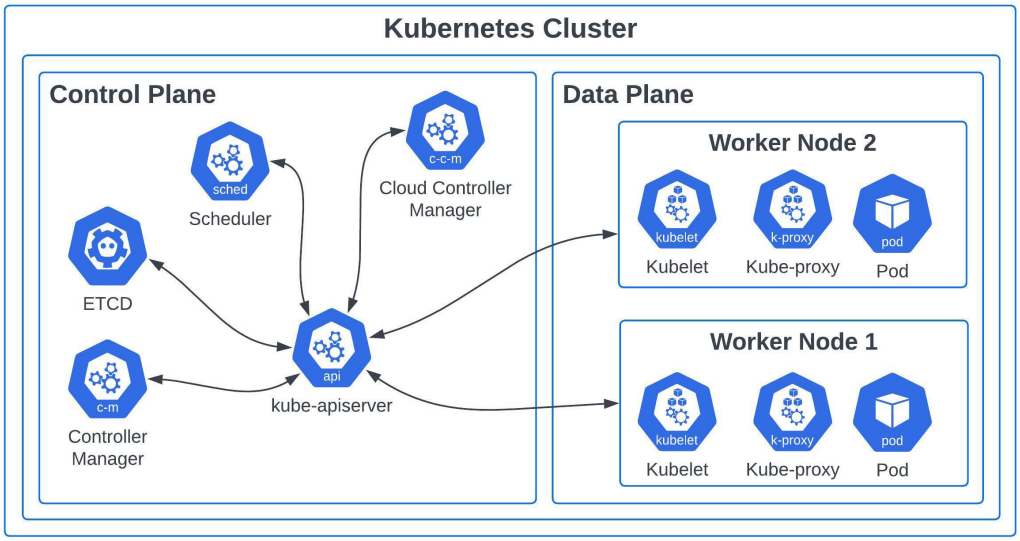

***Figure 7.1**: Kubernetes Architecture*

The client-server model was initially implemented to enable flexibility in the distributed application infrastructure model and decentralize the compute resources needed, with the client requesting a specific function and the server serving that computational request. The **Control Plane** and **Data Plane** should be considered as two separate sets of systems, grouped together by their logic applicationpurpose within the cluster. Any of these two components has a set of subcomponents deployed to address the multi-layered complexity of the Kubernetes container orchestrator. The **Control Plane** includes the components described in *Table 7.3, Control Plane Components*:

Type	Description
kube-apiserver	API server for the Kubernetes control plane
etcd	Cluster data key value store
kube-scheduler	Schedules and allocates the Pods
kube-controller-manager	Controls cluster related processes (for example, node controller)
cloud-controller-manager	Manages the interaction with the Cloud infrastructure, which will be discussed in depth in *Chapter 10, Kubernetes Cloud Security*

***Table 7.3**: Control Plane Components*

https://t.me/learningnets

The Data Plane is a set of one or more **worker nodes, each worker node** includes the components described in *Table 7.4, Worker Node Components*:

Type	Value
kubelet	Ensures containers are running on the Pod in the desired state
kube-proxy	Provides proxy capabilities to the worker node and ensure communication
Pod	The Kubernetes workloads deployable unit

***Table 7.4**: Worker Node Components*

Most recent Kubernetes installations set up keys, certificates and **Transport Security Layer** (**TLS**) encryption mechanism on either the kube-apiserver or the etcd data storage system in compliance with the **Center for Internet Security** (**CIS**) **Kubernetes Benchmark**, but a comprehensive hardening procedure would verify that anyway.

Control plane hardening

Many Linux distributions have forked and created their own Kubernetes version expanding the quantity of security best-practices needed. Without listing the much more popular Kubernetes cloud versions, interesting alternatives to the traditional Kubernetes cluster are **Canonical Kubernetes**, **Platform9**, and **CodeZero**. We will discuss the security aspects of the most popular Kubernetes cloud systems in *Chapter 10, Kubernetes Cloud Security*. There are many ways and many tools to install Kubernetes, but in essence, we can distinguish two main deployment approaches:

- For a **learning** environment, the recommended way is to install **minikube** with at least two nodes. Minikube is a lightweight Kubernetes single machine deployment system. It is known for drastically reducing the complexity of a complete Kubernetes deployment, thanks to its reduced footprint; nevertheless, it maintains all the characteristics of a full installation.
- For an on-premises or cloud **production** environment, there are three deployment methods: **kops**, **kubeadm** or **Kubespray**. The differences between the three methodologies and highlights of their use case scenarios are described in *Table 7.5, Kubernetes Production Deployment Methods*:

Type	Description
kops	Command-line tool similar to **kubectl** but aiming to deal with infrastructure creation and cluster provisioning that can interact with cloud providers systems and supports upgrades and add-ons deployments

https://t.me/learningnets

Type	Description
kubeadm	Aims to create a minimum cluster setup to pass the **Kubernetes Conformance Test** designed for vendors willing to obtain a certification released by the CNCF for their Kubernetes version; does not provide any infrastructure deployment, so control plane and worker nodes hosts must already be in place and configured with the desired virtualization driver and network connectivity; works with tools like Terraform or Ansible
Kubespray	Deployment automation tool based on a mix of **Ansible** playbooks, inventories and infrastructure provisioning tools like **Terraform**, Kubespray helps install Kubernetes on most cloud providers and bare metal systems

***Table 7.5**: Kubernetes Production Deployment Methods*

For the purpose of this chapter, **minikube** is the preferred approach because it enables the vast majority of readers to test their own environment. With **minikube** there is no need to have demanding hardware available in lab or to consider allocating budget for cloud resources. At the same time readers have the confidence that the suggested security measures are not going to negatively impact any production environment. The **minikube** security best practices begin from the installation of the Docker **rootless** system: the rootless mode is essentially the capability of the Docker daemon to run as a non-root service, creating a more secure environment. As reference, the Linux distribution used as host is **Ubuntu Server 22.04.2 LTS.** To enable the rootless mode, the Linux distribution used should comply with the requisites described in *Table 7.6, Rootless Mode Requisites*:

Type	Value
Login	Variable must be set to allow systemd and cgroups to work properly
User	**subuid** and **subgid** must be properly configured
Cgroup	Version 2 is supported
Delegation	Enable CPU and I/O to be available to non-root processes

***Table 7.6**: Rootless Mode Requisites*

The **`$XDG_RUNTIME_DIR`** is not set when one logs in to the machine as a root user, but it is set in case of a non-root user login via console or SSH. To verify that the login variable is in place, please run the following command:

```
1. $ echo $XDG_RUNTIME_DIR
2. /run/user/1000
```

https://t.me/learningnets

Containers running in rootless mode use user namespace to emulate privileges needed to create containers, as explained in *Chapter 3, Container Stack Security*, in the *Least Privilege* section. To verify that the user has at least 65,536 subordinates for UID and GID, it is possible to inspect **`/etc/subuid`** and **`/etc/subgid`**, and then install **`newuidmap`** and **`newgidmap`** contained in the **`uidmap`** package:

```
$ cat /etc/subuid
luigi:100000:65536
$ cat /etc/subgid
luigi:100000:65536
$ sudo apt-get install -y uidmap
```

Then, verify that Cgroup v2 is installed by running the following:

```
$ ls -l /sys/fs/cgroup/cgroup.controllers
-r--r--r-- 1 root root 0 Mar  2 12:58 /sys/fs/cgroup/cgroup.controllers
```

If the **`cgroup.controllers`** file is not present in the **`/sys/fs/cgroup`** folder, Linux is running Cgroup v1, but this should not be a problem with any modern Linux distribution. Lastly, in order to enable non-root users to access low-level machine resources like CPU, CPUSET and Disk I/O, a delegation process must be established:

```
$ sudo mkdir -p /etc/systemd/system/user@.service.d
$ touch /etc/systemd/system/user@.service.d/delegate.conf
# Add the following lines to the delegate.conf file
[Service]
Delegate=cpu cpuset io memory pids
$ sudo systemctl daemon-reload
```

After amending the **`systemd`** configuration, rebooting is recommended. The host is now ready for the Docker rootless mode installation:

```
$ curl -o rootless-install.sh -fsSL https://get.docker.com/rootless
$ sh rootless-install.sh
# Installing stable version 23.0.4
# Executing docker rootless install script, commit: 07206cb
...
# TLDR OMITTED
Installed docker.service successfully.
```

https://t.me/learningnets

```
$ export PATH=$HOME/bin:$PATH
$ docker context use rootless
rootless
Current context is now "rootless"
```

With Docker rootless mode installed, the next step is to install **`minikube`** and start the local cluster:

```
$ curl -LO https://storage.googleapis.com/minikube/releases/latest/
minikube_latest_amd64.deb
$ sudo dpkg -i minikube_latest_amd64.deb
$ minikube start --driver=docker --container-
runtime=containerd --nodes 2 -p my-cluster
😄 [my-cluster] minikube v1.30.1 on Ubuntu 22.04
✨ Using the docker driver based on user configuration
📌 Using rootless Docker driver
👍 Starting control plane node my-cluster in cluster my-cluster
🚜 Pulling base image ...
🔥 Creating docker container (CPUs=2, Memory=2200MB) ...
📦 Preparing Kubernetes v1.26.3 on containerd 1.6.20 ...
    ▪ Generating certificates and keys ...
    ▪ Booting up control plane ...
    ▪ Configuring RBAC rules ...
🔗 Configuring CNI (Container Networking Interface) ...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🔎 Verifying Kubernetes components...
🌟 Enabled addons: default-storageclass, storage-provisioner

👍 Starting worker node my-cluster-m02 in cluster my-cluster
🚜 Pulling base image ...
🔥 Creating docker container (CPUs=2, Memory=2200MB) ...
🌐 Found network options:
    ▪ NO_PROXY=192.168.49.2
📦 Preparing Kubernetes v1.26.3 on containerd 1.6.20 ...
```

https://t.me/learningnets

```
▪ env NO_PROXY=192.168.49.2
🔎 Verifying Kubernetes components...
🏄 Done! kubectl is now configured to use "my-cluster" cluster and "default" namespace by default
```

From *Figure 7.1, Kubernetes Architecture*, the centrality of the **kube-apiserver** component stands out; Kubernetes is indeed an API-based system, so it is the security around it. To ensure that basic security controls are in place, let's verify that the **kube-apiserver** is not exposed gathering the cluster address with the following command:

```
$ kubectl get all
NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   20m
```

The preceding command's output confirms three things: firstly, the cluster is running on a Class A internal IP (Internet Protocol) address range; secondly, there is no external IP assigned to the cluster; and thirdly, the cluster operates on port 443, meaning that the system is listening using HTTPS secure protocol. To verify that the cluster nodes have been successfully created, please execute the following command:

```
$ kubectl get nodes -o wide
NAME           STATUS   ROLES           AGE    VERSION   INTERNAL-IP    EXTERNAL-IP
my-cluster     Ready    control-plane   5m33s  v1.26.3   192.168.49.2   <none>
my-cluster-m02 Ready    <none>          4m48s  v1.26.3   192.168.49.3   <none>
```

Therefore, the nodes created by minikube are the hosts running Kubernetes infrastructure, one for the control-plane and another one for the worker node.

In this scenario, curling the cluster with the following command should time out:

```
$ curl -k https://10.96.0.1:443/api
curl: (28) Failed to connect to 10.96.0.1 port 443 after 75457 ms: Couldn't connect to server
```

https://t.me/learningnets

The "**failed to connect**" error is expected, but there are scenarios where the **kube-apiserver** is able to respond to a request, even without necessarily granting access, for example, by opening a shell on the worker node:

```
$ minikube ssh --native-ssh=false -p my-cluster -n my-cluster-m02
Last login: Sun Apr 23 23:56:27 2023 from 192.168.49.1
```

A return command like the one shown in the following code is expected because the API is listening but the anonymous user cannot interact with it:

```
$ curl -k https://192.168.49.2:8443
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {},
  "status": "Failure",
      "message": "forbidden: User \"system:anonymous\" cannot get path \"/\"",
  "reason": "Forbidden",
  "details": {},
  "code": 403
}
```

We will look at this topic in detail in *Chapter 8, Kubernetes Orchestration Security*, in the *API Bypass Risks* section, as the focus of this chapter is on hardening techniques. Run the following command to verify that the **kube-apiserver** has TLS encryption enabled:

```
$ kubectl get pods -n kube-system | grep api
kube-apiserver-my-cluster            1/1     Running   1 (11h ago)   23h
$ kubectl describe pod/kube-apiserver-my-cluster -n kube-system | grep tls
      --tls-cert-file=/var/lib/minikube/certs/apiserver.crt
      --tls-private-key-file=/var/lib/minikube/certs/apiserver.key
```

As stated, these parameters are usually auto populated during the installation, but the recommendation would be to adopt a custom certificate to be exchanged by the control plane components, given that a key management system is in place

https://t.me/learningnets

or a simple mechanism like **mkcert** has been implemented. Readers who need to adopt trusted certificates in development environments may find **`mkcert`** interesting and useful to run local machines with trusted self-signed certificates on **https://localhost:<anyport>**. You can visit **https://mkcert.org** for more information. The following code provides an example of how **`nginx`** with a custom TLS certificate can be implemented as ingress add-on:

```
$ kubectl -n kube-system create secret tls mysecret --key key.
pem --cert cert.pem
secret/mysecret created
$ minikube addons configure ingress -p my-cluster
-- Enter custom cert (format is "namespace/secret"): secret/mysecret
✅ ingress was successfully configured
$ minikube addons enable ingress -p my-cluster
💡 ingress is an addon maintained by Kubernetes. For any concerns con
tact minikube on GitHub.
You can view the list of minikube maintainers at: https://github.
com/kubernetes/minikube/blob/master/OWNERS
    ▪ Using image registry.k8s.io/ingress-nginx/kube-webhook-
certgen:v20230312-helm-chart-4.5.2-28-g66a760794
    ▪ Using image registry.k8s.io/ingress-nginx/kube-webhook-
certgen:v20230312-helm-chart-4.5.2-28-g66a760794
    ▪ Using image registry.k8s.io/ingress-nginx/controller:v1.7.0
🔎  Verifying ingress addon...
🌟  The 'ingress' addon is enabled
$ kubectl -n ingress-nginx get deployment ingress-nginx-
controller -o yaml | grep mysecret
- --default-ssl-certificate=secret/mysecret
```

A parameter that is often overlooked, even by the **CIS Kubernetes Benchmark**, but is mentioned in the Kubernetes **STIG Viewer** hardening guide is the TLS minimum version. The TLS protocol, like any other security mechanism, has a life cycle, and different versions have been released over the years. Today, TLS version 1.0 and version 1.1 are considered non secure, and they have both been deprecated in 2021. The version widely adopted currently is TLS version 1.2, but it is a protocol that was initially released in 2008, so it already has more than a decade of usage, while the next generation of the TLS protocol, TLS version 1.3, was released in 2018. Kubernetes supports all the four TLS versions mentioned earlier and therefore, it

https://t.me/learningnets

is highly recommended to set the parameter to TLS version 1.3. To achieve this, log in to the master node, make a backup copy of the API manifest file and add the required string, like in the following example:

```
$ minikube ssh --native-ssh=false -p my-cluster -n my-cluster
$ sudo cp /etc/kubernetes/manifests/kube-apiserver.yaml .
$ sudo vi /etc/kubernetes/manifests/kube-apiserver.yaml
...
    - --service-cluster-ip-range=10.96.0.0/12
    - --tls-cert-file=/var/lib/minikube/certs/apiserver.crt
    - --tls-private-key-file=/var/lib/minikube/certs/apiserver.key
    - --tls-min-version=VersionTLS13
    image: registry.k8s.io/kube-apiserver:v1.26.3
    imagePullPolicy: IfNotPresent
...
```

The Kubernetes data storage system is **`ectd`**, the same principle as that of any other data storage system should be applied to guarantee a better security posture: both **encryption at rest** and **encryption in transit** should be applied and enabled. The **`ectd`** data system is the central storage system to Kubernetes; if an attacker compromising the API server would gain access to the ectd data storage system, then the attacker has de-facto access to the entire cluster, being able to manipulate essentially any single function or component of the system, including accessing secrets, start pods, manage users and credentials. It is possible to check whether the etcd data storage system has been configured with TLS encryption in respect to the **encryption in transit** security control by checking on the **`etcd`** pod with the following command:

```
$ kubectl describe pod/{etcd-pod-name} -n kube-system
$ sudo cat /etc/kubernetes/manifests/etcd.yaml
    - etcd
    - --advertise-client-urls=https://192.168.49.2:2379
    - --cert-file=/var/lib/minikube/certs/etcd/server.crt
    - --client-cert-auth=true
    ...
    - --key-file=/var/lib/minikube/certs/etcd/server.key
```

If the two parameters **`--cert-file`** and **`--key-file`** are present in the etcd manifest file, the encryption in transit has been properly configured. With the encryption in transit configured properly, the next step is to verify that the **`etcd`** data storage

https://t.me/learningnets

system has **encryption at rest** enabled. To verify that the **etcd** system has encryption at rest enabled, run the following command querying the **kube-apiserver** and look for **the --encryption-provider-config** parameter:

```
$ kubectl describe pod/{kube-apiserver-pod-name} -n kube-system
```

If that parameter is not returned from the **kube-apiserver**, the **etcd** system does not have encryption at rest enabled, and it must be configured. The configuration file is as follows:

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
      - secrets
    providers:
      - identity: {}
      - aesgcm:
          keys:
            - name: key1
              secret: my-secret-key-1
            - name: key2
              secret: my-secret-key-2
      - aescbc:
          keys:
            - name: key1
              secret: my-secret-key-1
            - name: key2
              secret: my-secret-key-2
      - secretbox:
          keys:
            - name: key1
              secret: my-strong-secret-key-1
```

https://t.me/learningnets

The preceding encryption providers are detailed in *Table 7.7, etcd encryption providers*:

Type	Encryption	Strength
identity	None	None
secretbox	XSalsa20 / Poly1305	Strong
aesgcm	AES-GCM	must be rotated every 200k writes
aescbc	AES-CBC with PKCS#7	Weak
kms v1	DEK using AES-CBC with PKCS#7	Strongest but slow
kms v2	DEK using AES-GCM	Strongest and fast

***Table 7.7**: etcd encryption providers*

KMS is the fastest and strongest encryption provider, so it is also the recommended choice. It uses a **Data Encryption Key** (**DEK**) system with a key generated for each encryption and a **Key Encryption Key** (**KEK**) mechanism to facilitate key rotation. Cloud provider key managed systems like **AWS KMS** and **Azure Key Vault** adopt one or more of the encryption providers described earlier, and they can ingest keys into Kubernetes clusters through gRPC protocol. The following small lab will provide an example of how to enable encryption at rest for the etcd component:

```
1. $ minikube ssh --native-ssh=false -p my-cluster -n my-cluster
2. Last login: Tue Apr 25 22:56:09 2023 from 192.168.49.1
3. docker@my-cluster:$ head -c 32 /dev/urandom | base64
4. JuF9H4fz5CvAbu/oD/O+OUl9ZnDthRTHRQ16OSKLeP0=
5. docker@my-cluster:$ sudo mkdir -p /etc/kubernetes/enc
6. docker@my-cluster:$ cat <<EOF | sudo tee /etc/kubernetes/enc/enc.yaml
7. apiVersion: apiserver.config.k8s.io/v1
8. kind: EncryptionConfiguration
9. resources:
10.   - resources:
11.       - secrets
12.       - configmaps
13.       - pandas.awesome.bears.example
14.     providers:
15.       - aescbc:
16.           keys:
```

https://t.me/learningnets

```
17.             - name: key1
18.               secret: JuF9H4fz5CvAbu/oD/O+OUl9ZnDthRTHRQ16OSKLeP0=
19.       - identity: {}
20. EOF
21. docker@my-cluster:$ exit
22. logout
```

Log in to the control plane, creating a random base64 value used as a secret at line 18 for the **`enc.yaml`** file, set the **`--encryption-provider-config`** parameter on the **`kube-apiserver`** to point to the Encryption Configuration file location by editing the **`kube-apiserver`** manifest file at line 6, and add the reference to encryption provider in the **`volumeMounts`** and **`volumes`** section, as shown in the following example:

```
1.      - --tls-min-version=VersionTLS13
2.      - --encryption-provider-config=/etc/kubernetes/enc/enc.yaml
3.      image: registry.k8s.io/kube-apiserver:v1.26.3
4.      imagePullPolicy: IfNotPresent
5.  ...
6.      volumeMounts:
7.  ...
8.      - mountPath: /etc/kubernetes/enc
9.        name: enc
10.       readOnly: true
11. ...
12.   volumes:
13. ...
14.   - hostPath:
15.       path: /etc/kubernetes/enc
16.       type: DirectoryOrCreate
17.     name: enc
```

To verify that the data is encrypted, create a new secret called **`test-secret`** in the default namespace:

https://t.me/learningnets

```
$ kubectl create secret generic test-secret -n default \
--from-literal=mytest-secret=mytest-data
secret/test-secret created
```

The verification process needs the **etcdctl** command, which is usually installed on the **etcd** container. To login onto control plane containers in a Kubernetes **minikube** deployment, it is possible to **list** and **ssh** on the minikube control plane nodes by executing the following commands:

```
$ minikube node list -p my-cluster
my-cluster 192.168.49.2
my-cluster-m02 192.168.49.3
$ minikube ssh --native-ssh=false -p my-cluster -n my-cluster
```

Several tools are available on the control plane node, among which **crictl** can list and manage either the containers or the pods.

```
$ sudo crictl ps
CONTAINER      STATE     NAME                      POD ID
f6f5eb3476c0e  Running   storage-provisioner       1d1a2ec2067d7
1f8124d86b411  Running   kindnet-cni               325ce54832c28
79c45d5df013e  Running   coredns                   83d3bb1604446
ba894af13ddfc  Running   kube-proxy                8d85fa2fdc44e
ec7e1e508328b  Running   kube-controller-manager   d0d68bdcd08ab
b1b5832cfc814  Running   kube-apiserver            aec8a1424cc81
ad0a7f5df8f4d  Running   etcd                      d6988a3b45379
c60e52371559b  Running   kube-scheduler            40148808c32ea
```

A successful call to the **etcd** container must be authorized, so the parameters defined in *Table 7.8, etcd encryption parameters*, must be provided to the **etcdctl** command according to the Kubernetes cluster type in use:

Var	Minikube	Kubernetes
--cacert	/var/lib/minikube/certs/etcd/ca.crt	/etc/kubernetes/pki/etcd/ca.crt
--cert	/var/lib/minikube/certs/etcd/server.crt	/etc/kubernetes/pki/etcd/server.crt
--key	/var/lib/minikube/certs/etcd/server.key	/etc/kubernetes/pki/etcd/server.key

***Table 7.8**: etcd encryption parameters*

https://t.me/learningnets

From the control plane node, the following command will return a dump of the **test-secret** file prefixed with **k8s:enc:aescbc:v1** confirming data encryption at rest:

```
1. $ sudo crictl exec -i ad0a7f5df8f4d etcdctl \
2. --cacert=/var/lib/minikube/certs/etcd/ca.crt \
3. --cert=/var/lib/minikube/certs/etcd/server.crt \
4. --key=/var/lib/minikube/certs/etcd/server.key \
5. get /registry/secrets/default/test-secret
6. k8s:enc:aescbc:v1:key1:xc #NOT READABLE CHARACTERS
```

The **kube-controller-manager** and the **kube-scheduler** are not often mentioned because from the logic perspective, they need to interact with the **kube-apiserver**; so, they operate from a layer down within the control plane with no direct communication with the worker nodes. Nevertheless, their functionalities within the cluster are essential to Kubernetes, so they must be considered from the security standpoint when the goal is to achieve a better security posture. The **CVE-2020-8555** is a security vulnerability affecting the **kube-controller-manager** up to version 1.18.0, allowing **Server-Side Request Forgery** (**SSRF**) to leak 500 bytes of arbitrary data within the control plane's host network by certain users. The **kube-controller-manager** manages the overall Kubernetes cluster and controls its functions, ensuring that the expected number of pods are running as expected. To verify the current **kube-controller-manager** configuration, run the following command:

```
1. $ kubectl describe pod/{kube-controller-manager-pod-name} -n kube-
   system
```

The expected security parameters are detailed in *Table 7.9, Kube Controller Manager parameters*:

Type	Values on minikube
--use-service-account-credentials	True
--service-account-private-key-file	/var/lib/minikube/certs/sa.key
--bind-address	127.0.0.1
--root-ca-file	/var/lib/minikube/certs/ca.crt
RotateKubeletServerCertificate	true (applied via the --**feature-gates** option)

***Table 7.9**: Kube Controller Manager parameters*

All the previous parameters are usually configured by default in any Kubernetes installation, except for the **feature gates** option. To apply the option

https://t.me/learningnets

RotateKubeletServerCertificate to the kube controller manager, it is possible to modify the manifest file so that it looks as follows:

```
1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   creationTimestamp: null
5.   labels:
6.     component: kube-controller-manager
7.     tier: control-plane
8.   name: kube-controller-manager
9.   namespace: kube-system
10. featureGates:
11.   RotateKubeletServerCertificate: true
12. spec:
13.   containers:
14.   - command:
15.     - kube-controller-manager
16.     - --bind-address=127.0.0.1
17.     - --client-ca-file=/var/lib/minikube/certs/ca.crt
18.     - --root-ca-file=/var/lib/minikube/certs/ca.crt
19.       - --service-account-private-key-file=/var/lib/minikube/certs/
    sa.key
20.     - --use-service-account-credentials=true
21.     image: registry.k8s.io/kube-controller-manager:v1.26.3
22.     imagePullPolicy: IfNotPresent
23.     livenessProbe:
24. ...
```

The **feature gates** option can also be applied as a command-line parameter inside the manifest file, as it will be illustrated in the **kube-scheduler** example. The **kube-scheduler** selects the best modality to run newly created, scheduled and unscheduled Pods. The **kube-scheduler** filters and selects the nodes according to any Pod's specific needs, mainly around resources allocations such as CPU and RAM, but also according to their "labelling" or destination in more complex environments.

https://t.me/learningnets

To verify the current **kube-scheduler** configuration, run the following command:

```
$ kubectl describe pod/{kube-scheduler-pod-name} -n kube-system
```

The expected security parameters are detailed in *Table 7.10, Scheduler parameters*:

Type	Value
--profiling	false
--bind-address	127.0.0.1
AppArmor	true (applied via the --**feature-gates** option)

***Table 7.10**: Scheduler parameters*

The **--profiling** parameter is used to identify performance issues within the cluster, but due to its purpose, it generates a significant amount of information that can be potentially used as attack vector, especially in combination with the **--bind-address** option. The **AppArmor** feature, please refer to *Chapter 2, Hardware and HOST OS Security, section Host Hardening, AppArmor*, is enabled via the **--feature-gates** parameter. It accepts values in the following comma-separated format: **<"key=True|False","key=True|False">**. Therefore, to enable the **AppArmor** feature for the **kube-scheduler**, the following command-line parameter should be added to the **kube-scheduler** manifest file:

```
Containers:
  kube-scheduler:
    Container ID:  docker://2f0e7db06157c2d9c98bffe23ce-
942681be808e9084fc994e9f796b1d7d3a211
    Image:         k8s.gcr.io/kube-scheduler:v1.25.2
    Image ID:      docker://sha256:873dc124ec692aa7dae4f3b1b-
41898c8bda1ef989c08823dbf183286155d0eed
    Port:          <none>
    Host Port:     <none>
    Command:
      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
      --feature-gates=AppArmor=true
```

https://t.me/learningnets

The Kubernetes **Features Gates** is not a well-known set of components and features that can come in handy in increasing the level of customization of the Kubernetes cluster. There are several options that can be leveraged to increase the security posture of the cluster, and readers are encouraged to use those features for their specific applications. Those features have life cycles, and many of those are improved or deprecated in accordance with the overall Kubernetes system development cycle. All the components of both the control plane and the worker node accept the Feature Gates set with specific parameters designed to work with the logic mechanism of that specific component.

Worker node hardening

As shown in *Figure 7.1, Kubernetes Architecture*, there are two main components running on any **Worker Node**: the **`kubelet`** and the **`kube-proxy`**. In terms of security posture, the worker node is exposed much more than any of the elements of the control plane because the control plane is considered as an internal mechanism, while the worker node runs Pods that are potentially exposed to external traffic and users, as illustrated in *Figure 7.2, Worker Node*:

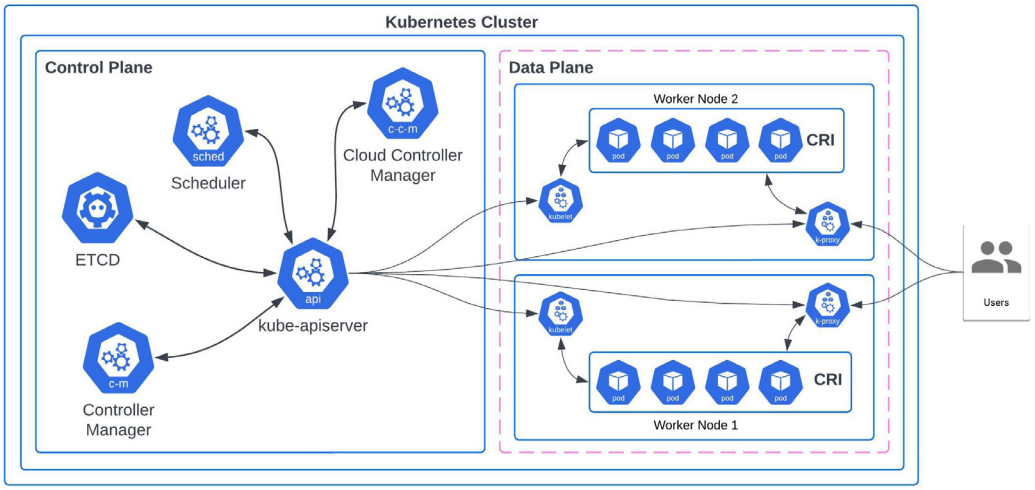

***Figure 7.2**: Worker Node*

The **`kubelet`** is essentially an agent running on the worker node in charge of managing the containers; it receives and executes requests of creating, destroying and configuration changing that need to be applied to the containers. The most common security parameters are detailed in *Table 7.11, Kubelet Security*:

https://t.me/learningnets

Type	Value
--anonymous-auth	False
--rotate-certificates	True
--tls-private-key-file	/path-to-private-key-file/node.key
--tls-cert-file	/path-to-cert-file/node.crt
--tls-min-version	VersionTLS13
RotateKubeletServerCertificate	true (applied via the --**feature-gates** option)

***Table 7.11**: Kubelet Security*

As per the kubelet system, **`kube-proxy`** runs on any worker node within the cluster, with the purpose of handling the network rules on the nodes on which it is running. The kube-proxy ensures that the worker nodes can establish and maintain network connections with internal and external resources as required and permitted. Due to the sensible nature of the kube-proxy's functionalities, being in essence the network handler, it is essential that the whole kube-proxy configuration file is "secure" with the highest set of restrictions possible. The recommended set of permissions for the **`kube-proxy`** config file is **`root:root`** (root or superuser for both owner and group) with **600** file permissions, this is equal to the following unix declaration:

```
1. -rw-------
```

This means that only the user root can read and write to the file, while preventing anybody else from doing so. This should be the default file permission value for all the configuration files. Later in this chapter, in the **POD Security** section, we will enhance the security best practices around POD security mechanism and look at how they interface with the **Worker Node** and its components, and we will discuss kube-proxy in the **Securing Network Communication** section.

Securing network communication

The network system and communications are managed into Kubernetes cluster by the **Kubernetes network proxy**, also known as **`kube-proxy`**. Each node runs a separate **`kube-proxy`** daemon system that reflects the network services as defined via the **`kube-apiserver.`** Other than managing internal traffic communication between the worker node and the control plane components, the kube-proxy also forwards traffic connection via services to the appropriate containers; please review *Figure 7.2, Worker Node*. **`Kube-proxy`** runs in three different modes:

- Iptables (default)
- IPVS (IP Virtual Server)
- User space (legacy, not recommended)

https://t.me/learningnets

The difference between the **iptables** and **IPVS** modes is from the load standpoint, because iptables reads firewalling rules sequentially, it can impact the performance of the cluster if the workload goes beyond 1000 services running in parallel. The Kubernetes cluster has a **hub and spoke** network communication model. All the API requests from worker nodes, including **`kubelet`**, **`kube-proxy`**, **PODs** and containers, are terminated at the **`kube-apiserver`**. No control plane component is designed to expose remote services except the **`kube-apiserver`**, as illustrated in *Figure 7.3*, *kube-proxy*. This guarantees the internal integrity of the control plane.

The **`kube-apiserver`** is configured to listen on a secure HTTPS port for remote connections adopting one or more mechanism of client authentication. As explained in the previous section, it is recommended to enable TLS encryption, providing the **`kube-apiserver`** with a root certificate bundle to verify the **`kubelet`** certificate. This option has been suggested because by default, **`kube-apiserver`** does not verify the **`kubelet`** certificate, exposing the connection to man-in-the-middle attacks. The connection from **`the kube-apiserver`** to services or PODS, which happens through the **`kube-proxy`** component, default to HTTP when neither encryption nor authentication are set, compromising the integrity of the network communication.

Kubernetes supports the **SSH tunnelling** mechanism to establish a secure communication path, where the **`kube-apiserver`** connects on the port 22, initiating an SSH Tunnel to each worker node. The SSH tunnel guarantees that the traffic between the nodes and the Kubernetes API Server is not exposed. Although SSH tunnelling is a secure mechanism, it creates overhead on the communication between the Kubernetes API Server and the nodes, which transforms into a bottleneck where the number of nodes reaches a certain threshold. In Kubernetes v1.18, a secure communication mechanism has been introduced to replace SSH tunnelling: **Konnectivity**. Currently SSH tunnelling is deprecated.

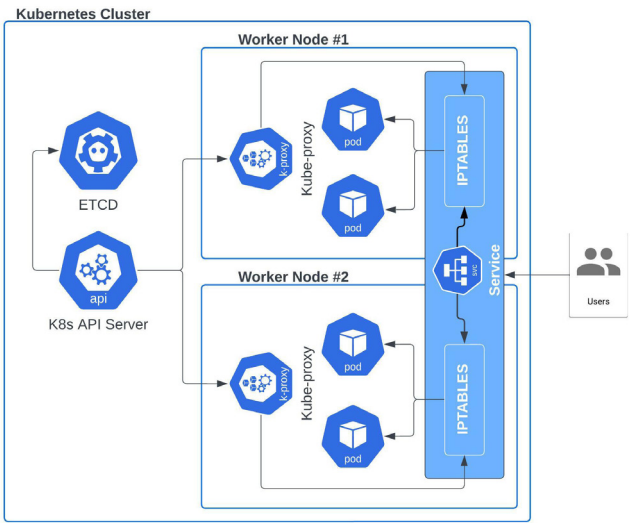

***Figure 7.3**: kube-proxy*

https://t.me/learningnets

Konnectivity provides TCP proxy level for the Kubernetes cluster to ensure secure control plane communication. The Konnectivity service uses a server-client model, with a **Konnectivity Server** to be deployed in the control plane network and a **Konnectivity Agent** to be deployed in the nodes network. Once enabled, the whole control plane to nodes traffic is redirected via Konnectivity. To deploy the Konnectivity system, a few steps need to be satisfied, including creating an egress selector configuration file, as shown in the following example code:

```
1. apiVersion: apiserver.k8s.io/v1beta1
2. kind: EgressSelectorConfiguration
3. egressSelections:
4. - name: cluster
5.   connection:
6.     proxyProtocol: GRPC
7.     transport:
8.       uds:
9.         udsName: /etc/kubernetes/konnectivity-server/konnectivity-
   server.socket
```

Then, implement the following steps:

- Enable Service Account Token Volume Projection (from Kubernetes v.120, it is enabled by default).
- Configure the kube-apiserver, **`--egress-selector-config-file`** parameter to point at the egress selector file.
- If the cluster uses **Unified Diagnostic Service** (**UDS**), add the volume configuration to the kube-apiserver.
- Retrieve a **X.509** certificate using the Kubernetes CA cluster certificate from **`/etc/Kubernetes/pki/ca.crt`** from one of the control plane systems.

The configuration files for the **Konnectivity Server** and the **Konnectivity Agent** are available on the Kubernetes docs website. Their implementation should be straightforward, and it is interesting from the security standpoint considering the system implementation via **Role Base Access Control** (**RBAC**).

Securing container runtime interface

The **Container Runtime** operates at the lowest level of any Kubernetes node; it is the software that starts and stops containers, among other functions. The most widely known and adopted container runtime is out of questions Docker, but Docker is not

https://t.me/learningnets

the only container runtime available on the market. The aim of this paragraph is to define a clear distinction between the **container runtime**, which was discussed along with the various types of container runtimes available within the Kubernetes ecosystem and the reasons why Kubernetes adopted them, in *Chapter 3 – Container Stack Security, Container Security*, and the **CRI** (Container Runtime Interface).

As the name suggests, **Container Runtime Interface** is an interface developed by the Kubernetes community in 2016 and introduced in Kubernetes v1.5 to provide greater flexibility in adopting a different container runtime if needed. The main reason behind this was to push the idea of supporting interchangeable container runtimes and trying to move the community to meet the **Open Container Initiative** (**OCI**) requirements.

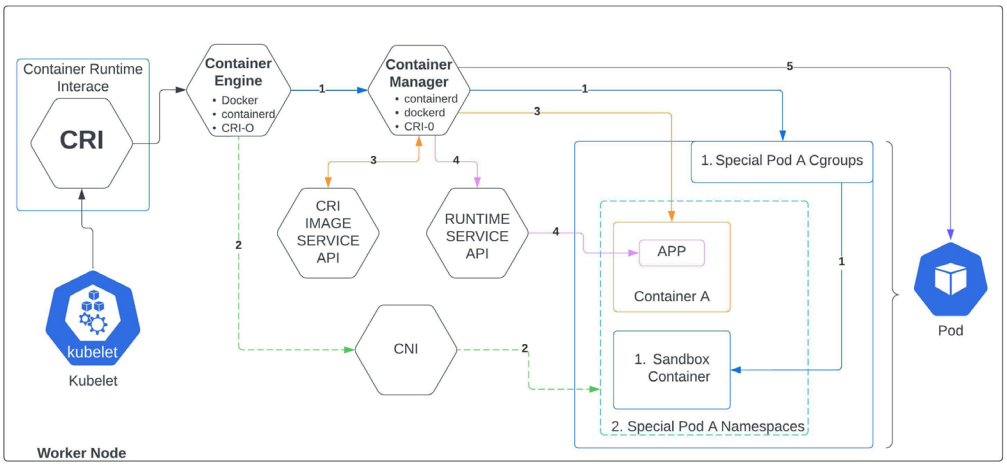

***Figure 7.4**: CRI*

The CRI (Container Runtime Interface) architecture is quite complex, and spinning up an application container is not a trivial process, as shown in *Figure 7.4, CRI.* A breakdown of the process illustrated in *Figure 7.4, CRI* is in the following list:

1. **Kubelet** starts the process, calling **`cri-containerd`** to create a POD; cri-containerd defers to containerd to create the sandbox container, a special isolated container used as the base system for the creation process, which sets the system bootstraps with kernel alignment and **Cgroups** configuration.
2. **`cri-containerd`** calls the **Container Network Interface** (**CNI**) to create the network **namespace** and configure all the related network activities.
3. **`cri-containerd`** reads the Kubelet requests and downloads the desired container image if not present on the node.

https://t.me/learningnets

4. **`cri-containerd`** calls the runtime service API system to start the container using the image at point 3 and deploy the app inside the container.
5. **`cri-containerd`** uses the information collected so far from the **sandbox** container and the initial deployment in the temporary **Container A** to build the final POD.

In essence, the role of **Container Runtime Interface** is to forward the **`kubelet`** requests to the **container runtime** of choice (either Docker, containerd or CRI-O); the container runtime will then process those requests to create the POD that will run the desired container and application within. During the containerization process, the container runtime interface handles the entire resources virtualization, including CPU units and RAM allocation, but more importantly, it handles the **`kernel syscall`** mechanisms, as illustrated in *Figure 7.5, CRI Logic*:

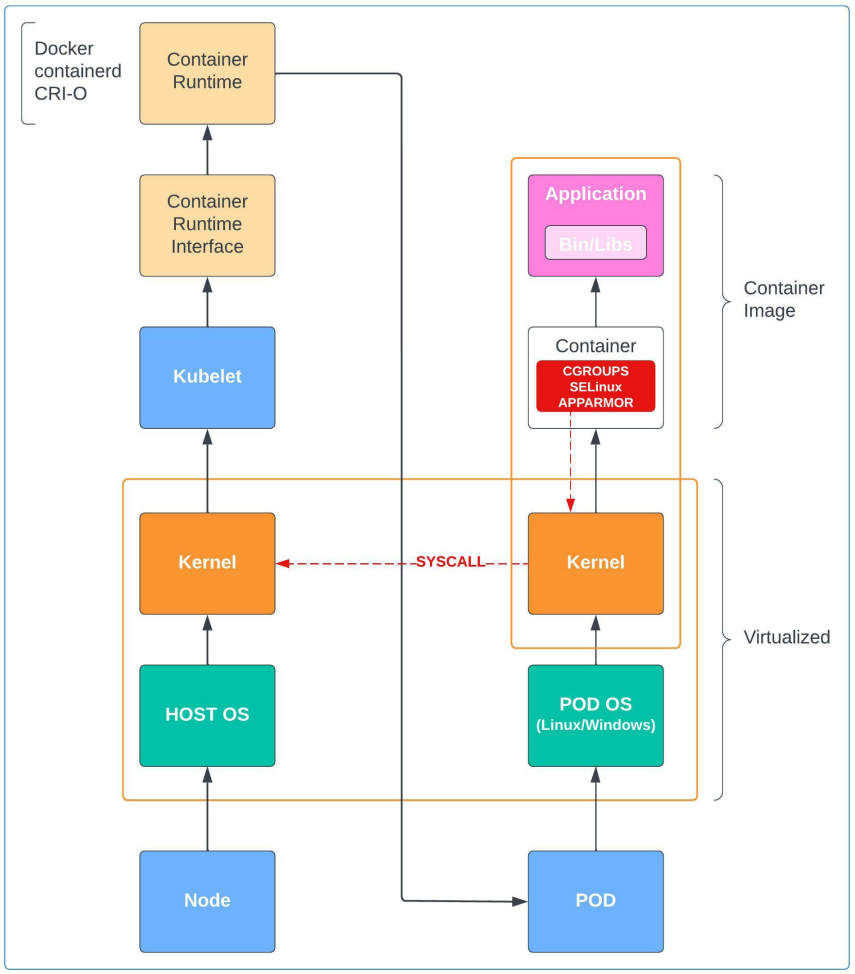

***Figure 7.5**: CRI Logic*

https://t.me/learningnets

The **CVE-2022-0811**, discovered at the beginning of the 2022, reveals a flaw in the **CRI-O** in the way in which the container runtime sets kernel options for the POD. This flaw allows users with permissions to deploy PODs using the **container escape** technique to execute arbitrary code as root users on the worker node. There are several threats that could appear in relation to containers deployment:

- Hidden malware in container images
- Exploitable container runtime due to security bugs
- Misuse of the resources allocated to the POD from the HOST OS like volumes, libraries or binaries
- Improper access control configurations or privilege escalation

The **CVE-2022-23648**, reveals a bug in the **`containerd`** container runtime affecting versions prior to 1.6.1 where containers launched with a particular image configuration could access files and folder on the host. The attack is also able to bypass policy-based enforcement systems or container security configurations. Being the **CRI** the point of contact between the worker node and the POD, once the containerization process is complete, the communication between the container and the host happens through the kernel for all the relevant system calls (for example, CPU processing and RAM allocation).

Therefore, the security around the CRI is an indirect process, based on the security functionality of the Linux kernel and the interpretation of those features applicable to the Kubernetes system. Among the kernel security features, as discussed in *Chapter 2, Hardening and Host OS Security, Host Hardening*, the Cgroups, Namespace, SELinux and Seccomp features are applied at the POD creation phase. The only kernel security feature applied directly to the worker node is AppArmor. **AppArmor** replaces the classic user and group permissions, characteristic of the Linux systems, with an application security model approach, aiming to confine application's execution into a defined execution limited set of resources.

Currently, all Kubernetes CRIs support AppArmor, so its application is encouraged to increase the node security posture. The recommended way to apply the AppArmor profile at the node level is through a **`DaemonSet`** that creates a POD to verify that the correct AppArmor profiles are loaded. The node assumes that AppArmor has been enabled into the cluster via the **`--feature-gates`** parameter described in *Table 7.10, Scheduler parameters*. The following code provides an AppArmor profile example:

```
1. . . .
2. profile k8s-apparmor-deny-write flags=(attach_disconnected) {
3.   # Allow all file read access to the / volume.
4.   file,
```

https://t.me/learningnets

```
5.    # Deny all file write access to the / volume.
6.    deny /** w,
7. }
8. . . .
```

Another way to implement AppArmor is at the worker node initialization time, using automation tools like Ansible or Salt. Kubernetes, on its own, does not offer any tool or methodology to shield the Container Runtime interface and the container runtime; the only available feature is the audit system, which collects information that the cluster does not consume. It also needs to be considered that POD resources are ephemerals, so their audit logs exist only during their running time. When the POD, and therefore the containers, are terminated, there will be no logging anymore. In this scenario, a debug tool like **Sloop** is very helpful because it can monitor past events of the Kubernetes cluster, recording histories of state changes of deployed resources.

POD security

A **POD** is a Kubernetes' deployable unit of compute resources. A POD is not a container, or it's better to say that a container is not a POD but a POD is a group of virtualized resources like CPU units, RAM, storage, and network deployed in conjunction with one or more application containers and their binaries and libraries. Kubernetes POD Security Standards defines three policies to cover the POD security landscape, as listed in *Table 7.12, POD Security Standards*:

Profile	Description
Privileged	Policy with no restrictions that allows privilege escalations
Baseline	Policy that applies a minimum set of restrictions preventing known privilege escalation and is the default POD configuration
Restricted	Maximum restricted policy as per POD hardening security practices

***Table 7.12**: POD Security Standards*

There is no security argument that can be leveraged to justify the **Privileged Policy**, except for the when it is purposefully unrestricted, and therefore reasonably applicable to test environments. The **Baseline Policy** is a balanced approach for the adoption of containerized workloads and simultaneously blocking known privileged escalation methodologies. In this regard, the set of security controls is listed in *Table 7.13, Baseline Controls:*

https://t.me/learningnets

Control	Policy
HostProcess	Windows PODs do not have privileged access to the Windows Node.
Host Namespaces	Namespace sharing with the host must be denied.
Privileged Containers	They overcome most security controls and must be disallowed.
Capabilities	Limit capabilities to only those allowed by the policy.
HostPath Volumes	Mapping to the host volume must be denied.
AppArmor	This would prevent disabling the default AppArmor profile.
SELinux	Only the default SELinux profile is allowed.
/proc Mount Type	Default /proc required. It reduces the attack surface.
Seccomp	Set to "Unconfined" is disallowed.
Sysctls	Disabled and isolated (no lateral movements).

***Table 7.13**: Baseline Controls*

The **Restricted Policy** is the highest level of security mechanism applicable to the POD. It aims to enforce POD hardening, but it also reduces the availability of the system. This rule is cumulative with the Baseline, which means it applies all the security controls of the Baseline Policy plus the security controls listed by the Restricted Policy, as per *Table 7.14, Restricted Controls*:

Control	Policy
Volume Types	Null values are not accepted. Must be one of the listed types.
Privilege Escalation	set-user-ID and set-group-ID file mode not allowed.
Running as Non-root	Containers are forced to run as non-roots users
Running as Non-root-user	runAsUser with 0 value is not allowed.
Seccomp	Set to "Unconfined" is disallowed. Profile must be explicit.
Capabilities	Only NET_BIND_SERVICE permitted.

***Table 7.14**: Restricted Controls*

The **security profiles** are applied at the control plan level and therefore, they enforce security mechanisms from the server standpoints. These are different from the **Security Contexts**, which are applied at the POD level instead. In essence, the Security Context is applying some of the security controls defined in the previous Baseline and Restricted policies, but at the POD level, that means that those security

https://t.me/learningnets

controls can be implemented during the deployment or rolling mechanism, or even better, considered as part of Software Development Life Cycle from the DevSecOps standpoint.

For instance, the security controls applicable through the Kubernetes POD Security Context mechanism are UID and GID (User ID and Group ID), SELinux (Security Enhanced Linux), Capabilities, AppArmor, Seccomp, running as privileged or unprivileged, and **`readOnlyRootFilesystem`**. The following a code example of applying security context to a POD configuration file, where security context is applied at row 6 for all the containers in this POD, and at row 20 where each container will disallow privilege escalation:

```
1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   name: security-context-example
5. spec:
6.   securityContext:
7.     runAsUser: 1000
8.     runAsGroup: 2000
9.     fsGroup: 1000
10.  volumes:
11.  - name: sec-ctx-vol
12.    emptyDir: {}
13.  containers:
14.  - name: sec-ctx- example
15.    image: my-image:1.0
16.    command: [ "sh", "-c", "sleep 1h" ]
17.    volumeMounts:
18.    - name: sec-ctx-vol
19.      mountPath: /data/example
20.    securityContext:
21.      allowPrivilegeEscalation: false
```

Linux **Capabilities** can grant higher privileges to a Linux process without granting privileges of a root user. With Security Context, it is possible to specify, within the

https://t.me/learningnets

POD configuration file, the capabilities the container needs to be granted, as per the following example:

```
1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   name: security-context-capability
5. spec:
6.   containers:
7.   - name: sec-ctx- capability
8.     image: my-image:1.0
9.     securityContext:
10.      capabilities:
11.        add: ["SYS_ADMIN", "SYS_TIME"]
```

Security Enhanced Linux (**SELinux**) allows Linux administrators better control over providing access by defining security policies for applications, processes, and files. Kubernetes implements SELinux at the POD layer via the **`securityContext`** argument that propagates the desired SELinux rules to each container the POD will manage. Among the three types of rules, the most intensive is the one related to file security. The section of the POD configuration file after the **`containers`** argument would look like the following code in this scenario:

```
1. ...
2. securityContext:
3.   seLinuxOptions:
4.     level: "s0:c123,c456"
```

The default mechanism before Kubernetes v1.25 was that the **Container Runtime (CR)** implements SELinux labelling recursively: all files on all the volumes mounted on the POD are target of the SELinux propagation. This methodology was very CPU-intensive and time-consuming. To speed up the SELinux implementation from v1.25, Kubernetes can apply the SELinux protection mechanism to a volume instantly with the mount option **`-o context=<label>`**.

Note: As illustrated in *Figure 7.5, CRI Logic*, the container's kernel is virtualized via the containerization mechanism allowed via the container runtime, so the SELinux security module must be enabled on the Worker Node Host OS to be loadable on the POD and the containers.

https://t.me/learningnets

To benefit from the SELinux instant volume implementation, the following conditions must be met:

- **`ReadWriteOncePod`** and **`SELinuxMountReadWriteOncePod`** must be enabled.
- POD must set **`seLinuxOptions`**.
- The volume must use either a **CSI** driver or legacy volume **iscsi** type.
    - The CSI driver must declare the mounting option **`-o context`** by setting **`spec.seLinuxMount:true`**.
- POD must have the **`accessModes:[“ReadWriteOncePod”]`** argument set on the **`PersistentVolumeClaim`** directive.

Seccomp is a Linux Kernel feature that sandboxes a system call process only to the basic executions processes like **`read()`**, **`write()`**, **`sigreturn()`**, and **`exit()`**, applying an execution separation between the user space and the system space. The Kubernetes' implementation of Seccomp at the POD layer occurs, including the **`seccompProfile`** in the **`securityContext`** section of the POD configuration file, as shown in the following code example:

```
1. ...
2. securityContext:
3.   seccompProfile:
4.     type: RuntimeDefault
```

The **`seccompProfile.type`** has three options: **`RuntimeDefault`**, **`Unconfined`** and **`Localhost`**, as explained for the POD Security Profiles. The **`Localhost`** type must be set only when a Seccomp **`localhostProfile`** argument is provided, such as in the following example code:

```
1. ...
2. securityContext:
3.   seccompProfile:
4.     type: Localhost
5.     localhostProfile: profiles-dir/my-seccomp-profile.json
```

The methodologies applied in this section found their security efficiency from the POD standpoint and require a higher degree of granularity in configuring the various security controls.

https://t.me/learningnets

POD escaping

In *Chapter 3, Container Stack Security*, we discussed the container escape attack vector several times. POD escaping has a similar footprint, and it is the attack technique that allows a POD to "escape" its own boundaries in terms of resources, network limitation or access to the underlying host system (the worker node). The **CVE-2022-0185** is one of the most interesting attack vectors recently discovered because it affects the **legacy_parse_param** function of the kernel. This function verifies the Linux kernel parameters length in term of bytes. An unprivileged user that can access the filesystem which does not support the **Filesystem Context API** function could generate a **heap-buffer overflow** (refer to *Chapter 4, Securing Container Images and Registries; Figure 4.6, Stack Buffer Overflow*) by sending more than 4095 bytes to the **legacy_parse_param** function and escalate their privileges.

The input is injected via the kernel **fsconfig** system call, by sending 4095 bytes or more to the **legacy_parse_param** function. The interesting thing is that to call fsconfig, a non-privileged user needs **CAP_SYS_ADMIN** privileges in the namespace in which the attack is executed. The **SYS_ADMIN** feature is defined via Capabilities, and that's the reason for the **CAP_** suffix; it qualifies itself as the process that runs tasks on behalf of the user with privileged access. Where the **CAP_SYS_ADMIN** capability is not available, the attacker can leverage **unshare**, another Linux kernel system call, which is very relevant to the Kubernetes cluster because it is used to create or clone a namespace.

In Kubernetes, namespaces are used to isolate PODs from processing resources, therefore being able to clone or create a different namespace leaves the door open for the attacker to leverage the **unshare** system call to gain **CAP_SYS_ADMIN** privileges. In March 2022, a vulnerability was identified from the incorrect use of the UNIX pipe handling, the **CVE-2022-0847** affecting Linux Kernels from version 5.8, which was later renamed to **Dirty-Pipe**, allowing an attacker to overwrite files on a system with arbitrary data. As discussed in *Chapter 4, Securing Container Images and Registries, Container image hardening*, a container image is a collection of read-only layers merged by the container runtime (containerd, Docker, CRI-O). There is only one read-write layer, which is ephemeral in nature, created on top of all the other layers using the **Copy-On-Write** (**COW**) mechanism. When the container is destroyed, the read-write layer is also destroyed.

A proof of concept uses **gcc** at the **RUN** directive in the Dockerfile. The gcc compiler runs a program written in C language, which is used to write files on the filesystem. After logging in to the container, even if the files are read-only and owned by root and the filesystem is also read-only, it can overwrite data by abusing the standard way in which Linux controls file access. The danger in the Dirty Pipe hack is that killing the container does not fix the issue; once the files have been overwritten with the arbitrary data, they will persist in the read-only layer of the container image that

https://t.me/learningnets

was attacked, and any container created using that layer will be affected by that attack vector.

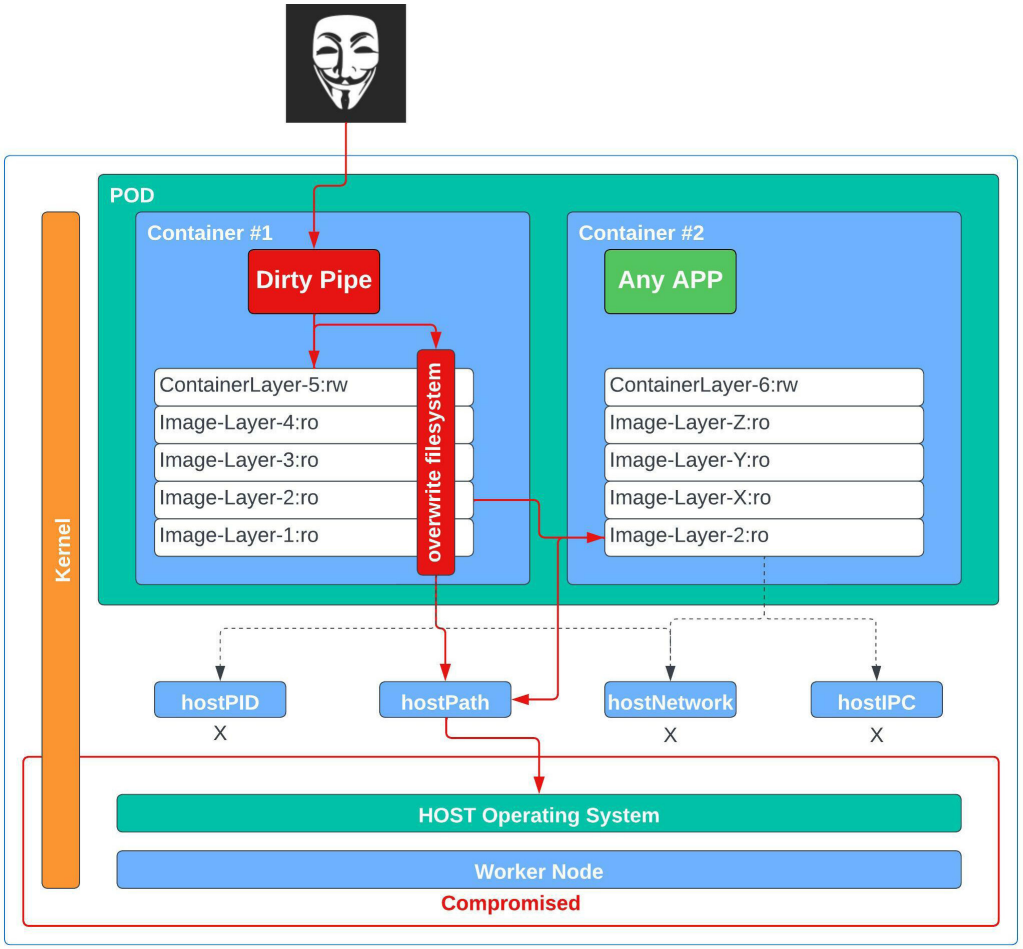

***Figure 7.6**: Dirty Pipe*

Mounting host volume as **`HostPath Mounts`** into the PODs is a discouraged technique, and it is a practice that should be acknowledged as dangerous, but it is still happening, especially when users feel safe by using the **`read-only`** flag. With Dirty Pipe, mounting the host volume in read-only has no use, because this vulnerability will be able to overcome the limitation and compromise the worker node, as illustrated in *Figure 7.6, Dirty Pipe*. The only way to really mitigate the **Dirty Pipe** is to upgrade the host systems with a kernel version that is not affected by the vulnerability.

https://t.me/learningnets

Hardening tools

Kubernetes adoption is de-facto a successful journey, thanks to the enormous efforts of over 30,000 individual contributors working effortlessly. In such a wide landscape, it is "normal" to end up considering additional tooling to increase the experience around Kubernetes. This section of the chapter aims to illustrate some, but not all, security-related tooling that are worth considering in any environment.

Before considering "external tools" though, it is important to draw attention to some of Kubernetes' internal solutions. Due to the extensive popularity of Docker, users think about any tool that relates to containers in terms of a Docker tool. While we have discussed many of these during our journey in the first part of this book, we also have acknowledged that Kubernetes has introduced a new mechanism called CRI, which supports native container engines like **`containerd`** and CRI-O, through the use of a container runtime shim demon like **`dockershim`**.

Each container engine has its own mechanism and complexity and can handle containers differently and with different tools. While the users have no issue using Docker commands, it is understandable that not many would know how to interact with containerd or CRI-O. Kubernetes, through the container runtime interface, has defined a standard that can be quite handy and efficient. The kubelet makes requests to the container engine via the container runtime interface, and it does not really need to know which brand the container engine has; all that the kubelet wants is a response to a request, whether it is a **`docker run`**, a **`nerdctl run`**, or anything else.

Note: At the time of writing this book, nerdctl is the official command-line interface command for containerd. While CRI-O provides a command-line tool, this is mainly used for testing CRI-O container engine and not for managing containers.

That's the reason why Kubernetes has introduced a command-line interface command for the container runtime interface, called **`crictl`**. This very useful, mostly unknown, command is part of the Kubernetes cri-tools suite, along with a validation tool called **`critest`**. It is worth remembering that as debug tool, the PODs created by crictl will not follow the normal life cycle, and the kubelet will eventually take them down to preserve the integrity of the cluster. To debug Kubernetes worker nodes, **`crictl`** offers, on top of the classic **Docker** commands, the options to list containers and PODs and run a POD as a sandbox. This is a useful mechanism to create containers with the container images available in the cluster, and to verify the deployment beforehand. To create a POD sandbox, refer to the following code example:

```
1. $ cat <<EOF | sudo tee pod-conf.json
2. {
3.     "metadata": {
```

https://t.me/learningnets

```
4.          "name": "nginx-sandbox",
5.          "namespace": "default",
6.          "attempt": 1,
7.          "uid": " 1000"
8.      },
9.      "log_directory": "/tmp",
10.     "linux": {
11.     }
12. }
13. EOF
14. $ crictl runp pod-conf.json
15. 3933c51901c112ef7d4162e25e7e4726869436779f06c40fa24c833153840de5
```

To create a container inside the POD sandbox, use the code following:

```
1. $ cat <<EOF | sudo tee container-conf.json
2. {
3.    "metadata": {
4.        "name": "alpine"
5.    },
6.    "image":{
7.        "image": "alpine"
8.    },
9.    "command": [
10.       "ls"
11.   ],
12.   "log_path":"busybox.0.log",
13.   "linux": {
14.   }
15. }
```

Then, run the following command that recalls both the pod configuration and the container configuration files:

https://t.me/learningnets

```
1. $ sudo crictl run container-conf.json pod-conf.json
2. cfc5f9157892481c5fd5df0646813fee6df97c401342f2540a3c11418d11c2bb
```

The POD sandbox defines the boundaries of the containers in a closed space, where applications can be tested safely. When thinking about tools that are not internal to Kubernetes, many are worth examining, but that would be sidetracking the focus to an infinite list of external application that could potentially be useful only to specific arguments. When considering Kubernetes from the software development life cycle standpoint, and therefore looking at this with DevSecOps eyes, four domains need cover, as illustrated in *Table 7.15, External Hardening Tools*:

Type	Description
SAST (Static Application Security Testing)	Checkov
Hardening	Kube-bench
Threat Analyzer	Kube-hunter
DAST (Dynamic Application Security Testing)	Falco

***Table 7.15**: External Hardening Tools*

Today, everything is automation, compute resources can be deployed with tools like Ansible, Chef and Puppet. The gap those tools were not able to fulfil is closed by the adoption of **Terraform**, the most popular **Infrastructure as Code (IaC)** tool. With Terraform resources like virtual private networks, gateways, subnets, clusters, and identify and access management can be remotely deployed, also into the cloud, to sustain compute resources needed to run any workload. A security tool that can analyze the code prior to the infrastructure deployment is essential for keeping the environment secure.

Checkov is one of the tools of this category, and it also covers SCA (Software Composition Analysis) for open-source packages and images. This tool is a powerful system and is capable of implementation via Kubernetes, AWS CloudFormation, Helm, Dockerfile, Azure ARM Templates and Serverless. It can also be implemented with versioning systems like GitHub, GitLab, Circle CI, and automated in CI/CD systems like Jenkins.

Hardening is not a trivial task, and it could be challenging to achieve in complex systems like Kubernetes. There are security frameworks that address Kubernetes Hardening methodologies: the **Center for Internet Security** (**CIS**) **Kubernetes Benchmark**, and the **National Security Agency Cybersecurity and Infrastructure Security Agency** (**NSA CISA**) **Kubernetes Hardening Guide**. The quantity of information provided by both frameworks is outstanding, and implementing such security controls manually would be very difficult. **Kube-bench** is a hardening tool designed to verify that Kubernetes has been deployed securely by executing checks documented in the **CIS Kubernetes Benchmark** framework.

https://t.me/learningnets

Note: The OWASP (Open Web Application Security Project) Foundation has recently introduced the Kubernetes Top 10, a list of the top 10 security issues affecting organizations when they decide to adopt the container orchestrator.

The Mitre has also defined the **Mitre ATT&CK Kubernetes Matrix**, which is a complex attack framework, and the threat analyzer system **kube-hunter** is mapping the attack techniques discussed in that framework in conjunction with a collection of in-house creative attack methodologies to mimic attackers inside or outside the cluster. **`Kube-hunter`** supports four scanning options, as listed in *Table 7.16, Kube-hunter Scanning Options*:

Type	Description
Remote	Specifies a remote node for hunting
Interface	Scans all the networks interfaces into the cluster
Network	Scans a specific CIDR
Auto-discovery	Connects to the kube-apiserver to retrieve a list of resources

***Table 7.16**: Kube-hunter Scanning Options*

Falco is a runtime security tool designed to inspect kernel events and aggregate those events with metadata collected from the various components of the Kubernetes cluster. Falco has a rich collection of inbuilt security controls, including the following:

- A shell has been invoked in a POD or a container.
- The container mounts sensitive paths, such as **`/proc`**.
- A process in the container is generating unexpected child processes.
- The container operates in privileged mode.
- Sensitive data read, such as **`/etc/shadows`**.
- Known binaries invoking outbound network connections, such as **`ls`**.
- Non-device files are written in the device space **`/dev`**.
- A POD with privileged access is started.

Falco comes with a plugin system to enhance the detection capabilities; some of them are described in *Table 7.17, Falco plugins*:

Type	Description
k8saudit	Ingest Kubernetes audit events.
cloudtrail	Connects to AWS S3 to read CloudTrail data.
docker	Reads Docker events.

https://t.me/learningnets

Type	Description
okta	Reads OKTA events.
github	Reads the Webhooks events.
k8saudit-eks	Ingest Kubernetes audit events for AWS EKS.

***Table 7.17**: Falco plugins*

Updating life cycle

Kubernetes is an infrastructure system, a cluster that serves the deployment of applications through the containers, so its life cycle from the software development standpoint should be reduced, or it should be slower than a conventional application; nevertheless, it is common for the Kubernetes community to release, on average, three major versions per year, sometimes even more. Kubernetes upgrade of the cluster is not a trivial task; there are many moving parts, such as kube-apiserver, etcd, kube-controller-manager, kube-scheduler, and the worker node components. This is where smart tools like **kubeadm** come into play.

Kubeadm is a tool built by the Kubernetes community to manage complex processes in a simple way. Many underestimate the value of such tool, especially users who do not usually need **K8s** (abbreviation for Kubernetes) in production environments. *Table 7.18, Kubeadm Features* provides an overview of the kubeadm features:

Type	Description
kubeadm init	initializes a control-plane node
kubeadm join	initializes a worker node
kubeadm upgrade	upgrades the cluster to a new version
kubeadm token	manages token for **kubeadm join**
kubeadm reset	reverts changes made by **kubeadm init** or **kubeadm join**
kubeadm certs	manages certificates
kubeadm kubeconfig	manages kubeconfig files

***Table 7.18**: Kubeadm Features*

Among the various aspects covered by kubeadm in the previous table, **`kubeadm token`** and **`kubeadm certs`** are the most relevant to this section of the book. When a new node is initialized via **`kubeadm join`**, a token is generated to establish bidirectional trust between the worker node and the control plane. However, even if this concept was created to serve kubeadm, it can also be utilized as an RBAC (Role-Based Access Control) policy. Each token has a correspondent secret in the **`kube-system`** namespace and has the following structure:

https://t.me/learningnets

```
1. apiVersion: v1
2. kind: Secret
3. metadata:
4.   name: bootstrap-token-mytoken
5.   namespace: kube-system
6. type: bootstrap.kubernetes.io/token
7. stringData:
8.   description: “The default token generated by kubeadm init”
9.   token-id: mytoken
10.  token-secret: a-secret
11.  expiration: 2022-12-10T07:51:33Z
12.  usage-bootstrap-authentication: “true”
13.  usage-bootstrap-signing: “true”
14.   auth-extra-groups:  system:bootstrappers:worker,system:bootstrap-
   pers:ingress
```

Tokens can be created, deleted, generated and listed via the **kubeadm token** command. Not many know that the client certificates created by kubeadm have a 1-year life cycle, after which they expire. The following command-line arguments are then useful to verify the status of the certificates and manage them accordingly:

- kubeadm certs renew
- kubeadm certs certificate-key
- kubeadm certs check-expiration
- kubeadm certs generate-csr

The certificates are renewed automatically when **`kubeadm upgrade`** is invoked and an upgrade of the Kubernetes cluster is, therefore, executed.

Conclusion

In this chapter, we discussed the various aspects of Kubernetes architecture and looked at how the various components of the control plane and the worker nodes impact the security of the whole cluster. We discussed the main aspect of securing network communication within the cluster, and also analyzed the threat of non-secure external communications. We also analyzed why securing the container runtime interface is important and explored how to secure the minimal deployable

https://t.me/learningnets

unit in Kubernetes, the POD, and the security concerns, and the threat and attack vectors around it. Finally, we touched upon interesting hardening tools and understood why it is important to keep Kubernetes updated.

In the next chapter, we will learn about Kubernetes and the various aspects of orchestration security.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

https://t.me/learningnets

Chapter 8
Kubernetes Orchestration Security

Introduction

The term "**orchestration**" recalls the music practice of combining several music instrument families for the purpose of achieving the intended performed melody. In this sense, Kubernetes assumes the same meaning: different container systems (Web servers, databases, message brokers) are deployed to work together, aiming to deliver the application that will serve that specific business goal. In Kubernetes, as for the orchestra, the individual components of instrument families can be replicated, increased or reduced, also known as **scaling**, according to the specific needs of requested music performance. Hence, this parallelism brings us to consider the scaling of containerized applications, as the volume of single container applications increases or decreases.

Container orchestration is not only scaling; many other aspects must also be considered for a comprehensive analysis of the container orchestrator potentiality: automation of provisioning, deploying, networking, and managing of containers. The Kubernetes hardening process created by the **National Security Agency** (**NSA**) and the **Cybersecurity and Infrastructure Security Agency** (**CISA**), discussed in *Chapter 7, Kubernetes Hardening*, highlights the three main sources of compromised for the Kubernetes system; please refer to *Table 7.1, Kubernetes Compromise Sources*, and to *Table 7.2, Kubernetes Threats Reference Matrix*.

https://t.me/learningnets

Kubernetes is a complex system made of several moving parts combined as a single platform, and it can deploy, scale and manage thousands of containers at any given time. While this sounds amazing from a DevOps perspective, achieving a good securing posture can be extremely challenging; therefore, while we have discussed the individual security requirements of Kubernetes' main components in the previous chapter, the focus of this chapter switches to achieving security from the orchestration standpoint. To provide a better prospective on the various topics treated by the National Security Agency CISA Kubernetes Hardening Guide, the **Malicious threat actors** and the **Supply chain** source of compromise discussed in the previous chapter are mapped and summarized in *Figure 8.1, NSA and CISA Kubernetes Hardening Guide Reference Table*:

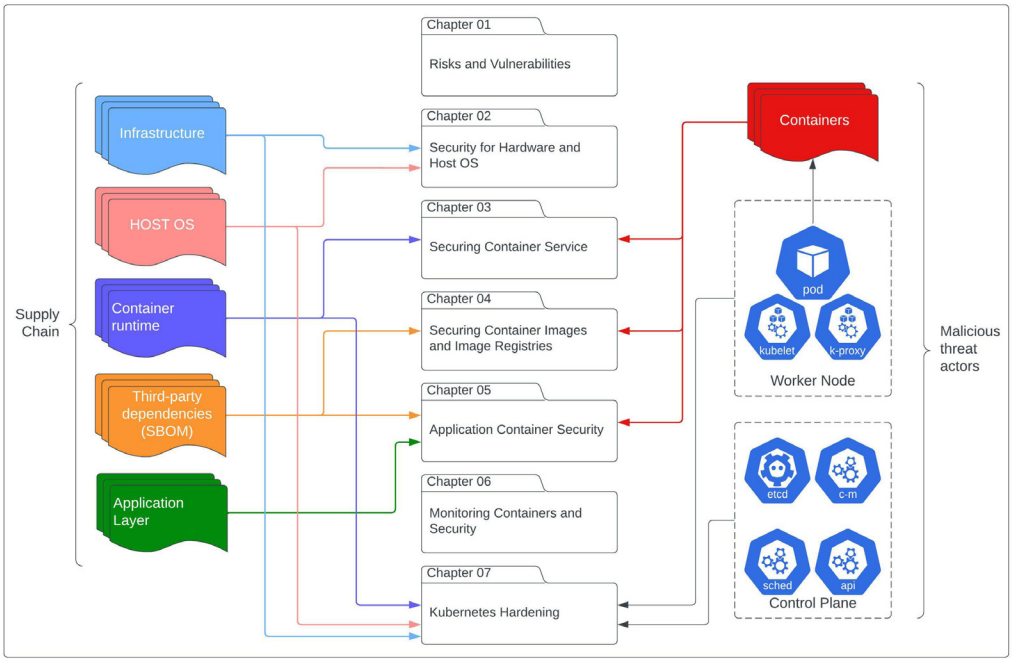

***Figure 8.1**: NSA and CISA Kubernetes Hardening Guide Reference Table*

This chapter will then focus on the third and final Kubernetes hardening outstanding topic: the "insider threats". As we continue this journey in the Kubernetes environment created in the previous chapter, the reference to files location or code is extracted from the **minikube** cluster or the Kubernetes cluster, as needed.

Structure

In this chapter, we will discuss the following topics:

- Authentication and authorization

https://t.me/learningnets

- API bypass risks
- RBAC vs ABAC
- Admission controller
- Securing secrets
- Cluster isolation
- Audit logging
- POD escaping privilege escalation
- Assess and verify

Objectives

The aim of this chapter is to outline security best practices for securing the Kubernetes platform in a DevSecOps way, facing the security complexity of the cluster from the orchestration standpoint and simultaneously fulfilling the security requirements described in *Table 7.2, Kubernetes Threats Reference Matrix*, in the *Insider Threats* section of the previous chapter.

Authentication and authorization

Authentication and authorization in Kubernetes play a very important part in securing the orchestration platform. These two terms are often considered interchangeable, but in reality, they are very different. In essence, authentication is the process that verifies the identity of the user that is trying to access a given system, while authorization is the process to verify that the user has the permissions to perform that action. Kubernetes provides two categories of users as per *Table 8.1, Kubernetes Users*, in a very standard approach:

Type	Description
service accounts	machine users managed by Kubernetes itself
users	allocated to humans

***Table 8.1**: Kubernetes users*

Users cannot be added through API calls; indeed, Kubernetes does not have an object to represent user accounts, but any access attempt made by users with a valid certificate that has been signed by the certificate authority of the Kubernetes system itself is considered trusted, and therefore, authenticated. Service accounts are created and managed by the **`kube-apiserver`**; they are bound to pre-determined namespaces, and they use credentials stored as secrets accessible by the PODs to allow in-cluster communications and processing.

https://t.me/learningnets

Note: All the requests must be authenticated to be executed, including both the users and service accounts. Non-authenticated requests are treated as anonymous and rejected unless anonymous access is enabled, which is, of course, discouraged.

Kubernetes offers several authentication strategies, which are listed in *Table 8.2, Authenticators Plug-ins*, but they all follow the same process:

- The client provides the credentials to the kube-apiserver.
- The kube-apiserver calls the authentication plugin to verify the identity through an identity provider.
- The identity provider verifies the credentials provided by the client.
- If the credentials are ok, the kube-apiserver checks the permissions as per the authorization mechanism, or it returns an **HTTP 401 Unauthorized** and error blocks the request.

Type	Description
X509 Client Certs	`--client-ca-file=/etc/kubernetes/pki/etcd/ca.crt` (installed by default)
Static Token File	Enabled by passing to the server `--token-auth-file=your_token`, the bearer token contains at least token, username and userid
Bootstrap Tokens	Stored as secrets in the Kubernetes `kube-system` namespace, with the `--enable-bootstrap-token-auth` flag, it is used for a new cluster bootstrapping
Service Account Tokens	Created automatically by the `kube-apiserver` and leveraged by the PODs to communicate within the cluster
OpenID Connect Tokens	User authentication layer on top of the **OAuth 2.0** stack; works with Azure AD and Google
Webhook Token	Rest API call outside the cluster domain
Authenticating Proxy	Uses a proxy that verifies the user identity, but it needs to present a valid client certificate to the `kube-apiserver` for validation
client-go credentials	Enable Kubernetes to implement non-natively supported authentication protocols such as OAuth2, Kerberos, LDAP, and SAML
Client API access	Verifies your client authentication methodology against the **kube-apiserver**

***Table 8.2**: Authenticators Plug-ins*

https://t.me/learningnets

From Kubernetes version 1.22.0, the **user impersonation** feature was added, adding the interesting capability of a user that can achieve authentication acting as another user. The user impersonation feature was introduced with the principle of least-privilege in mind, where users should have the minimum set of permissions required, and they should only elevate that set on a need basis and temporary.

This feature is different from the role base access control that we will discuss in the next section of this chapter, because it does not set permissions based on roles but allows users authenticated onto the cluster with an initial set of permissions (or role) to assume another, higher set of permissions aiming to complete a task that they would usually not be allowed to.

For this example, we are going to create a **`devops-team`** group bound to a view-only role and a **`devops-team-admin`** role with permissions to change or modify things in the cluster. The devops-team group is logged in to the cluster initially with view permissions only; when they need to change things, they can invoke **`kubectl`** with the **`--as=devops-team-admin`** parameter via the user impersonation feature to achieve the level of permissions set for the devops-team-admin user. The following code shows how to create the **`devops-team`** role binding to access the cluster in view only mode:

```
1. apiVersion: rbac.authorization.k8s.io/v1
2. kind: RoleBinding
3. metadata:
4.   name: devops-team-view
5.   namespace: devops-team
6. roleRef:
7.   apiGroup: rbac.authorization.k8s.io
8.   kind: ClusterRole
9.   name: view
10. subjects:
11. - apiGroup: rbac.authorization.k8s.io
12.   kind: Group
13.   name: devops-team
```

Then, use the following example code for the **`devops-team-admin`** role:

```
1. apiVersion: rbac.authorization.k8s.io/v1
2. kind: RoleBinding
3. metadata:
```

https://t.me/learningnets

```
4.    name: devops-team-admin
5.    namespace: devops-team
6.  roleRef:
7.    apiGroup: rbac.authorization.k8s.io
8.    kind: ClusterRole
9.    name: admin
10. subjects:
11. - apiGroup: rbac.authorization.k8s.io
12.   kind: User
13.   name: devops-team-admin
```

Kubernetes will need to apply the user impersonator feature by creating a cluster role to allow the assumption of the new set of permissions, as shown in the following code:

```
1.  apiVersion: rbac.authorization.k8s.io/v1
2.  kind: ClusterRole
3.  metadata:
4.    name: devops-team-impersonator
5.  rules:
6.  - apiGroups: [""]
7.    resources: ["users"]
8.    verbs: ["impersonate"]
9.    resourceNames: ["devops-team-admin"]
```

And finally, use the cluster role binding for the **devops-team-admin** user impersonator feature:

```
1.  apiVersion: rbac.authorization.k8s.io/v1
2.  kind: ClusterRoleBinding
```

https://t.me/learningnets

```
3.  metadata:
4.    name: devops-team-admin-impersonate
5.  roleRef:
6.    apiGroup: rbac.authorization.k8s.io
7.    kind: ClusterRole
8.    name: devops-team-impersonator
9.  subjects:
10. - apiGroup: rbac.authorization.k8s.io
11.   kind: Group
12.   name: devops-team
```

This concept recalls in principle the Linux **sudo** command, and readers may look for something similar when using the command line interface to connect to their cluster to preserver the least-privilege security principle: introducing **kubectl-sudo**. Aiming to reduce the attack surface, **kubectl-sudo** is not a Kubernetes plugin per se but a way of providing a sudo like mechanism to access the orchestrator. The logic is to reduce the cluster administrators default privileges to the minimum and grant them the capability to impersonate users and groups with a higher level of permissions called masters.

Interestingly, Cloudogu, a German consulting company focused on DevOps toolchain containerized software development, has two more open-source projects along the line of the same Linux sudo principle: **helm-sudo** and **sudo-kubeconfig**. Any authorization request in Kubernetes is processed only upon authentication, so the cluster expects the entity to be logged in before verifying the granted level of permissions. The request must contain attributes common to any REST API system, using the well-known Linux firewall approach, where everything is denied by default, except what is specifically allowed.

In Kubernetes clusters with multiple authorization mechanism enabled, each request is checked in sequence. If any of the authorization modules denies or approves a request, the result is immediately acknowledged by the cluster, and no other authorization system is involved. If all the authorization modules cannot either approve or deny the request, by default the request is denied. *Figure 8.2*,

https://t.me/learningnets

Authentication and Authorization, provides a visual representation of the model described so far:

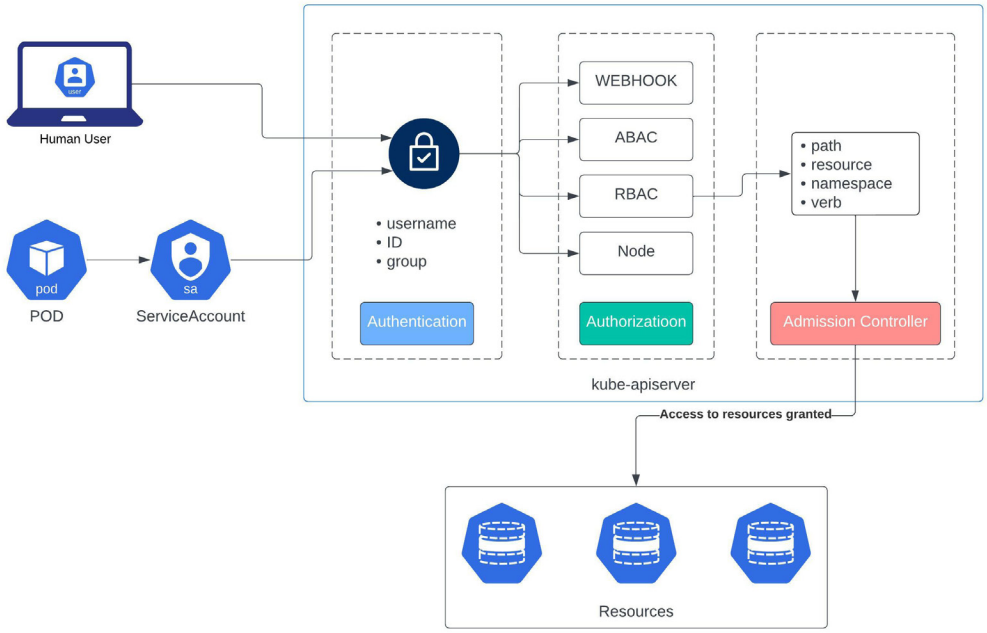

***Figure 8.2**: Authentication and Authorization*

The authorization modes supported by Kubernetes are listed in *Table 8.3, Kubernetes Authorization Modes*:

Type	Description
Node	Authorization mechanism specifically built for **kubelet**
ABAC	Attribute-based Access Control; authorization is granted via policies
RBAC	Role-based Access Control; access control based on roles that users can assume
Webhook	HTTP callback event executed through POST

***Table 8.3**: Kubernetes Authorization Modes*

One of the interesting functions provided by the authorization mechanism is the **can-I** parameter, with which users can check whether they are allowed to execute a certain command. The following command query the cluster to check whether the user logged in can create secrets in the **`myspace`** namespace:

```
1. $ kubectl auth can-i create secrets--namespace myspace
```

https://t.me/learningnets

If this function is combined with the **user impersonation** mechanism explained earlier, the users logged in can check whether a given function is allowed while elevating temporary their privileges, as per the following command:

```
1. $ kubectl auth can-i create secrets --namespace myspace --as devops-
   team-admin
```

Users with permissions to create or edit PODs in a namespace potentially have the capability of escalating their privileges in that specific namespace. There are a few escalation paths that should be carefully considered from the administration standpoint:

- Allowing namespaces to mount arbitrary secrets can lead to accessing secrets meant for other type of workloads or obtaining a service account token with higher permissions.
- Allowing namespace to use arbitrary service accounts can lead the **kube-apiserver** to act on other workloads using the impersonation mechanism or execute privileged actions as source of compromise.
- Allowing namespace to mount **configmaps** can lead to gathering information related to other workloads.
- Allowing namespace to mount volumes can lead to gathering information related to other workloads and potentially change it.

When building custom resource definitions affecting these four areas, administrators should consider the impact of such changes because they could potentially lead to privilege escalation paths.

API bypass risks

The **kube-apiserver** should be hardened by adopting a **Transport Layer Security** (**TLS**) certificate. When the handshake is established and verified via TLS, the request reaches the authentication step. The authentication depends on the modules installed in the cluster, as listed in *Table 8.2, Authenticators Plug-ins*, while the authorization mechanism is expecting the request to present the username, the action to be performed and the target object of the action. To understand the logic behind the API bypass risk, let's assume that the user Luigi is logged in to the cluster and has the following policy that grants him read-only permissions in the **devops-team** namespace:

```
1. {
2.     "apiVersion": "abac.authorization.kubernetes.io/v1beta1",
3.     "kind": "Policy",
```

https://t.me/learningnets

```
    "spec": {
        "user": "Luigi",
        "namespace": "devops-team",
        "resource": "pods",
        "readonly": true
    }
}
```

Therefore, the following action is authorized:

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "spec": {
    "resourceAttributes": {
      "namespace": "devops-team",
      "verb": "get",
      "group": "devops-team",
      "resource": "pods"
    }
  }
}
```

But if user Luigi is going to create or update anything in the same namespace, the request will be denied:

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "spec": {
    "resourceAttributes": {
      "namespace": "devops-team",
      "verb": "put",
      "group": "devops-team",
      "resource": "pods"
```

https://t.me/learningnets

```
10.     }
11.   }
12. }
```

The preceding code is clear and straightforward. It is part of the functional logic around the API server and its interaction with human users, where the authentication and authorization are controlled by administrators with tools like RBAC, ABAC, Admission controller or policies. Nevertheless, there are four possible sources of compromise that readers should consider when thinking about **`kube-apiserver`** security: Static PODs, Kubelet API, etcd API, and Container runtime socket. These four are known as the **`kube-apiserver` bypass risks**.

Static PODs are managed directly by the **kubelet** system on the node itself and the kube-apiserver has no visibility of them. They are not to be confused with PODs managed by the control plane, for example, PODs created via deployment. Static Pods are special PODs used in cases like bootstrapping or are used by the cluster itself to spin up control plane components like Controller Manager, Scheduler or the API Server itself as Static PODs. When the Kubernetes cluster is created, the **`kubeadm`** also creates PODs manifest files in a folder that is monitored by the **`kubelet`** system. Each kubelet system directly manages any manifests that have not been created via user input as Static PODs, and since they are not visible to the API Server, the kubelet also tries to spin up a mirror POD in the kube-apiserver. An attacker with permissions to amend the manifest files could inject modified Static PODs to assume control of various components of the control plane or compromise the node by mounting **hostPath** from the underlying host. If the Static POD fails the admission control, it won't be registered as mirror POD by the kube-apiserver, but it will still run on the node as a possible source of compromise.

The **kubelet** system running on the worker node exposes an API system, not to be confused with the **`kube-apiserver`**, on port 10250; and depending on the Kubernetes distribution, it could also be exposed on the control plane nodes. When users have RBAC access permissions to the **Node** object and related resources, that access also grants them the authorization to talk to the kubelet API. Furthermore, this kind of access to the kubelet API is not logged by any audit system and does not need to undergo the admission control mechanism. An attacker with access to the kubelet API may be able to bypass any authorization mechanism.

As explained in the previous chapter, Kubernetes uses **`etcd`** as datastore. The only control plane component that needs access to etcd is the kube-apiserver. Similar to the kubelet API, the etcd system also has its own API system, and the etcd API is not logged by any audit system and does not need to undergo the admission control mechanism. Access to the etcd API is managed by default via client certificate authentication. Any certificate generated by a **Certificate Authority** (**CA**) trusted by the etcd system grants full access to the datastore. An attacker who can retrieve the

https://t.me/learningnets

certificate private key or create a trusted certificate can therefore, obtain administrator permissions and access the datastore.

As discussed in *Chapter 7, Kubernetes Hardening*, in the *Securing container runtime interface* section, container runtime is the software that manages containers on the worker node. The runtime communicates with the kubelet system on the worker node via a Unix socket. An attacker that can access the Unix socket on the container runtime can therefore, access and manage running containers or create new containers eventually. The impact is defined by whether the compromised containers have access to **secrets**, the vulnerabilities that could be exploited or the information the attacker could use to escalate privileges in order to gain access to the underlying host, or compromise one or more control plane components.

For each of the four security issues affecting the Kubernetes API Server that we discussed so far, mitigation methodologies can be placed to reduce the risk surface, as described in *Table 8.4, API Bypass Risks Mitigations*:

Risk	Mitigation
Static PODs	• Enable Static POD manifest if required by the worker node • Restrict users access to the manifest folder • Restrict users access to the kubelet configurations • Audit storage locations for the kubelet configuration files and Static PODs manifests
kubelet API	• Restrict access to the **node** object resources • Allow a trusted IP range to communicate with the kubelet API port • Verify that the kubelet authentication is configured in certificate mode • Disable kubelet port non-authenticated read-only
etcd API	• Verify that the trusted CA is used only to authenticate against the etcd system • Restrict access to the private key • Allow a trusted IP range to communicate with the etcd API port
Container Runtime Socket	• Restrict filesystem access only to the root user when possible • Use the Linux kernel namespace (not the Kubernetes namespace) to isolate the kubelet process from other services running on the worker node, for example, the **kube-proxy** • Do not allow **hostPath** mount • Restrict users access to the worker node

***Table 8.4**: API Bypass Risks Mitigations*

https://t.me/learningnets

RBAC vs ABAC

Role-based Access Control (**RBAC**) and **Attribute-based Access Control** (**ABAC**) are two of the four authorization modes discussed earlier in this chapter. RBAC is a method of controlling access to a system or network based on roles, while ABAC defines paradigms where access permissions are granted via policies. The RBAC defines four types of objects, as per *Table 8.5, RBAC Objects*:

Type	Description
Role	Contains rules to set permissions in a specified namespace
ClusterRole	Contains rules to set permissions in a non-namespaced resource
RoleBinding	Grants the Role permissions to a user or group in a specified namespace
ClusterRoleBinding	Grants the Role permissions to a user or group across the whole cluster

***Table 8.5**: RBAC Objects*

Note: Role and ClusterRole rules have no "deny" condition because they represent a permission set. The permission set implies "allowing rules", meaning that without rules there is no permission.

The RBAC API system forbids user escalation by the means of modifying roles or role bindings objects; enforcing this mechanism at the API layer, the cluster guarantees its application even when RBAC is not the chosen authorization mechanism. In an RBAC authorization mechanism scenario, users can create or update a role only if at least one of the following conditions is true:

- The user has all the permissions contained in the role within the same scope of the object affected by the change.
- The user has explicit permissions to escalate.

Similarly, users can create or update a role binding if one of the following conditions is true:

- The user has all the permissions contained in the referenced role.
- The user has explicit permissions on the **bind** object.

By default, RBAC does not grant any permissions to any **service account** that is not the **kube-system** namespace. This choice provides great flexibility to manage the service accounts across the whole cluster, and Kubernetes has a high granularity configuration capability that can result in time and effort to manage them. On the other side of this equation, administrators would feel comfortable, especially in

https://t.me/learningnets

development environments, granting broader permissions, just to ease the burden on the administration side.

Note: Readers with experience in cloud services like AWS Identity and Access Management (IAM) would recognize the ABAC approach when dealing with AWS IAM Policy.

The following description aims to identify a five-step incremental security approach that can be applied to the RBAC logic. The first of the five steps are the least secure; security increases with each step, and upon reaching the fifth step, the maximum degree of security is considered applied through RBAC, as illustrated in *Figure 8.3*, *Securing RBAC Service Account*:

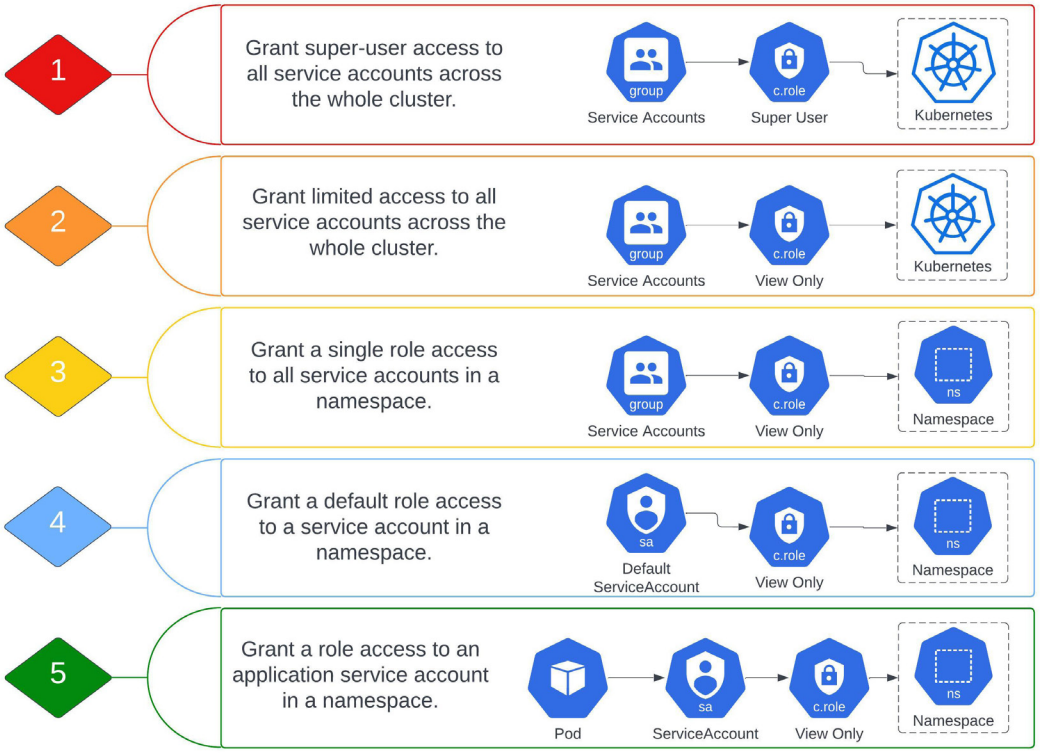

***Figure 8.3**: Securing RBAC Service Account*

Attribute-based access control (**ABAC**) as an authorization mechanism became prominent over a decade ago, as part of an attempt to improve US federal organization's access control architecture. It is indeed the evolution of the access control lists because the logic behind it is based on policies. ABAC provides great flexibility and granularity, but it is computationally intensive, due to the quantity of attributes that must be considered and the conditions applied during

https://t.me/learningnets

the authorization process. The key logic of the ABAC approach is the capability to not need any specific relationship between the subject and the object of the policy, meaning that users granted access to a specific resource do not necessarily need to be related to that resource.

This is in contrast with the RBAC model, where tendentially administrators create a logic relationship between the users and the groups with the resource, for example, users working in the human resource department are granted permissions to access human resources-related systems and are denied access to the developer system with that very same role. Clusters running older versions of Kubernetes commonly use ABAC permissive policies, which poses a threat to the integrity of the system because the authorization mechanism does not define an authorization process priority; in essence, any authorization module can pick up the authorization request.

Moving from an ABAC approach to an RBAC approach, which is far more secure, can be a painful exercise and can disrupt the existing running workloads. Moving from a permissive ABAC policy to a permissive RBAC role, such as number one in *Figure 8.3, Securing RBAC Service Account*, is highly discouraged because it goes against security best practices. The recommended way to solve this issue is to use the "**parallel authorizers**" method. Instruct the cluster to run RBAC and ABAC, specifying the ABAC policy that needs to be ported into a RBAC role, as with the following command line parameter:

```
1. --authorization-mode=RBAC,ABAC      --authorization-policy-file=abac-
   legacy-policy.json
```

To gather information on porting the ABAC policy into the RBAC policy, the **kube-apiserver** produces authentication and authorization logs with a minimum log level of 5. For all the API calls rejected by the RBAC authorization module and granted by the ABAC authorization module, it possible to identify users, groups or service accounts, which can be used subsequently to define a new RBAC role. When all the workloads are running with no disruptions and no RBAC, denial messages are logged by the kube-apiserver; the ABAC authorization module can be safely removed. An interesting tool to be considered to evaluate RBAC permissions is the **Palo Alto Networks** solution called **RBAC Police**. This tool can help identify the RBAC permissions of the Kubernetes users, groups, service accounts, PODs, and nodes, and evaluate them to detect which of the Kubernetes identities have risky permissions and can potentially pose as threat to the integrity of the cluster.

Admission controller

Admission controllers are Kubernetes cluster plugins used to enhance the most advanced security features. These plugins enforce and govern the cluster. In consideration of the access mechanism, the admission controller kicks-in as the third

https://t.me/learningnets

logic element, after the authentication and authorization processes have already been executed by the cluster. From Kubernetes 1.26, the default admission controllers are listed in *Table 8.6, Admission Controllers List*:

Type	Description
CertificateApproval	Monitors requests to approve CertificateSigningRequest
CertificateSigning	Monitors updates of CertificateSigningRequest
CertificateSubjectRestriction	Rejects requests of groups **system:masters**
DefaultIngressClass	Monitors **Ingress** objects with no specific ingress class and adds them to the default ingress
DefaultStorageClass	Monitors **PersistentVolumeClaim** objects with no specific storage class and add them to the default storage class
DefaultTolerationSeconds	Sets the tolerance in seconds for the **notready:NoExecute** and **unreachable:NoExecute**
LimitRanger	Ensures that incoming requests are not violating the constrains set in the **LimitRange** of the namespace
MutatingAdmissionWebhook	Is the first phase of the Admission Controller logic
NamespaceLifecycle	Ensures that terminating namespaces cannot have new objects
PersistentVolumeClaimResize	Validates the volume resize requests
PodSecurity	Replaces **PodSecurityPolicy** and validates POD creation requests
Priority	Evaluates POD creation priority. POD is rejected if no priority class is set
ResourceQuota	Enforces the resource quota assigned to the POD
RuntimeClass	Verifies and enforces the POD overhead configuration
ServiceAccount	Monitors **ServiceAccount** objects
StorageObjectInUseProtection	Protects storage objects from being deleted
TaintNodesByCondition	Controls PODs allocation
ValidatingAdmissionPolicy	Implements Common Expression Language for incoming requests
ValidatingAdmissionWebhook	Second phase of the Admission Controller logic

***Table 8.6**: Admission Controllers List*

The Admission Controller process has two phases, as illustrated in *Figure 8.4, Admission Controllers*: the **mutating admission** and the **validating admission**. Admission controllers can behave as mutating controllers or validating controllers. For example, the **`LimitRanger`** described in *Table 8.6, Admission Controllers List*,

https://t.me/learningnets

assumes the validation admission (Phase 2) when it blocks PODs from exceeding the specific set or resources requirements within the namespace and assumes the mutating admission (Phase 1) when it allows PODs to augment the default resources limits. The difference between the two is self-explanatory: mutating admission can eventually mutate objects, although it can also reject API calls with mutating requests, and the validating admission cannot mutate objects; it can only enforce policies on the resources. The security advantages of the validating admission over the mutating admission is two folds: administrators would consider disabling the mutating admission to drastically increase the security posture of the cluster in the first place; and as per *Figure 8.4, Admission Controllers*, the validating admission is executed after the mutating one, so once a request is validated, it persists in the **`etcd`** datastore, even if a mutation has been validated by the system itself, decreasing the security posture of the cluster. Refer to the following figure:

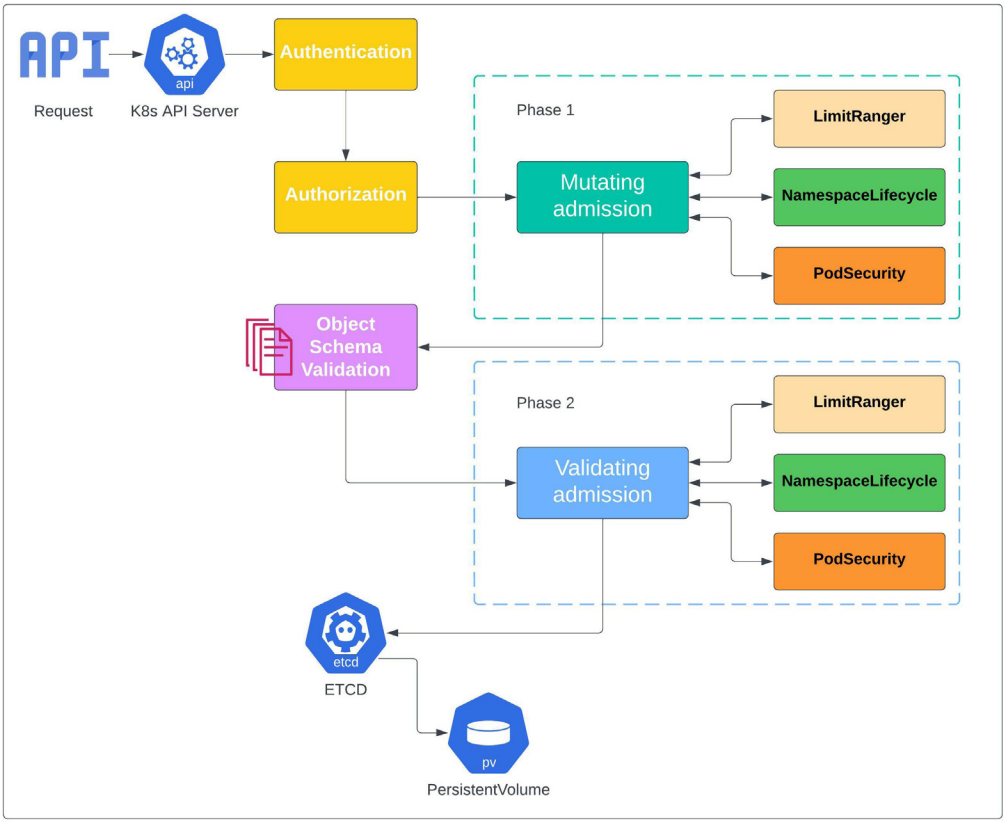

***Figure 8.4**: Admission Controllers*

The main security advantages of using admission controllers are listed in *Table 8.7, Admission Controller Advantages*:

https://t.me/learningnets

Type	Description
Security	Creating a security baseline that can be applied to all the namespaces or to the entire cluster
Governance	Enforcing adoption of best security practices such as labelling, resource limitation, and annotations
Configuration Management	Validating the configuration of resources running in the cluster and prevent misconfigurations that can affect the cluster security posture

***Table 8.7**: Admission Controller Advantages*

Among all the admission controllers, **`PodSecurity`** is likely the most prominent and interesting as it directly affects the container workloads and the software development life cycle. The **Pod Security Standards** defined into the Kubernetes cluster has three different policies: **privileged**, **baseline** and **restricted**. The privileged policy is completely unrestricted, the baseline policy offers a minimal restrictive approach and block known privilege escalations, and the restricted policy is the most secure one and follows best POD security hardening practice. It is possible to enforce the **Pod Security Standards** configuring the built-in **PodSecurity** admission controller at the cluster level, fulfilling the security orchestrator requirements in managing either default behaviors or exemptions. The following code provides an example of PodSecurity configuration:

```
1. apiVersion: apiserver.config.k8s.io/v1
2. kind: AdmissionConfiguration
3. plugins:
4. - name: PodSecurity
5.   configuration:
6.     apiVersion: pod-security.admission.config.k8s.io/v1
7.     kind: PodSecurityConfiguration
8.     defaults:
9.       enforce: "restricted"
10.      enforce-version: "latest"
11.      audit: " restricted "
12.      audit-version: "latest"
13.      warn: " restricted "
14.      warn-version: "latest"
15.    exemptions:
```

https://t.me/learningnets

```
16.       usernames: []
17.       runtimeClasses: []
18.       namespaces: []
```

In the above example, the "**defaults**" applies when the POD mode **Label** is not set with the latest policy version, and it provides no exemption to the policy. The **`defaults.labels`** are described in *Chapter 7, Kubernetes Hardening*, *Table 7.12, POD Security Standards*, as part of a broader discussion around POD Security.

Note: PODs are often created indirectly, for example, via deployment. In this case, the enforce label is not applied to the workload resources, only to the PODs. Audit and warning can help in modifying the deployment template to meet the security requirements.

Exemptions to the policy must be explicitly declared. A request that meets the defined policy exemptions is ignored and can bypass the policy enforcement. Based on this note, it is not recommended to exempt service accounts, as it would exempt any user that leverages that service account.

Securing secrets

Secrets are objects that contain sensitive information, such as tokens, passwords, and SSH keys. Kubernetes uses **secrets** to provide control on how sensitive information is managed and aims to reduce the risk of exposure. The default storage methodology for secrets in the cluster is not encrypted but encoded as **base64** strings, which is not sufficient from the security standpoint. The following is a list of recommended actions from the orchestration standpoint:

- Enable **etcd** encryption at rest
- Control access to **secrets**
- Improve datastore management policies
- Use external **secrets** management systems

The first step to enhance security around secrets is to enable encryption data at rest in **etcd**. This procedure requires etcd version 3.0 or higher and minimum Kubernetes version 1.26 to encrypt custom resources. Refer to *Chapter 7, Kubernetes Hardening*, *Control plane*, for a detailed description on how to enable etcd encryption at rest. As per discussion in the previous section about **Role-based Access Control** (**RBAC**), the least-privilege principle should be applied when managing access to secrets; this can be achieved by considering the approach defined in *Table 8.8 – Least-privilege access to secrets*:

https://t.me/learningnets

Type	Description
Components	• Restricts access to **list** or **watch** only to privileged system components • Allows the use of **get** access if it is necessary for the component
Humans	• Forbids access to use **list**, **get**, or **watch** to users, and allows only administrators with a view-only access role • For more fine-grained access control mechanism, consider using a third-party tool

***Table 8.8**: Least-privilege access to secrets*

Users that can create PODs have access to the secrets that the PODs invoke, even if the policies are forbidding users access to secrets. To mitigate this behavior, consider the following:

- Implementing auditing rules to alert on secrets specific events
- Using short-lived or one-time secrets

To improve datastore management policies, consider wiping out the etcd database storage volume when no longer in use, and if there are multiple etcd datastore instances, adopt **Transport Layer Security** (**TLS**) communication to protect data in transit, especially in high-availability clusters configuration scenarios.

It is not uncommon to use external secrets management system; this solution keeps sensitive information outside the domain of the cluster, so PODs must be configured to look up that information externally. This is possible thanks to the **Kubernetes Secrets Store CSI Driver**, essentially a DaemonSet that connects the kubelet with the external system. The external secrets management systems that are currently supported are AWS Provider, Azure Provider, GCP Provider and Hashicorp Vault Provider. There are few more actions worth considering, even if they do not apply specifically to cluster administration and are more directed to the application and container life cycle:

- Restrict access only to the container that needs it
- Protect secret after accessing it
- Do not share secret manifests

In an environment where a POD has multiple containers running, access to secrets should be allowed only to those containers that need authentication mechanism, by defining appropriate mounting variables. Containers should protect the value of the secrets after consuming it, so appropriate measure should be in place to avoid transmitting secrets to untrusted parties or logging the secrets in log systems. If a manifest file is adopted to configure secrets, appropriate security measures should

https://t.me/learningnets

be considered in controlling access to the manifest file, as secrets would be available to any user who can read the manifest.

Cluster isolation

A Kubernetes cluster is a complex environment and can be hard to manage; also, it is very common to "share" the usage of the cluster between teams, especially those that are logically closer to each other. For example, the same cluster could be used by dev, devops, testing, and quality or when different applications share the same member of the teams involved in the release cycle. If an application is used by the human resource or the facility team, they could share the same cluster with different levels of access. Sometimes it is the application that defines who needs access to what; the broader the breath of the application, the more it will span over multiple teams. These considerations should also include environment separation, such staging versus production, with some stakeholders accessing the cluster with a lower level of permissions and others with a higher level of permission, defining **coexisting tenancy model**. There are usually two levels of isolation, as described in *Table 8.9, Isolation Level*:

Type	Description
hard multi-tenancy	• Strong isolation • No trust • More difficult to achieve on the Data Plane • Regulatory and compliance requirements
soft multi-tenancy	• Soft isolation • Common for internal environments • Easier to achieve on the Control Plan

***Table 8.9**: Isolation Level*

There are a few ways to implement multi-tenant solutions and achieve isolation in Kubernetes; each of these methodologies has its own trade-off that affects the isolation and implementation level, complexity and management; but it usually comes down to two techniques: **Control Plane Isolation** and **Data Plane Isolation**. The main goal of the Control Plane Isolation method is to ensure that tenants cannot access or in any way affect each other's **`kube-apiserver`** and resources. Isolation is achieved as per the description in *Table 8.10, Control Plane Isolation*:

https://t.me/learningnets

Type	Description
Namespaces	• Address the isolation mechanism within a single cluster. • Objects with the same name can span in different namespaces. • Security policies can be scoped at the namespace layer. • In a multi-tenant environment, it helps segment the workload into logical units. • It requires configuration of other resources, such as networking.
Access controls	• Adopting the Least-Privilege principle to ensure the minimum access level is set. • Use RBAC to enforce authorization by enforcing Role and RoleBinding at the namespace layer. • In a multi-tenant environment, restrict access to the appropriate namespace with RBAC. • Do not grant access at the cluster level. • When a policy grants more permissions than it should, considering adopting a policy review process.
Quotas	• Manage physical node resources. • Limit the number of PODs per namespace. • Quota has no control over network traffic.

***Table 8.10**: Control Plane Isolation*

The Data Plane Isolation is referred to PODs and workload to ensure that tenants cannot access or in any way affect each other's running resources. Isolation is achieved as per the description in *Table 8.11, Data Plane Isolation*:

Type	Description
Network isolation	• All PODs can communicate to each other by default, and network traffic is unencrypted. • Consider adopting Network Policies. • Consider implementing a service mesh, which can help increasing the network isolation, acting at the OSI Layer 7.

https://t.me/learningnets

Storage isolation	• Adopt **Dynamic Volume Provisioning**. • Do not implement volume types that need node resources. • **PersistentVolumeClaim** is a resource confined by the namespace. • Be mindful of the **PersistentVolume** resource, which can span to the whole cluster. • Adopt **StorageClasses** to increase isolation.
Sandboxing	• Provides workloads isolation in shared cluster. • Recommended when running suspicious code or for troubleshooting purposes. • **gVisor** is a tool that intercepts system calls from the containers and executes them via a userspace kernel with limited and controlled access to the host node. • **Kata Containers** provides an additional virtualized environment allowing to run containers in a virtual machine inside the POD, and therefore, blocking the system calls to reaching the node.
Node isolation	• Running PODs are assigned to a specific tenant. • POD escaping is limited to the node environment. • Slightly more secure and easier to implement with respect to sandboxing. • It can be achieved using **POD Node Selectors** or **Virtual Kubelet**.

***Table 8.11**: Data Plane Isolation*

There are two methods to implement multi-tenancy for a Kubernetes cluster while preserving isolation: **namespace per tenant** and **virtual control plane per tenant**. The namespace per tenant approach is very well supported, implies namespace isolation, and provides a methodology to allow proper tenants communication, but it can be complicated to configure and cannot be used to isolate resources that cannot be confined into a namespace. On the contrary, the control plane virtualization per tenant approach can apply isolation to non-namespace Kubernetes resources but has much more difficult tenant-to-tenant interaction and a higher cost in terms of resource utilization.

Audit logging

The Kubernetes auditing system is a chronological record set that registers the actions within the Kubernetes cluster. The cluster records three set of activities:

https://t.me/learningnets

users, applications that interact with the **`kube-apiserver`**, and the control plane.

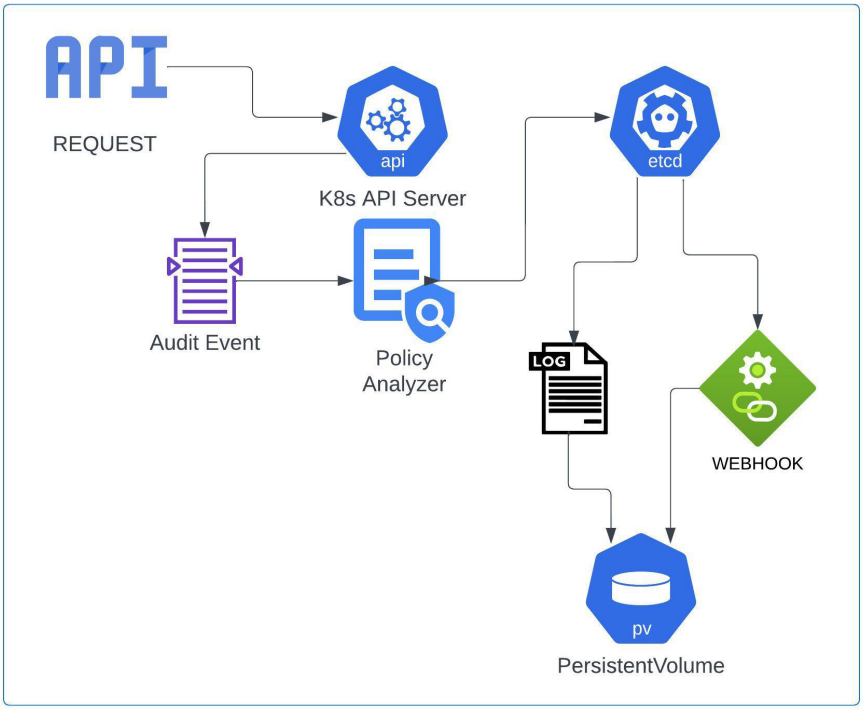

***Figure 8.5**: Auditing*

The audit system in Kubernetes follows the workflow illustrated in *Figure 8.5, Auditing*, where any request generates an audit event, the audit event is processed by an audit policy analyzer, which defines what to record from the audit event, and the event is then written to a persistent volume. The API requests ingested into the audit system have four possible stages, as described in *Table 8.12, Audit Request Stage*:

Type	Description
RequestReceived	It is generated as soon as the audit system received the request.
ResponseStarted	Only the response header is sent, not the response body.
ResponseComplete	The response body is sent, the audit request has been completed.
Panic	It is only generated by panic events.

***Table 8.12**: Audit Request Stage*

The audit policy defined into the audit system provides priority rules about what data should be recorded when the event is processed. The first matching

https://t.me/learningnets

rule determines the audit level. The audit levels are described in *Table 8.13, Audit Levels*:

Type	Description
None	Do not log events.
Metadata	Only log metadata, such as user, resource, timestamp, or verb.
Request	Log request body and its metadata, but do not log response body.
RequestResponse	Log request body, metadata, and response body.

***Table 8.13**: Audit Levels*

Kubernetes provides a sample audit policy that helps configure the audit system as per the following example code:

```
1. apiVersion: audit.k8s.io/v1
2. kind: Policy
3. omitStages:
4.   - "RequestReceived"
5. rules:
6.   - level: RequestResponse
7.     resources:
8.     - group: ""
9.       resources: ["pods"]
10.  - level: Metadata
11.    resources:
12.    - group: ""
13.      resources: ["pods/log", "pods/status"]
14.  - level: None
15.    resources:
16.    - group: ""
17.      resources: ["configmaps"]
18.      resourceNames: ["controller-leader"]
19.  - level: None
20.    users: ["system:kube-proxy"]
```

https://t.me/learningnets

```
    verbs: [“watch”]
    resources:
    - group: “”
      resources: [“endpoints”, “services”]
  - level: None
    userGroups: [“system:authenticated”]
    nonResourceURLs:
    - “/api*”
    - “/version”
  - level: Request
    resources:
    - group: “”
      resources: [“configmaps”]
      namespaces: [“kube-system”]
  - level: Metadata
    resources:
    - group: “”
      resources: [“secrets”, “configmaps”]
  - level: Request
    resources:
    - group: “”
    - group: “extensions”
  - level: Metadata
    omitStages:
      - “RequestReceived”
```

Policy objects cannot be created via **kubectl** command line; the **audit.k8s.io** would return a no match error. The policy can be invoked by modifying the kube-apiserver manifest file:

```
spec:
  containers:
  - command:
```

https://t.me/learningnets

```
4.      - kube-apiserver
5.      - --allow-privileged=false
6.      - --audit-policy-file=/home/docker/audit.yaml
```

Table 8.14, Audit Rules, provides a brief description of the above-mentioned configuration:

Line	Description
3	The **omitStages** forbid the cluster to generate audit events for all the RequestReceived; this will help in reducing the noise.
6	The RequestResponse level logs PODs changes.
9	"pods" does not match **resource** requests.
13	PODs log and status is logged at **Metadata** level.
18	The **configmap** resource **controller-leader** is not logged.
24	Resources **endpoints** and **services** with **watch** requests are not logged.
28	Non-resource **api** requests are not logged.
29	Non-resource **version** requests are not logged.
34	The namespace resource **kube-system** logs audit request body of **configmap** changes.
38	Resource **secrets** logs changes at the **Metadata** level.
39	Resource **configmap** logs changes at the **Metadata** level.
42	Logs at the request level resources in **extensions**.
45	Logs all the **RequestReceived** at the **Metadata** level.

***Table 8.14**: Audit Rules*

Note: The group "" with empty double quote is intended for the core API group.

For readers who want to create their own audit profile, it is recommended to use the Google Container-Optimized OS. Kubernetes has a **configure-helper.sh** script to generate a starting point audit policy that can be tailored to individual user needs. Audit events are stored with the **Log backend** and the **Webhook backend** mechanisms: the former writes JSON locally via **volumeMounts**, while the latter sends audit event to a remote web system to ingest logs into an external monitoring tool.

https://t.me/learningnets

POD escaping privilege escalation

In *Chapter 7, Kubernetes Hardening*, we discussed a possible scenario of **POD Escaping**, where the attacker can eventually escape the POD and gain access to the underlying host, aka the **worker node**. Worker nodes come with a set of pre-determined credentials based on the type of Kubernetes distribution or service installed; these credentials may vary, but they are usually divided into **kubelet** credentials (restricted via **NodeAuthorizer** and **NodeRestriction**), and **PODs Service Accounts** credentials.

Users logically associate PODs with applications, and therefore their focus is on how to manage workloads, but there are also cluster-managed PODs or PODs plug-in, with their own set of permissions via service account. These are usually running as a **DaemonSet** spanning two or more worker nodes. The way in which cluster resources are deployed may vary according to the specific Kubernetes distribution or service adopted, and the criticality is that administrators do not have visibility into "**cluster-related microservices**". To successfully escalate privilege, an attacker would need to leverage PODs Service Accounts or DaemonSet permissions to grant what is described in *Table 8.15, Attack Vectors*:

Attack	Description
Change Authentication	Change identity
Change Authorization	Change permissions (user impersonation)
Tokens	Retrieve or issue a service account token
Remote Code Execution (RCE)	Execute code on PODs
Steal PODs	Move PODs from one node to another

***Table 8.15**: Attack Vectors*

https://t.me/learningnets

Interestingly, Kubernetes clusters as service, including the most famous public cloud providers, are much more exposed to this threat than on-premises distribution because in order to connect with the cluster and make it available to the user, they need to provide a sort of interface that can manage the nodes. Refer to the following figure:

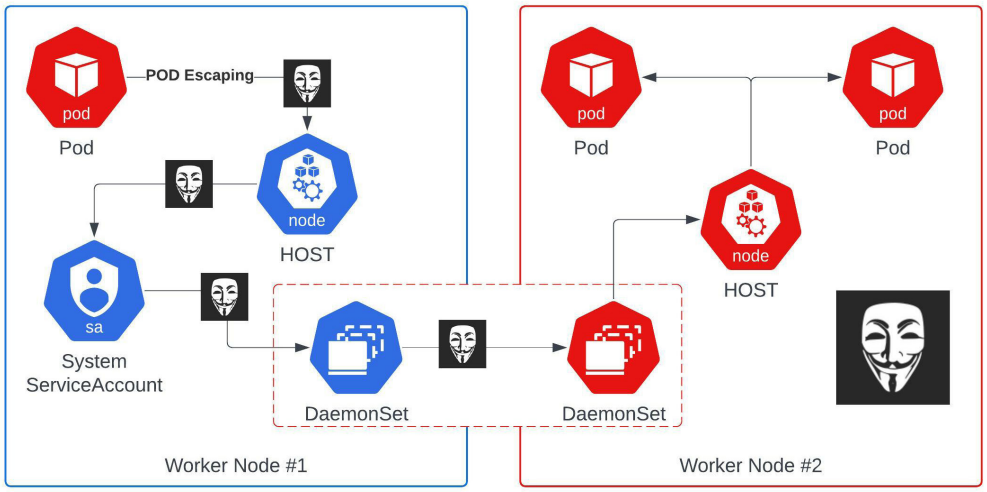

***Figure 8.6**: POD Escaping Privilege Escalation*

During the Blackhat USA 2022, researchers at the Palo Alto Networks were able to demonstrate the escalation technique illustrated in *Figure 8.6, POD Escaping Privilege Escalation*, and named the threat as the "**Trampoline POD**".

Assess and verify

Applying security to Kubernetes is an evolving journey. There are many things to consider, and keeping track of all the aspects involved is challenging. In *Chapter 7, Kubernetes Hardening*, we introduced **Mitre ATT&CK Kubernetes Matrix**, a framework aiming to provide guidelines to help organizations understand the attack

https://t.me/learningnets

surface of the Kubernetes environment in depth. The matrix highlights strategies and techniques attackers may use to target the Kubernetes clusters, as illustrated in *Figure 8.7, Mitre ATT&CK Kubernetes Matrix*:

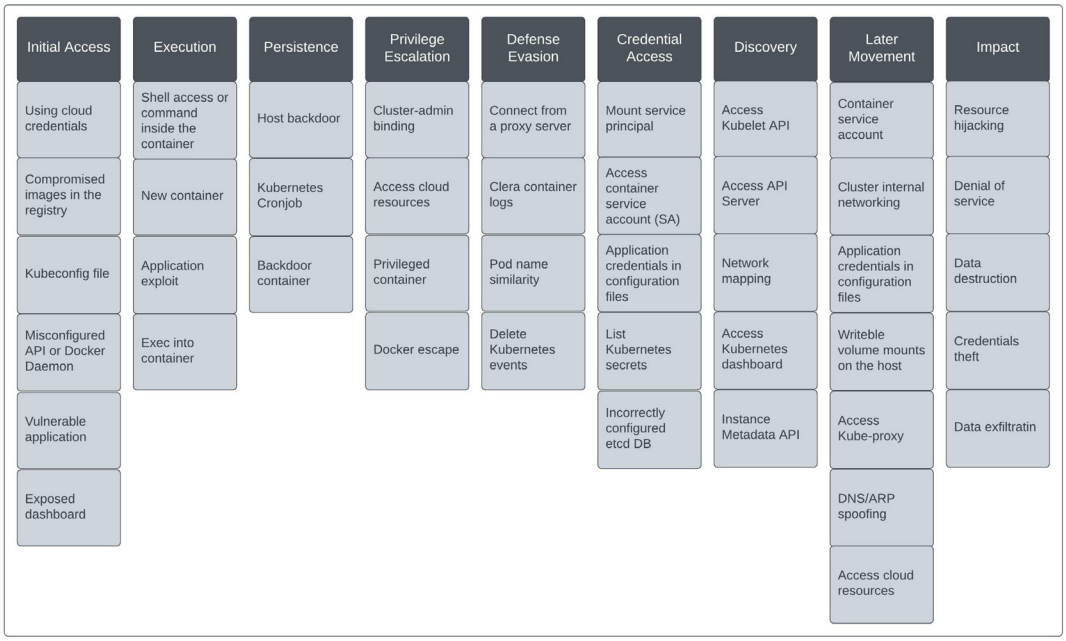

***Figure 8.7**: Mitre ATT&CK Kubernetes Matrix*

Despite the completeness of such framework and its very useful approach to create a security baseline in any Kubernetes environment, sometimes it is hard to find a logic way to start applying security best practices to such complex systems. *Table 8.16, Kubernetes Security List*, provides a quick summary of guidance to consider when aiming to achieve a better cluster security posture. This list is not meant to be complete; it just describes a baseline that can be quickly evolved and integrated with other tools and methodologies.

https://t.me/learningnets

Type	Description
Authentication and authorization	• The **system:masters** group has no more use after the initialization of the cluster. • Root certificate is shielded. • Certificates are managed. • The control plane, kube-controller-manager has **--use-service-account-credentials** enabled. • RBAC best practices are applied. • A procedure for periodic access review is in place.
Network security	• Network policies are applied. • **CNI** plugins are implementing network policies. • Encryption in transit is applied within the cluster. • The control plane components are exposed only internally. • PODs and containers' reverse access is filtered. • Load Balancer and external Ips utilization are restricted.
POD Security	• **RBAC** permissions are in place. • POD Security is applied. • Hardware resources access is controlled and limited. • SELinux, AppArmor and Seccomp are enabled.
Secrets	• Encryption at rest is configured. • Service accounts credentials are not available to PODs. • **ConfigMaps** are denied access to sensitive information. • An external secrets management tool is available.
Images	• Reduce the content of the image to the strict necessary to run the application. • Non-root user approach is adopted. • The tag **latest** is avoided. • Reference to the image is made by **sha256** digest. • Images are regularly scanned during the software development life cycle.
Admission Controller	• The list of admissions controllers is reviewed regularly. • The mutating admission is disabled. • The PODSecurity controller is enforced.

***Table 8.16**: Kubernetes Security List*

https://t.me/learningnets

Lastly, there is an interesting tool that applies the general security principles described in the previous table and can be used to assess the initial cluster security posture that is worth noting: **Kubeaudit**. Kubeaudit is a **Shopify** tool used to address various security concerns; it can run as a **kubectl** plugin, as a standalone Docker container, or simply as a command line tool installed via **brew**. The methodology applied to audit the cluster leverages the "auditors", a set of security controls that inspect the system to identify misconfigurations. The **auditors** can be executed together or individually, depending on the target of the auditing. These are briefly described in *Table 8.17, Kubeaudit Auditors*:

Type	Description
AppArmor	Verifies that containers running in PODs have the AppArmor flag enabled
ASAT	Automount Service Account Token (ASAT) auditor inspect containers that have the **serviceAccount** flagged as deprecated, and for the default **serviceAccount** mounted
Capabilities	Audit containers that do not add or drop the recommended capabilities
Deprecated APIs	Inspect the clusters for Kubernetes resources using a deprecated version of the API system
HostNS	Inspect the clusters for containers with HostPID, HostNetwork or HostIPC enabled
Image	Looks out for containers using a wrong version of the desired image or images without tags
Limits	Looks out for containers exceeding the required CPU or RAM limits or that do not specify those limits
Mounts	Looks out for containers mounting sensitive host path, such as **/proc**, **/etc**, **/root** and similar
Netpols	Looks out for namespaces that are not denying network traffic by default as per network policies specifications
Non-root	Looks out for containers running as root
Privesc	Looks out for containers allowed to use privilege escalation features
Privileged	Looks out for containers running with privileged permissions
rootfs	Looks out for containers mounting writable filesystems
Seccomp	Looks out for containers running in PODs without enabling Seccomp

***Table 8.17**: Kubeaudit Auditors*

https://t.me/learningnets

Kubeaudit runs in **manifest** mode, **local** mode, or **cluster** mode. The manifest mode supports the **autofix** feature that aims to correct the misbehaving resources. With it being an offline fix, the resources will be fixed only when the manifest file is applied again. The **cluster** mode can be executed running kubeaudit as a Docker image either with or without RBAC enabled. Of course, enabling RBAC is recommended. A brief description of the three modes is provided in *Table 8.18, Kubeaudit Modes*:

Description	Description
Manifest	When a manifest file is provided, kubeaudit scans the manifest file only.
Local	It leverages the kubeconfig file to connect to the cluster.
Cluster	In this mode, the whole cluster will be audited.

***Table 8.17**: Kubeaudit Modes*

Kubeaudit supports three types of results: error, warning, and info. These are also considered the three levels of severity the tool can produce when auditing a system.

Conclusion

In this chapter, we discussed the various security aspects of Kubernetes from the orchestration standpoint. This journey started from the authentication and authorization methodologies and how to secure them, and then we discussed the risks associated with the API server, the main differences between RBAC and ABAC, which one to choose and why.

We then moved on to understand what admission controllers are and why we need them, and we also looked at which aspect of the security landscape they cover within the cluster. Moving on, we got an overview of how Kubernetes manages secrets and how to protect them. Cluster isolation is another important part of establishing a security baseline for the Kubernetes cluster, and we complemented this argument with the audit and logging capabilities.

We also expanded on the concept of POD escaping discussed in the previous chapter, and looked at how it could escalate to compromise the whole cluster. Lastly, we focused on security guidelines and tooling to assess, audit and verify the security posture of the cluster.

In the next chapter, we will learn about Kubernetes Governance, and the various security aspects of adopting policies and procedure, and we will also go through tooling.

https://t.me/learningnets

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

https://t.me/learningnets

Chapter 9
Kubernetes Governance

Introduction

In information security, **governance** is the capability of a process to assess and manage the risks associated with the adoption of information technology solutions. Information security governance is, therefore, a set of policies, processes, and procedures by which an organization controls and coordinates the security requirements and activities that define the risk appetite.

Applying governance to Kubernetes is not a trivial task. The intrinsic complexity of the cluster does not facilitate the implementation of governance solutions. The Cloud Native Computing Foundation defines Kubernetes governance as a critical area of the cluster security aiming to achieve production-ready systems at scale. Governance should therefore be considered as a mix of different set of tools applied at different layers of the cluster, indeed there is no single governance tool that can achieve alone an optimal security posture. Security policies are sets of rules adopted to achieve the desired security requirements, often to fulfill compliance purposes. In Kubernetes, those are enforced to raise guardrails around the perimeter and reduce the attack surface by enabling control on the component of the cluster.

https://t.me/learningnets

Structure

In this chapter, we will discuss the following topics:

- Policy engines
- Admission controller threat model
- Network policies
- Resources management
- Security policies
- Limits and limitation

Objectives

This chapter provides an insight into how to apply governance by means of security policies that can be enforced to control security in the Kubernetes system. The topics of this chapter are to be considered as complementary arguments to the previous two chapters, where hardening and security orchestration were deemed to define different strategies to achieve a better security posture of the cluster.

Policy engines

In *Chapter 4, Securing Container Images and Registries*, we briefly introduced the **Open Policy Agent (OPA)** in the context of *Scanning and verifying images*. **OPA** is the most widely adopted Kubernetes policy engine, implemented as an additional layer of security through the admission controller mechanism discussed in the previous chapter. There are several use cases that can be mentioned, but to travel through the topics we have touched in this book so far, the following are the most relevant examples:

- All the container images must be deployed from a secure registry.
- All PODs must specify resource quota.
- The production namespaces must have RBAC enabled.

https://t.me/learningnets

Open Policy Engine was not created with Kubernetes in mind; indeed, the policy engine is an independent software created by Styra, with the mission to solve the cloud native authorization evolution, a system that has reached the Cloud Native Computing Foundation **graduated** maturity level in the early 2021. As such, OPA has a steep learning curve due to the adoption of **Rego**, a high-level declarative query language used to compose policies. Nevertheless, OPA can be integrated into Kubernetes as an admission controller through a mechanism called **Gatekeeper**. Gatekeeper is essentially the integration system between OPA and Kubernetes.

At the time of writing this chapter, Gatekeeper is at version 3.12. This version integrates the **OPA Constraint Framework** that fully supports Kubernetes CRD-based policies. **CRD** stands for Custom Resource Definition, which is not a Kubernetes resource but a custom resource added to the cluster by creating a custom API object. In this case, the imported policies generated by OPA Constraint Framework become Kubernetes objects or resources once admitted to the cluster through the admission controller. The OPA Constraint Framework, built with **kubebuilder**, allows declarative configured policy, overcoming the facto the OPA language limitation imposed by the Rego language, providing validating admission and mutating admission control, along with audit capabilities. A constraint is a declarative code enforcing a given system to meet a specific set of security requirements; consider the following example:

```
1. apiVersion: constraints.gatekeeper.sh/v1beta1
2. kind: RequiredLabel
3. metadata:
4.   name: require-production-label
5. spec:
6.   match:
7.     namespace: ["my-namespace"]
8.   parameters:
9.     labels: ["production"]
```

The objects produced by the OPA Constraint Framework successively admitted to the Kubernetes cluster are, therefore, consumed by the cluster itself as custom

https://t.me/learningnets

resources and applied through the Gatekeeper to perform the governance tasks they have been created for. Refer to the following figure:

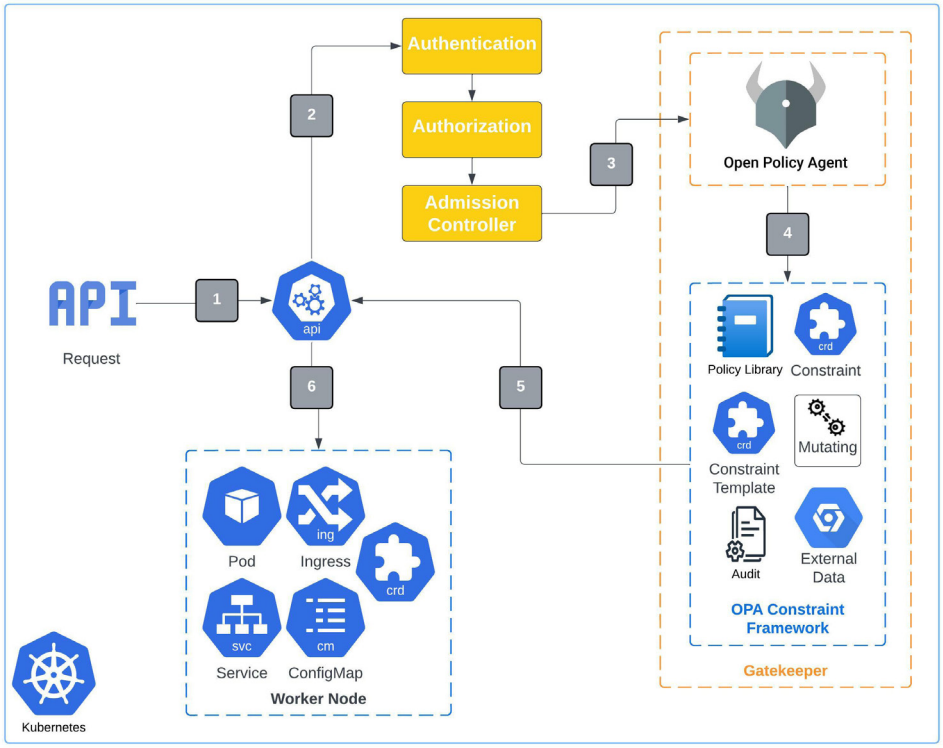

***Figure 9.1**: OPA Gatekeeper*

A workflow diagram to help visualize the Gatekeeper implementation is provided in *Figure 9.1, OPA Gatekeeper*:

1. An API request is received by the Kubernetes API Server.
2. The request is authenticated, and then the authorized is evaluated via admission controller.
3. The Admission Controller passes the request to the Open Policy Agent through the Gatekeeper.
4. The OPA Engine weights the request against the OPA Constraint Framework.
5. The OPA Engine responds to the **`kube-apiserver`**.
6. The Kubernetes API Server applies executes the request as a result of the OPA Engine evaluation.

https://t.me/learningnets

As explained in *Figure 8.4, Admission Controllers*, in the previous chapter, the admission controller system has two phases: the **mutating admission** and the **validating admission**. The **`kube-apiserver`** allows the admission controller to intercept requests before they become objects in Kubernetes; those requests are then routed to Gatekeeper, evaluated through OPA and redirected against the relevant resources once API requests are generated. The OPA Constraint Framework brings the features described in *Table 9.1, OPA Constraint Framework Features*:

Type	Description
Policy library	A set of ready-to-go policies for General cluster governance and Pod Security governance
Constraints	A YAML-coded declaration to enforce a specific set of requirements
Constraint template	A template to implement new constraints based on the input parameters and the Rego code to implement the constraint
Mutation	The mutation (or in Kubernetes terms, the mutating admission feature) that enables Gatekeeper to modify, change or update Kubernetes resources based on the constraints
Audit	Follows the resources lifecycle and periodically evaluate them against the constraints, eventually detecting misconfiguration and enforcing mutation
External data	Supports external data sources to enhance Gatekeeper capabilities to apply security requirements to the cluster, such as the following: • Ratify, an artifact security metadata verification engine • Container images signature verification engine

***Table 9.1**: OPA Constraint Framework Features*

The immediate advantage of Gatekeeper 3.12 is the **policy library**, which provides several constraints that can be utilized to improve security governance. The policy library general cluster governance group contains the policies described in *Table 9.2, General Policy Library*:

Type	Description
Allowed Repositories	Defines a list of allowed repositories from which container images can be pulled
Automount Service Account Token for Pod	Enables or disables any POD controlling the **automountServiceAccountToken** parameter
Block Endpoint Edit Default Role	Locks out editing the cluster endpoints; this vulnerability has been addressed in CVE-2021-25740

https://t.me/learningnets

Type	Description
Block NodePort	Blocks all services using type NodePort
Block Services with type LoadBalancer	Blocks all services using type LoadBalancer
Block updating Service Account	Blocks service accounts update on all the cluster resources external to the POD
Block wildcard ingress	Blocks wildcard (*) hostnames on the ingress system
Container Limits	Sets CPU and RAM limits to be restricted within the constraint limits
Container Requests	Sets CPU and RAM requests to be restricted within the constraint limits
Container Ratios	Sets the maximum allowed ratio between container limits and requests
Disallow Anonymous Access	Blocks Role and ClusterRole association with **system:anonymous** and **system:unauthenticated**
Disallow tags	Enforces container images to be tagged differently from the disallow list
External IPs	Enforces Service **externalIPs** to a pre-defined allowed list
HTTPS only	Enforces Ingress to HTTPS only
Image Digests	Enforces container images to adopt the digest
Pod Disruption Budget	Enforces POD disruptions in high availability deployment scenario
Required Annotations	Resources are requested to adopt annotations
Replica Limits	Limits resources replica to the value specified within ranges
Required Labels	Resources must contain the labels and values provided
Required Probes	PODs must have readiness and liveness probes set
Required Resources	Containers must have defined the allocated set of resources
Storage Class	Storage classes must be defined
Unique Ingress Host	Ingress rules must be unique
Unique Service Selector	Selectors on services must be unique within a namespace

***Table 9.2**: General Policy Library*

The POD Security topic was discussed initially *in Chapter 7, Kubernetes Hardening*, in the *POD Security* section, and subsequently in *Chapter 8, Kubernetes Orchestration Security*, in the *Admission Controller* section, where we acknowledged that **`PODSecurity`** replaces **`PODSecurityPolicy`**. The OPA Gatekeeper Library implements POD

https://t.me/learningnets

Security Policy as either constraints or constraints templates and brings into the picture a new aspect of Kubernetes security controls: the **POD Governance**. To clarify the relationship between security requirements, their initial application through the cluster POD Security Policy and policy engine system that governs them with the OPA Constraint Framework, refer to *Table 9.3, POD Governance*:

Security Control	POD Security Policy	Constraint
Privileged containers	Privileged	privileged-containers
Host namespaces	• hostPID • hostIPC	host-namespaces
Host networking and ports	• hostNetwork • hostPorts	host-network-ports
Volume types	volumes	volumes
Host filesystem	allowedHostPaths	host-filesystem
flex-volume drivers approved list	allowedFlexVolumes	flexvolume-drivers
read-only root file system	readOnlyRootFilesystem	read-only-root-filesystem
Container user and group IDs	• runAsUser • runAsGroup • supplementalGroups • fsgroup	users
root privileges escalation	• allowPrivilege Escalation • defaultAllow PrivilegeEscalation	allow-privilege-escalation
Linux capabilities	• defaultAddCapabilities • requiredDropCapabilities • allowedCapabilities	capabilities
Container SELinux	seLinux	seLinux
Container Proc mount types	allowedProcMountTypes	proc-mount
Container AppArmor profile	annotations	apparmor
Container seccomp profile	annotations	seccomp
Container sysctl profile	• forbiddenSysctls • allowedUnsafeSysctls	forbidden-sysctls

***Table 9.3**: POD Governance*

https://t.me/learningnets

Whenever a policy or constraint is missing and is required by the business needs, the constraint template helps in formulating a new constraint. It is a mix of fixed values and Rego language; the fixed values are as follows:

- **validation**: The parameters schema for the constraint
- **targets**: The resources target of the constraint
- **rego**: The rego expression that defines what the constraint wants to achieve
- **libs**: Library functions utilized by the rego package

An example of such implementation is given in the following constraint template code, assuming that the example policy aims to block ingress resources deployed in two or more namespaces from sharing the same hostname:

```
1. apiVersion: gatekeeper.sh/v1beta1
2. kind: ConstraintTemplate
3. metadata:
4.   name: blockingresshostsharingalternativenamespace
5. spec:
6.   crd:
7.     spec:
8.       names:
9.         kind: BlockIngressHostSharingAlternativeNamespace
10.      validation:
11.        openAPIV3Schema:
12.          properties:
13.            labels:
14.              type: array
15.              items: string
16.  targets:
17.    - target: admission.k8s.gatekeeper.sh
18.      rego: |
19.        package kubernetes.admission
20.        import data.kubernetes.ingresses
21.        deny[msg] {
22.            some alt_namespace, alt_ingresses
```

https://t.me/learningnets

```
            input.request.kind.kind == "Ingress"
            input.request.operation == ["CREATE", "UPDATE"]
            host := input.request.object.spec.rules[_].host
            ingress := ingresses[alt_namespace][alt_ingresses]
            alt_namespace != input.request.namespace
            ingress.spec.rules[_].host == host
             msg := sprintf("Host %q conflict! The ingress object belongs to a different namespace", [host, alt_namespace, alt_ingresses])
        }
```

Once the constraint template has been deployed into the cluster, the constraint can be deployed as customer resource definition; in our assumed case, the constraint example is provided in the following example code:

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: BlockIngressHostSharingAlternativeNamespace
metadata:
  name: ingress-conflict
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["ingress"]
  parameters:
    namespace: ["alt_namespace"]
```

In this context, an audit mechanism can be deployed to verify the application of the preceding constraint within the cluster. Continuing the ingress conflict example the deployment YAML file would look like the following audit code:

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: BlockIngressHostSharingAlternativeNamespace
metadata:
  name: ingress-conflict
spec:
  match:
```

https://t.me/learningnets

```
7.      kinds:
8.        - apiGroups: [""]
9.          kinds: ["ingress"]
10.   parameters:
11.     labels: ["alt_namespace"]
12. status:
13.   auditTimestamp: "2023-18-01T06:12:55Z"
14.   byPod:
15.   - enforced: true
16.     id: gatekeeper-controller-manager-0
17.   violations:
18.   - enforcementAction: deny
19.     kind: ingress
20.     message: 'Host conflict, the ingress object belongs to a differ-
    ent namespace'
21.     name: kube-system
```

Recently, a new player has joined the policy engines universe, **Kyverno**. Kyverno, in contrast with OPA, is designed to work with the tools that Kubernetes users have the habit of using, such as **`kubectl`**, **`kustomize`**, and eventually, **`git`**. The policy logic is a linear set of rules with a one-to-one relationship between the target resource and the expected action, as illustrated in *Figure 9.2, Kyverno*.

https://t.me/learningnets

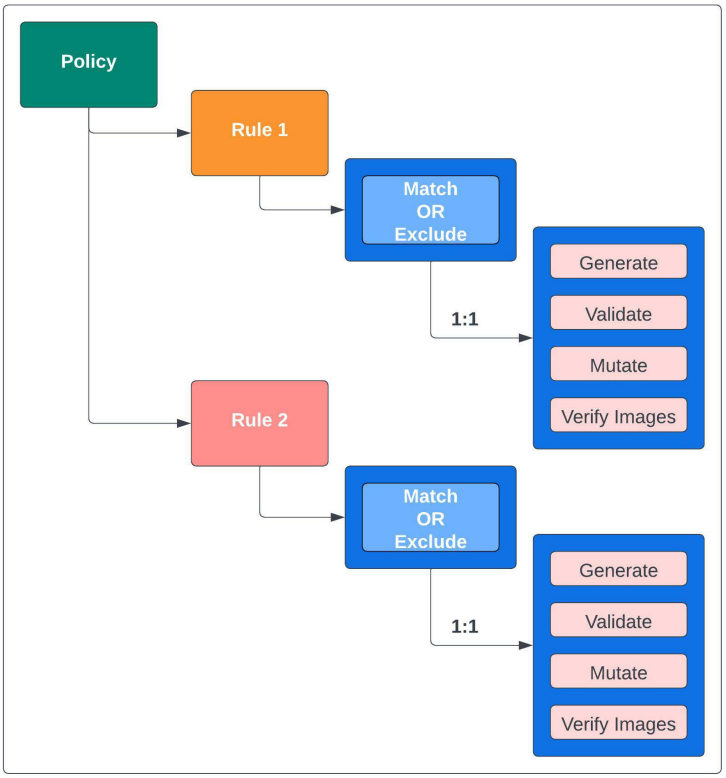

***Figure 9.2**: Kyverno*

Furthermore, once Kyverno has been deployed as admission controller, its policies are treated as Kubernetes native resources with no external language to learn, and there is no need to use Gatekeeper as implementation mechanism, and to define new rules via any customized template. Policies can be applied at either the cluster or the namespace layer, a simple idea of confining resources in the two mainstream logical perimeters of the Kubernetes cluster. Also, **Kyverno** is a policy engine created for cloud native solutions, so it provides great deployment flexibility and a rich set of pre-defined policy categories, a summary of which is provided in *Table 9.4, Kyverno Policies*:

https://t.me/learningnets

Type	Description

AWS	• IRSA required on node DaemonSet
	• AWS ELB Encryption required
	• AWS EKS
	  o Karpenter node eviction required
	  o Communication restriction via Network Policy
	  o Namespaces control Pod Security Admission via labels
	  o Container base images built from trusted sources
	  o Block ENV VARS containing secrets
	  o Limits and Requests check
	  o POD lifecycle management via liveliness and readiness
	  o PodDisruptionBudget
	  o Pod Security Standards profile restriction
	  o Container images verification via **Cosign**

Best Practices	• Check deprecated Kubernetes API Server versions • Block Container Runtime Interface socket mount • Ingress disallows empty host • Block "latest" tag • Block all Capabilities • Require workloads to declare Limits and Requests on cluster resources • POD life cycle management via liveliness and readiness • Enforce root filesystem in read-only mode • Enforce restriction on external IPs • Enforce restriction on image registries • Enforce restriction on defaultBackend for ingress deployment
Cert-Manager	• 100 days certificate renewal • dnsNames limited to a single name • Chain of trust established between a domain and the cluster certificate issuer
Consul	• TLS minimum version check

https://t.me/learningnets

Istio	• Sidecar Injection • mTLS enforcing • TLS on Host • AuthorizationPolicies
KubeVirt	• VirtualMachineInstance SSH service • InstanceType Enforcement
Linkerd	• Sidecar injection at the namespace level • Policy Annotation • Blocking Pod Injection
Multi-Tenancy	• Network Policy • DNS • Quota • StorageClass • RoleBinding
Nginx	• Block custom snippets to retrieve secrets from the cluster to address CVE-2021-25742 • Ingress restriction on annotation values for CVE-2021-25746 • Ingress path mitigation on CVE-2021-25745

POD Security	• Baseline PSS (POD Security Standard) Policy
	• Restricted PSS (POD Security Standard) Policy
	  ○ Container-Level Control
	  ○ Spec and Container-Level Control
	• POD Security Admission

Supply Chain	• Enforce vulnerability scan on the image • Implement SBOM verification • Addresses Apache Commons Text library as per CVE-2022-42889
Tekton	• Block TaskRun • Check TaskRun for vulnerabilities • PipelineRun requires namespace to be declared

***Table 9.4**: Kyverno Policies*

https://t.me/learningnets

Interestingly, Kyverno proposes security best practices to deploy the Kyverno policy engine inside the Kubernetes cluster with the maximum level of security posture to preserve the integrity and security of the cluster, as illustrated in *Table 9.5, Kyverno Security Best Practices*:

Type	Description
POD Security	POD Security Standard restricted profile
RBAC	Kyverno RBAC Roles and RoleBindings configuration to set the appropriate level of required permissions; specific customization can be provided via the **kyverno:view** and the **kyverno:generate** roles
Networking	Encrypted by default, network communication should be also restricted; a set of communication port is provided for fine-tuning
Webhooks	Both mutating and validating configurations are created
Recommendation	Minimum set of recommended policies are POD Security Standard and Best Practices

***Table 9.5**: Kyverno Security Best Practices*

For example, the following code illustrates how to enforce the AWS **Elastic Kubernetes Service** (**EKS**) node DaemonSet to use **IAM Role for Service Account** (**IRSA**):

```
1. apiVersion: kyverno.io/v1
2. kind: ClusterPolicy
3. metadata:
4.   name: enforce-aws-node-irsa
5. spec:
6.   validationFailureAction: enforce
7.   background: true
8.   rules:
9.   - name: validate-aws-node-daemonset-irsa
10.    match:
11.      any:
12.      - resources:
13.          kinds:
14.          - DaemonSet
15.          names:
```

https://t.me/learningnets

```
16.          - aws-node
17.          namespaces:
18.          - kube-system
19.    validate:
20.      message: "Enforce aws-node daemonset to use IRSA."
21.      pattern:
22.        spec:
23.          serviceAccountName: "!aws-node"
```

Due to the very flexible nature of this policy engine, and thanks to the availability of the **kyverno** command-line tool, the engine can be used in DevSecOps scenarios like **Continuous Integration and Continuous Deployment** (**CI/CD**) pipelines, in combination with version control system such as **git**. This enables a high level of security posture plus governance within the secure software development life cycle. An interesting part of this policy engine is also the **report** and **monitoring** capabilities. The report can be requested via command line, the result can be filtered per policy or per cluster, and the output of the report will provide the information in the format mentioned in *Table 9.6, Kyverno Report*:

Type	Description
pass	The count of resources to which the policy has been applied successfully
fail	The count of resources to which the policy has been applied unsuccessfully
warn	The count of resources to which the policy has been applied and returned a warning
error	The count of resources to which the policy has been applied and returned an error
skip	The count of resources for which the policy has been skipped

***Table 9.6**: Kyverno Report*

The command-line report tool can be invoked via **`kubectl`**, as shown in the following example:

```
1. $ kubectl get policyreport -A
2. NAMESPACE     NAME
   PASS   FAIL   WARN   ERROR   SKIP   AGE
3. kube-system   forbid-root-containers
   4      1      0      1       0      16s
4. kyverno       forbid-root-containers
   12     0      1      0       0      51s
5. default       forbid-root-containers
   1      1      0      0       1      5m
```

https://t.me/learningnets

Although doing manual investigation is useful, it can be cumbersome in complex environments. So, Kyverno implements a ready-to-use Grafana dashboard that can display out-of-the-box metrics like the following:

- The total count of active policies and rules
- The total count of rules execution
- Any latency associated with the rule processing
- The history of cluster policies ingested via Kyverno and their status over time

Lastly, Kyverno supports **Customer Resource Definition** (CRD) that can be used with integrated development environment tools such VS Code.

Admission controller threat model

In *Chapter 8, Kubernetes Orchestration Security*, in the *Admission Controller* section, we discussed the potential and use cases for the admission controller mechanism in Kubernetes. In this chapter, we discussed how policy engines like OPA and Kyverno can leverage the admission controller to be plugged into the cluster for enhancing security governance, and how they can integrate external sources to expand their policy system baseline. Similar to the `kube-apiserver`, the admission controller is one of the key elements of a Kubernetes cluster. Due to its peculiarity the admission controller is at the same time source of concerns because it enables external systems to be plugged into the cluster and let those systems to govern the cluster behavior. The admission controller could be leveraged by an attacker to bypass the natural cluster defense mechanisms. The Kubernetes **Special Interest Group** (**SIG**) has developed a reference threat model to define the threats to be considered when designing or implementing plugins for the admission controller. The **Admission Controller Threat Model** can be used by the following groups:

- Authors who are developing or working on admission controller implementation
- Companies deploying an admission controller add-on as part of their security requirements
- Auditors who need to assess the security of newly deployed or existing Kubernetes clusters that use the admission controller system
- Developers who are responsible for implementing new features or updating the existing ones

This threat model considers that the admission controller can be implemented as described in *Table 9.7, Admission Controller Implementation*:

https://t.me/learningnets

Type	Description
Workload	This is a common installation methodology for Kubernetes admission controllers; refer to *Table 8.2, Admission Controllers List*.
Webhooks	This is a common installation methodology for third-party admission controllers to connect to the kube-apiserver.

***Table 9.7**: Admission Controller Implementation*

The implementation strategy is not the only mechanism that can determine a threat; there are two other characteristics to consider, as per *Table 9.8, Admission Controller Features*:

Type	Description
Rules	The admission controller implementation provides a set of rules that can block or allow container workloads into the Kubernetes cluster, aiming to add a layer of security controls.
Mutation	The admission controller implementation provides the capability to mutate container workloads and therefore, to change or modify Kubernetes resources configurations to comply with policies deployed into the cluster.

***Table 9.8**: Admission Controller Features*

When considering the mentioned assumptions as non-true, the various scenarios resulting from this threat model analysis can be still help provide threat mitigation for the purpose of elevating the overall security posture of the Kubernetes cluster. The admission controller threat model today has defined 18 threat scenarios that can potentially impact the admission controller system, as described in *Table 9.9, Admission Controller Threat List:*

ID	Threat	Impact
T1	Traffic floods webhook	Spin up a malicious workload that would usually be blocked
T2	Workload causes timeouts	Spin up a malicious workload that would usually be blocked
T3	Misconfiguration exploit	Deploy a malicious workload that would usually be blocked or mutated
T4	Modify or delete webhook	Bypassing the admission controller
T5	Webhook Access credentials	Denial of Service
T6	Cluster root credentials	Whole cluster compromised

https://t.me/learningnets

ID	Threat	Impact
T7	Network sniffing	Transmitted information exposed
T8	Man in the middle	Intercepting requests and responses
T9	Spoofing	Forging responses
T10	Rule abuse	Host node compromise
T11	Namespace exposure	Cluster node compromise
T12	Missing match	Cluster node compromise
T13	Bad string matching	Spin up a malicious workload that in normal conditions would be blocked
T14	No rule features	New features or old features in old cluster may not have been assessed, so the admission controller could be bypassed
T15	Privileged container	The admission controller would be forbidden from enforcing policies
T16	Privileged hostPath mount	The admission controller POD disruption
T17	SSH access	Disable or change the admission controller operations
T18	Policy compromise	Information leak due to potential data transmission to external systems

***Table 9.9**: Admission Controller Threat List*

The Kubernetes **Special Interest Group** (**SIG**) has provided 10 mitigation techniques to address the threats described in the previous table. These mitigations are described in *Table 9.10, Admission Controller Mitigation Factors*:

ID	Mitigation
M1	Apply and restrict Role-Based Access Control (RBAC) permissions, including the following: • ValidatingWebhookConfigurations • MutatingWebhookConfigurations • ClusterRoleBindings • ClusterRoles • Workloads • Services
M2	Close Webhook that fails; be mindful of legit webhook failing due to access restrictions or network communication issues; it could prevent legit container workload from being deployed into the cluster

https://t.me/learningnets

M3	Request Webhook connection authentication through exchange of certificates that can be validated by a trusted Certification Authority (CA)
M4	Apply encryption in transit between webhooks and the API server, using at least TLS version 1.2
M5	Webhook mutual authentication should be required; the API server should authenticate via TLS the webhook and vice versa
M6	Regularly audit and test the rules endorsed by the admission controller ensuring that the policy is effectively enforcing the security requirements defined
M7	Privileged containers should be restricted by the admission controller only to the system services
M8	Regularly audit Webhook configurations to verify the admission controller overall security
M9	Admission controllers implementing external systems access should restrict external network traffic by configuring network policies
M10	Utilize a dedicated Webhook for each individual admission controller minimizing the attack surface of the cluster

***Table 9.10**: Admission Controller Mitigation Factors*

Not all the mitigation techniques described in the preceding table can be utilized for all the aspects of this threat model.

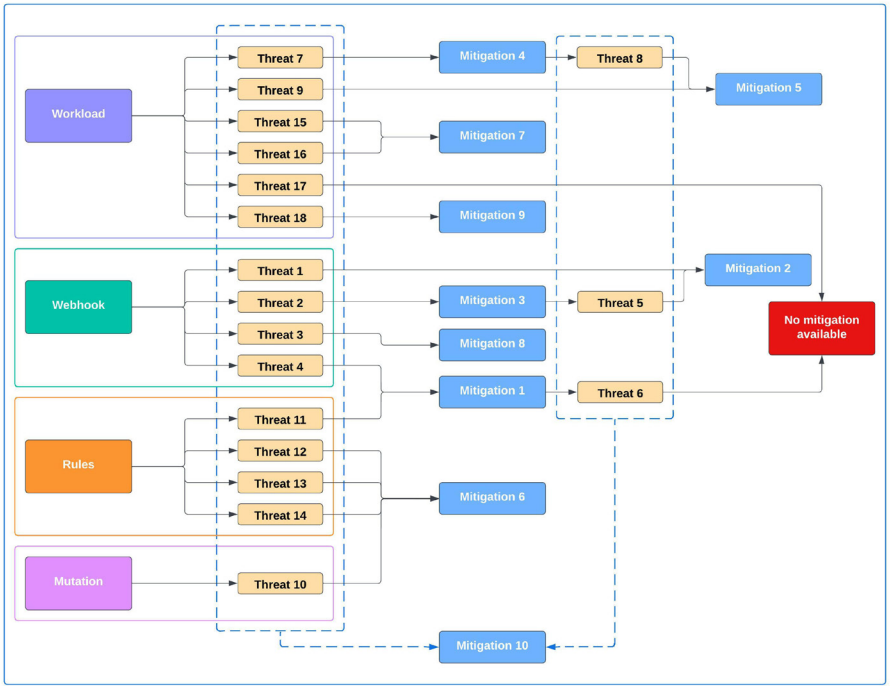

***Figure 9.3**: Admission Controller Threat Model Mapping*

https://t.me/learningnets

The preceding figure is divided into the four initial domains described in *Table 9.7, Admission Controller Implementation*, and *Table 9.8, Admission Controller Features*. These four domains have been associated to the threat list described in *Table 9.9, Admission Controller Threat List*, which has been mapped to the mitigation methods described in *Figure 9.3, Admission Controller Threat Model Mapping*. As readers may note, different threats can have the same mitigation technique, like the threats with IDs 12, 13, 14 and 10, and some threats may not have a mitigation at all, like threats with IDs 6 and 17. There are also threats that depend on other threats, and they may require one or more mitigation methodology to fully comply with the governance of the cluster. The mitigation ID 10 is broadly used to reduce the attack surface of the cluster as a whole.

Network policies

All the policy engines and rules we have discussed so far are somehow affecting Kubernetes resources, individual components of the cluster or group of resources. It is possible to control the network traffic within the Kubernetes cluster at the IP address level, at layer 3, Network Layer or layer 4, Transport Layer, of the **Open System Interconnection** (OSI) model. This can be achieved through **Kubernetes Network Policies**. The Network Policies system governs how a POD communicates with other Kubernetes network resources like endpoints, ingresses, and services. The network resources that a POD is allowed to communicate with are described as a combination of the following:

- Other PODs
- Allowed namespaces
- CIDRs ranges

In *Chapter 7, Kubernetes Hardening*, in the *Securing Container Runtime Interface* section, we briefly discussed the **Container Network Interface** (**CNI**) as part of the components that contribute to the creation of a POD, in the context of securing the Container Runtime Interface. A container network interface is a network plugin used by Kubernetes to deploy network policies into the cluster. In essence, this is a prerequisite for the network policies to work as expected.

Note: CNI is fully supported by Kubernetes version 1.26 and above; for versions prior to 1.24, it can be managed via kubectl. The recommended container network interface version is 1.0.0.

Every network device has two communication directions: an **inbound** or ingress that accepts or denies the network traffic coming from outside, and an **outbound** or egress that pushes the traffic on the device outside. By default, a POD is a non-isolate network system either in egress or ingress communications, which means it

https://t.me/learningnets

is accepting network traffic from everywhere and is also generating network traffic toward everywhere. When a POD achieves isolation via egress and ingress network policies, the only connections allowed are the ones explicitly allowed by the ingress or egress lists of the mentioned network po1licies. Note that network policies do not act as a firewall; even if they tend to achieve POD security, their rule mechanism behaves slightly differently because they do not conflict; and there is no order of rule evaluation. Rules are additive, meaning communication is the combination of all the applicable rules enforced by the policies. The following code provides an example of network policy to be applied for the purpose of isolating PODs labelled as **`role=mysql`** in the "**`my-namespace`**" namespace:

```
1. apiVersion: networking.k8s.io/v1
2. kind: NetworkPolicy
3. metadata:
4.   name: network-policy-01
5.   namespace: my-namespace
6. spec:
7.   podSelector:
8.     matchLabels:
9.       role: mysql
10.  policyTypes:
11.    - Ingress
12.    - Egress
13.  ingress:
14.    - from:
15.        - ipBlock:
16.            cidr: 172.11.0.0/16
17.            except:
18.              - 172.11.2.0/24
19.        - namespaceSelector:
20.            matchLabels:
21.              project: webserver
22.        - podSelector:
23.            matchLabels:
24.              role: http
```

https://t.me/learningnets

```
25.       ports:
26.         - protocol: TCP
27.           port: 3306
28.   egress:
29.     - to:
30.         - ipBlock:
31.             cidr: 10.0.1.0/24
32.       ports:
33.         - protocol: TCP
34.           port: 6121
35.           endport: 6221
```

The key elements of the preceding code are described in *Table 9.11, Network Policy Elements*:

Type	Description
podSelector	This network policy will be applied to all the PODs with label **role:mysql** created and running in the namespace **my-namespace**.
policyTypes	A network policy could define either ingress or egress, but this policy declares both.
ingress rules	Allow ingress connections for PODs meeting the following criteria: • PODs are labelled as **role:http** • PODs are running in the namespace labelled as **project:webserver** • PODs have IP address within the CIDR block 172.11.0.0/16 , except 172.11.2.0/24 • PODs requesting connection to port TCP 3306
egress rules	Allow egress connection for PODs meeting the following criteria : • PODs are labelled as role:mysql • PODs with addresses within the CIDR block 10.0.1.0/24 • PODs requesting connection from port TCP 6121 to TCP 6221

***Table 9.11**: Network Policy Elements*

When defining rules in network policies, be mindful of using the elements **`to`** and **`from`**; for example, the following code is extracted from the previous code from line 19 to 24, and it means "allow connection from any POD in namespace labelled as **`project=webserver`**" **or** "allow connection from all the PODs in **any** namespace

https://t.me/learningnets

labelled as **`role=http`**", resulting in two different applications of the element **`from`**:

```
1.   ...
2.   ingress:
3.   - from:
4.     - namespaceSelector:
5.         matchLabels:
6.           project: webserver
7.     - podSelector:
8.         matchLabels:
9.           role: http
10.  ...
```

The following code at line 7 has no dash suffix on the **`podSelector`** element, meaning "allowing connection from any POD labelled with **`role=http`** running in the namespace labelled with **`project=server`**":

```
1.   ...
2.   ingress:
3.   - from:
4.     - namespaceSelector:
5.         matchLabels:
6.           project: webserver
7.       podSelector:
8.         matchLabels:
9.           role: http
10.  ...
```

There is only one dash difference between the two code examples, but the result of the policy may be quite different and may lead to unexpected results. It is recommended to verify that Kubernetes has interpreted the policy with **`kubectl describe my-policy-name`**. There are cases where ipBlock can't be applied or it is simply not effective, such as IP rewriting due to LoadBalancer, or in cloud environments where the network traffic is rerouted according to the network service implemented or managed. In a scenario where namespaces have no policies at all, all the traffic in any direction is allowed by default. Default behavior can be changed to the following:

https://t.me/learningnets

- Deny ingress traffic by default
- Deny egress traffic by default
- Deny ingress and egress traffic by default

Based on what we described before, to achieve this, the **`podSelector`** should be declared as follows, using the appropriate **`policyTypes`**:

```
1. ...
2. metadata:
3.   name: default-deny-all
4. spec:
5.   podSelector: {}
6.   policyTypes:
7.   - Egress
8.   - Ingress
```

As a reminder, and to set each piece of the puzzle in the right place, network policies are part of the cluster isolation topic discussed in *Chapter 8, Kubernetes Orchestration Security*, in the *Cluster Isolation* section.

Resources management

This section of the chapter is not related to how to specify CPU, RAM and storage **requests** and **limits**. Resource quotas are considered policies, their application is mostly related hardware resource optimization. There are, of course, security requirements on resource quotas, especially in relation to the cluster hardware capacity and POD allocation. For instance, one of the common scenarios is having PODs running in different namespaces being allocated on the same node.

This argument does not aim to discourage the application of resource requests or limits of containers or PODs; it is a good practice to define boundaries around the hardware resources that those Kubernetes components can obtain. Rather, it is an invite to move one step further and considering apply the same concepts to namespace and nodes. The combination of resource requests and limits at the container level is also what qualifies a POD to obtain a **Quality of Service** (**QoS**) class within the namespace. Refer to the following figure:

https://t.me/learningnets

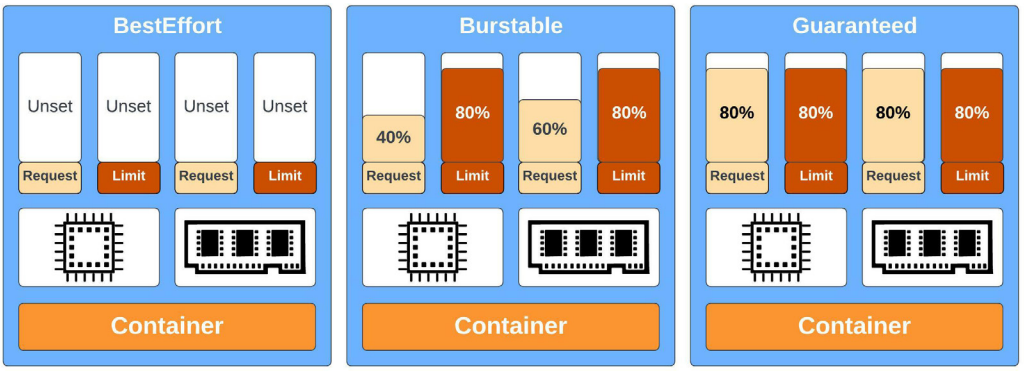

***Figure 9.4**: QoS Classes*

As illustrated in *Figure 9.4, QoS Classes*, there are three Quality of Services classes:

- The QoS BestEffort container has neither requests nor limits set.
- The QoS Burstable container has both CPU and memory requests values lower than their limits.
- The QoS Guaranteed container has both CPU and memory requests values equal to their limits.

The QoS POD classes are significant to the cluster because they represent the metrics used by Kubernetes to apply decisions on the resources' allocation. Resources management in the cluster is enabled using the **`ResourceQuota`** admission controller (refer to *Table 8.6, Admission Controllers List*, in the previous chapter for an overview of the controllers available in Kubernetes), and it is enforced in any namespace where **`ResourceQuota`** is declared. The resource types available to namespaces are the same for the containers or the PODs, and they are listed in *Table 9.12, Namespace ResourceQuota*:

Type	Description
cpu	It is an abbreviation for **requests.cpu**.
memory	It is an abbreviation for **requests.memory**.
requests.cpu	Total sum of CPU request by all PODs running in the namespace cannot exceed this value.
limits.cpu	Total sum of CPU limit by all PODs running in the namespace cannot exceed this value.
requests. memory	Total sum of memory request by all PODs running in the namespace cannot exceed this value.

https://t.me/learningnets

Type	Description
limits.memory	Total sum of memory limit by all PODs running in the namespace cannot exceed this value.
hugepages-<size>	Total number of huge pages in the namespace cannot exceed this value.

***Table 9.12**: Namespace ResourceQuota*

Similarly, it is possible to limit the storage resources in the namespace, as listed in *Table 9.13, Namespace Storage ResourceQuota*:

Type	Description
requests.storage	The total number of persistent volume requests in the namespace cannot exceed this value.
storage-class-name/ requests	The total number of storage requests within a specific storage class in the namespace cannot exceed this value at any given time.
persistentvolume-claims	The total number of persistent volumes that can exist in the namespace cannot exceed this value at any given time.
storage-class-name/ persistentvolume-claims	The total number of persistent volumes claims within a specific storage class in the namespace cannot exceed this value at any given time.
ephemeral-storage	It is an abbreviation for **requests.ephemeral-storage**.
requests.ephemer-al-storage	The total number of ephemeral storage requests within the namespace cannot exceed this value.
limits.ephemeral-stor-age	The total number of ephemeral storage limits within the namespace cannot exceed this value.

***Table 9.13**: Namespace Storage ResourceQuota*

Another interesting feature is to set a count quota per namespaced resource using the **count/something** suffix; for example, **count/secrets** will set a limit to the total number of secrets that can be created, along with **count/services**, **count/deployments**, **count/jobs**. To achieve full namespace isolation, from Kubernetes version 1.24 onward, it is possible to scope out a quota limit. The quota limit defines a set of namespaces that are allowed to enable POD affinity by declaring the `CrossNamespacePodAffinity` parameter. A quick reminder about the affinity property: it is the capability of a POD to attract or repel other PODs or Nodes. This is a strategic allocation of PODs in terms of resources or deployment affinity. This feature comes in useful for system segmentation considering both **affinity** and **anti-affinity** use cases. A simple example for POD affinity would be all the webserver PODs need to be deployed within the same namespace conversely the database PODs need to be deployed into a

https://t.me/learningnets

different namespace (affinity). The POD anti-affinity use case is slightly different, and should not be considered as the opposite logic of the POD Affinity use case. For instance, an example of anti-affinity is applicable to distributed systems or high availability applications where it is better not to have POD deployed on the same node or within the same namespace. From the security standpoint, having PODs crossing namespaces is a serious concern and is hard to handle, but by applying the **`CrossNamespacePodAffinity`** parameter, governance can be achieved by setting a hard limit of 0, as shown in the following example code:

```
1. apiVersion: v1
2. kind: ResourceQuota
3. metadata:
4.   name: disable-cross-namespace-affinity
5.   namespace: my-namespace
6. spec:
7.   hard:
8.     pods: "0"
9.   scopeSelector:
10.    matchExpressions:
11.    - scopeName: CrossNamespacePodAffinity
12.      operator: Exists
```

A **node** is a full operating host system that assumes the node function, which means not all the host resources are assigned to Kubernetes or the various components that constitute a cluster; some of the CPU and memory resources are assigned to the host itself. A node is then a combination of the following resources:

- Physical hardware resources
- Operating system resources
- Resources reserved for Kubelet
- Resources available to PODs

What Kubernetes manages on the host to govern the worker node is a set of three policies, as described in *Table 9.14, Node Resource Managers*:

https://t.me/learningnets

Policy	Description
CPU Manager	The kubelet system uses the **Completely Fair Scheduler** (**CFS**) to enforce CPU limits on PODs; refer to *Chapter 3, Container Stack Security*, the *Resource Limitation* section. Two policies can be applied: • The **none** policy is the default policy on the node, and it does nothing more than what the CFS can do, deferring all the affinity decision to the scheduler. • The **static** policy provides the **Guaranteed** POD class access to the exclusive CPU cycle, usually allocated to components like the container runtime or the **kubelet**, exposing the host to potential source of compromise.
Memory Manager	It enables the **hugepages** and memory allocation for PODs. PODs must have a QoS Guaranteed class and a **Non-Uniform Memory Access** (**NUMA**) yield affinity to the node. Two policies can be applied: • The **none** policy is the default policy on the node, and it does not intervene in memory allocation. • The **static** policy allows the **Guaranteed** POD class to be allocated on NUMA nodes, where the available memory can be reserved.
Device Manager	As the name suggests, the policies implemented by the device manager are utilized as device plugin framework for the purpose of elevating the hardware resources visibility up to the kubelet layer. Readers can see this as a sort of hypervisor, where hardware resources are virtualized. The list includes, but it is not limited to, **Field Programmable Gate Array** (**FPGAs)**, **Graphic Processor Units** (**GPUs**), and **Network Interface Cards** (**NICs**).

***Table 9.14**: Node Resource Managers*

Prior to Kubernetes version 1.18, the CPU and Device managers were making decisions in relation to resource allocation independently, with obvious repercussions for cluster stability and security. From that release onward, Kubernetes introduced **Topology Manager** as a feature gate to govern and orchestrate hardware resource allocation across the three domains described in the previous table. The Topology Manager is implemented on the node as a kubelet component and offers an interface for the other components, called Hint Providers, for the purpose of consuming information about the hardware resources on the node. The scope of application is dual: **Container Scope** and **POD Scope**. The container scope is the default choice, and it is equal to no policy. The Topology Manager will arbitrarily assign containers to nodes. When the POD scope is selected, the Topology Manager will allocate all the containers of a POD to a single node or to a shared set of nodes. A description of the topology options is given in *Table 9.15, Topology Manager Policies*:

https://t.me/learningnets

Policy	Description
none	No topology alignment is performed.
best-effort	The Topology Manager queries each Hint Provider to understand their resources status, and it syncs container assigned resources in any given POD with the resources available on the nodes. If affinity is not met, the POD is anyway admitted to the node.
restricted	The Topology Manager queries each Hint Provider to understand their resource status, and it syncs container assigned resources in any given POD with the resources available on the nodes; if affinity is not met, the POD is rejected, triggering a POD admission failure. From the security perspective, any requests or limits that do not meet the policy criteria are considered unsafe.
single-numa-node	The Topology Manager queries each Hint Provider to understand their resources status, and it syncs container assigned resources any given POD with the resources available onto a single node; but if affinity is not met, the POD is rejected, triggering a POD admission failure.

***Table 9.15**: Topology Manager Policies*

When configured with POD scope, the Topology Manager interprets the container requirements as the requirements of the entire POD. So, all the containers running inside the POD will have the same topology decision.

Security policies

In *Chapter 7, Kubernetes Hardening*, in the *POD Security* section, we described what POD Security Standards is and how it affects the security of the Kubernetes cluster. We also discussed how POD Security Standards is applied to the cluster by means of the *Admission Controller* in *Chapter 8, Kubernetes Orchestration Security*. The practical execution of POD Security Standards can be applied in three different ways, each way affecting a dedicated Kubernetes perimeter:

- At the cluster level
- At the namespace level
- At the POD level

At the cluster level, the most efficient way to apply **POD Security Standards** is to use the **POD Security Admission Controller**. In this configuration, when a POD is created everywhere in the cluster, the POD Security admission controller checks the configuration against the POD Security Standards. As we know, the POD Security Standards has three pre-defined policies as per *Table 7.12, POD Security Standards*: **privileged**, **baseline** and **restricted**. It is possible to verify the impact of changing policy using the **`--dry-run=server`** parameter.

https://t.me/learningnets

```
$ kubectl label --overwrite ns --dry-run=server --all \
pod-security.kubernetes.io/enforce=baseline
```

For example, the preceding command will return something like this:

```
namespace/default labelled
namespace/kube-public labelled
namespace/kube-node-lease labelled
Warning: existing pods in namespace "kube-sys-
tem" violate the new PodSecurity enforce level "baseline:latest"
Warning: etcd-my-cluster-control-
plane (and 3 other pods): host namespaces, hostPath volumes
Warning: kindnet-xyz72: non-
default capabilities, host namespaces, hostPath volumes
Warning: kube-proxy-
a4trg: host namespaces, hostPath volumes, privileged
namespace/kube-system labelled
namespace/local-path-storage labelled
```

The execution of the preceding command with the **restricted** policy would eventually create even more warnings. The policy can be used as a configuration file that the API server can consume upon cluster creation, as the following example code demonstrates:

```
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
- name: PodSecurity
  configuration:
    apiVersion: pod-security.admission.config.k8s.io/v1
    kind: PodSecurityConfiguration
    defaults:
      enforce: "restricted"
      enforce-version: "latest"
      audit: "restricted"
      audit-version: "latest"
```

https://t.me/learningnets

```
13.       warn: "restricted"
14.       warn-version: "latest"
```

Note: The restricted policy offers the highest level of security, but it can also compromise the availability of the system. It is recommended to test the application of such policy and eventually consider excluding the kube-system.

It is possible to verify the effectiveness of the policy creating a simple POD:

```
1.  apiVersion: v1
2.  kind: Pod
3.  metadata:
4.    name: mysql
5.  spec:
6.    containers:
7.      - image: mysql
8.        name: mysql
9.        ports:
10.         - containerPort: 3306
```

The output is alerting that the POD deployment is violating the **PODSecurity** restricted policy configuration by allowing POD with Privilege Escalation capability:

```
1.  Warning: would violate PodSecurity "restricted:latest": allowPrivi-
    legeEscalation != false (container "mysql" must set securityContext.
    allowPrivilegeEscalation=false), \
2.  unrestricted capabilities (contain-
    er "mysql" must set securityContext.capabilities.drop=["ALL"]), \
3.  runAsNon-
    Root != true (pod or container "mysql" must set securityContext.
    runAsNonRoot=true), \
4.  seccompProfile (pod or container "mysql" must set securityContext.
    seccompProfile.type to "RuntimeDefault" or "Localhost")
```

It is possible to apply the same principle at the **namespace** level:

```
1.  $ kubectl label –overwrite ns my-namespace \
2.     pod-security.kubernetes.io/warn=baseline \
```

https://t.me/learningnets

And it is also possible to create a configuration file similar to the one used for the cluster case that can be consumed by the API server; of course, this time the policy would apply to the namespace my-namespace only. When thinking about how to apply the policy at the **POD level**, two systems can be leveraged: **AppArmor** or **Seccomp**. For the purpose of this section, remember that the main difference between AppArmor and Seccomp is that the former applies security boundaries at the application level, while the latter restricts the calls to the kernel that a process is able to make. Assuming that both are enabled and that Kubernetes is at least version 1.25, we also need to ensure that the container runtime supports AppArmor and that the AppArmor profile is loaded. Refer to the following code:

```
1. container.apparmor.security.beta.kubernetes.io/my-
   container: <profile>
```

The preceding code shows how to enable AppArmor for the container **my-container**; the <profile> parameter must be one of the options listed in *Table 9.16, AppArmor Profiles*:

Eviction	Description
unconfined	The container will run with no AppArmor profile.
runtime/default	The container will run with the default AppArmor profile loaded into runtime.
localhost/profile	The container will run with the default AppArmor profile loaded into the host.

***Table 9.16**: AppArmor Profiles*

To use Seccomp, the kubelet system must have the feature gate **`SeccompDefault`** loaded; this should be present by default in most recent cluster versions, but the worker nodes and the host must also have the corresponding feature enabled. With both running the **`SeccompDefault`**, the kubelet will use the Seccomp **`RuntimeDefault`** profile rather than the **`Unconfined`** profile. The default profile sufficiently provides a strong set of security reequipments and in the meantime, preserves the functionality of the cluster. Seccomp is an efficient way to retain the containers syscall against the Kernel, but it also affects the workload and the interactions of the containers with the underlaying system, stopping them from working as they should. As such, it is possible to create custom Seccomp profile that can be used as audit mechanism. For example, the following code will create a Seccomp profile called **`inspect.json`**:

```
1. {
2.     "defaultAction": "SCMP_ACT_LOG",
3.     "architectures": [
4.         "SCMP_ARCH_X86_64",
```

https://t.me/learningnets

```
        "SCMP_ARCH_X86",
        "SCMP_ARCH_X32"
    ]
}
```

Then, the following code will create a POD that leverages the custom Seccomp profile:

```
apiVersion: v1
kind: Pod
metadata:
  name: seccomp-audit
  labels:
    app: seccomp-audit
spec:
  securityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: profiles/inspect.json
  containers:
  - name: my-audited-container
    image: hashicorp/http-echo:0.2.3
    args:
    - "-text=syscalls executed"
    securityContext:
      allowPrivilegeEscalation: false
```

The **`http-echo`** is a mini webserver created by Hashicorp for demo purposes, and it usually runs unrestricted. By login into the container and checking the syslog file in **`/var/log/syslog`**, the file would eventually expose syscalls made by the container, as per the following log example (note that the audit log has been edited to comply with publishing guidelines):

```
Jan  22 21:11:22 my-vm kernel: audit: pid=29064 exe="/http-
echo" syscall=43
Jan  22 21:11:22 my-vm kernel: audit: pid=29064 exe="/http-
```

https://t.me/learningnets

```
    echo" syscall=64
3. Jan    22   21:11:22   my-vm   kernel:   audit:   pid=29064   exe="/http-
    echo" syscall=102
4. Jan    22   21:11:22   my-vm   kernel:   audit:   pid=29064   exe="/http-
    echo" syscall=188
5. Jan    22   21:11:22   my-vm   kernel:   audit:   pid=29064   exe="/http-
    echo" syscall=1
```

The **syscall=** entry in the syslog file illustrates the system calls made by the container. Reversing this approach, it is possible to create a test POD that applies a custom policy, with only the system calls that the cluster admin allows. The following code can be used as baseline to fine-tune the policy:

```
1. {
2.     "defaultAction": "SCMP_ACT_ERRNO",
3.     "architectures": [
4.         "SCMP_ARCH_X32",
5.         "SCMP_ARCH_X86",
6.         "SCMP_ARCH_X86_64"
7.     ],
8.     "syscalls": [
9.         {
10.            "names": [
11.                "accept4",
12.                "arch_prctl",
13.                "brk",
14.                "bind",
15.                "clone",
16.                "clock_gettime",
17.                "close",
18.                "connect",
19.                "dup2",
20.                "epoll_create1",
21.                "epoll_ctl",
```

https://t.me/learningnets

```
22.                 "epoll_wait",
23.                 "epoll_pwait",
24.                 "execve",
25.                 "exit",
26.                 "exit_group",
27.                 "fcntl",
28.                 "futex",
29.                 "getpid",
30.                 "getsockname",
31.                 "gettid",
32.                 "getuid",
33.                 "ioctl",
34.                 "listen",
35.                 "madvise",
36.                 "mmap",
37.                 "mprotect",
38.                 "munmap",
39.                 "nanosleep",
40.                 "open",
41.                 "openat",
42.                 "poll",
43.                 "pselect6",
44.                 "recvfrom",
45.                 "read",
46.                 "readlinkat",
47.                 "rt_sigaction",
48.                 "rt_sigprocmask",
49.                 "rt_sigreturn",
50.                 "sched_getaffinity",
51.                 "sched_yield",
52.                 "setsockopt",
```

https://t.me/learningnets

```
                "sigaltstack",
                "sendto",
                "set_tid_address",
                "setitimer",
                "socket",
                "vfork",
                "write",
                "writev"
            ],
            "action": "SCMP_ACT_ALLOW"
        }
    ]
}
```

These syscalls are only allowed by the policy, so by exec into the container and checking the syslog file, we can verify that all the other system calls are not being executed.

Limits and limitations

Kubernetes provides the capability to limit the number of process IDs, also known as PIDs, that a POD can invoke. These can, eventually, also be reserved for each node and allocated to the operating system or services running on top of it. It is very important to implement mechanisms to ensure that containers and PODs are not exhausting the PIDs available onto the node, preventing them from taking over the worker node and blocking vital components like kubelet or **`kube-proxy`** from working properly. Kubernetes allows three type of Process ID limits: **Node PID limit**, **POD PID limit**, and **PID based eviction**. Kubernetes can be configured to reserve a certain quantity of PIDs, preventing them from being abused by the PODs, by using the **pid=<number>** parameter in the **`--system-reserved`** option.

Kubernetes can also be configured to limit the total number of PIDs a POD can invoke as processes by setting the **`PodPidsLimit`** parameter in the kubelet config file. The limitation is indeed applied at the node level, as per the following example code:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
enableServer: true
```

https://t.me/learningnets

```
4. address: "10.10.10.100"
5. port: 20250
6. ...
7. evictionHard:
8.     memory.available:  "300Mi"
9. evictionSoft:
10.    memory.available:  "300Mi"
11. PodPidsLimit: 1000
12. ...
```

POD **Eviction** is a feature implemented by the cluster or the node to terminate misbehaving PODs based on abnormal consumption of resources. POD Eviction can be executed by the cluster; in this case, the **`kube-controller-manager`** periodically checks the status of the nodes and initiates POD termination requests when needed. Alternatively, POD eviction can be executed by the worker node itself; in this case, the **kubelet** system checks for physical resources on the host and initiates POD termination requests according to their priority or classes. The eviction mechanism at the cluster level is initiated directly via API call to the **Eviction API** system or programmatically using the **kubectl drain** command. To create an eviction object against the Eviction API, the policy expressed in JSON format would initiate the POST request:

```
1. {
2.   "apiVersion": "policy/v1",
3.   "kind": "Eviction",
4.   "metadata": {
5.     "name": "eviction",
6.     "namespace": "my-namespace"
7.   }
8. }
```

The eviction mechanism at the node level is referred to as **Node-pressure Eviction**. In essence, the kubelet service monitors memory, CPU, and disk space on the host, and when these resources reach a pre-defined consumption level, the Kubelet intentionally stops POD resource requests or initiates termination requests. In order to assume decision-making on the POD Eviction, the kubelet uses the criteria described in *Table 9.17, Eviction Criteria*:

https://t.me/learningnets

Eviction	Description
Signals	These are the status of any resource at any given time.
Thresholds	This is the minimum number of resources that the node should have available. It defines two options: • An **evictionSoft** option expects the grace period to be fully executed. • An **evitcionHard** option has no grace period.
Interval	Thresholds are evaluated on their interval values.

***Table 9.17**: Eviction Criteria*

The kubelet can initiate POD eviction based on the pressure conditions described in *Table 9.18, Node Pressure*:

Type	Description
PIDPressure	The number of process IDs has been reached.
MemoryPressure	The memory threshold has been reached.
DiskPressure	The disk space threshold has been reached.

***Table 9.18**: Node Pressure*

Kubelet then classifies the POD eviction with the following termination order:

- POD classes **BestEffort** and **Burstable** are evicted first based on their assigned priority and resource exceeding level.
- POD classes **Burstable** and **Guaranteed** are evicted last if their resource usage is less than the requests according to their priority.

The QoS mechanism explained in the previous section is not used as an eviction order parameter. There are a few but substantial limitations, especially in the Kubernetes network policies system, that should be mentioned; these are not available today:

- Redirect internal traffic to a common gateway
- Encryption in transit via TLS
- Node policies
- Logging network security events
- Querying policies and auditing tools

To overcome some the aspects not yet implemented into Kubernetes, readers are invited to go through *Chapter 12, Service Mesh Security*.

https://t.me/learningnets

Conclusion

In this chapter, we discussed various topics of the Kubernetes cluster from the security governance standpoint. We started defining what security governance is and what application of Kubernetes governance entails. We then analyzed what a policy engine is and how it can be technically implemented into Kubernetes. We had a deep discussion about the Open Policy Agent (OPA) policy engine and its complexity rule implementations, and we also introduced a valid and powerful alternative like Kyverno.

Moving on, we analyzed the security aspects of the Admission Controller Threat Model created by the Kubernetes SIG and dove into the complexity of the Kubernetes network policies and resources management. We have then discussed the complexity of the Security Policies and the various different options available for the POD Security Standards policy. Finally, we discussed how to apply limits and got a quick overview of the limitations we want to overcome.

In the next chapter, we will learn about the security of the most popular Kubernetes Cloud services, such as AWS EKS and Azure AKS.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

https://t.me/learningnets

https://t.me/learningnets

Chapter 10
Kubernetes Cloud Security

Introduction

Kubernetes is the most famous and most widely adopted container orchestrator system, and it is also the most utilized among the cloud providers. Public cloud providers like **Amazon Web Services** (**AWS**), Microsoft Azure, and **Google Cloud Platform** (**GCP**), have all created managed Kubernetes services integrated in their cloud platforms. These Kubernetes cloud services can leverage specific characteristics of the respective cloud systems and can be easily integrated with other cloud services. The cloud Kubernetes service becomes then a powerful component of a much wider and complex multi-service system that can include cloud storage, load balancing, monitoring and auditing cloud services, identity management, networking and serverless systems. In the previous chapters we have demonstrated the complexity of the Kubernetes cluster, its highly integrable modularity and expansion characteristics in conjunction with the cloud ecosystem's capabilities, provide endless possibilities. Shifting the focus from the Kubernetes system to the cloud environment from the security perspective requires an extreme effort. Although the concepts around the shared responsibility model of what are the cloud provider's responsibilities and what are the customer's responsibilities are often well-defined, but things become shady with container-based service. This is due to the shift in responsibility when moving between the various cloud services offerings, such as **Software-as-a-Service** (**SaaS**), **Platform-as-a-Service** (**PaaS**), and **Infrastructure-as-a-Service** (**IaaS**). For

https://t.me/learningnets

instance, the customer's security responsibility is reduced to the bare minimum when adopting software as a service solution, while it is amplified to the maximum with infrastructure as a service solution. Kubernetes is a container orchestrator solution, and the spectrum of its applications is too wide to be well defined or contained in a single security model, which would be anyway largely dependent on the specific cloud provider. However, being a cloud native platform solution, the Cloud Native Security Model aims to standardize Kubernetes security with the **4C security model**.

Structure

In this chapter, we will discuss the following topics:

- Cloud Native Security Model
- Amazon Elastic Kubernetes Service
- Azure Kubernetes Service
- Google Kubernetes Engine
- OpenShift
- Rancher
- Tanzu

Objectives

This chapter aims to familiarize you with the fundamental concepts of Kubernetes cloud security, describing the general security model applied to containerized and orchestrated cloud resources and the major cloud Kubernetes services available on the market, along with their pros and cons from the security perspective. We will also discuss few Kubernetes versions with a hybrid or on-premises footprint.

Cloud native security model

The **Cloud Native Security** model is a fairly recent concept that aims to address the challenges of using a containerized application and the infrastructure underneath, whether or not it is adopted as a service.

https://t.me/learningnets

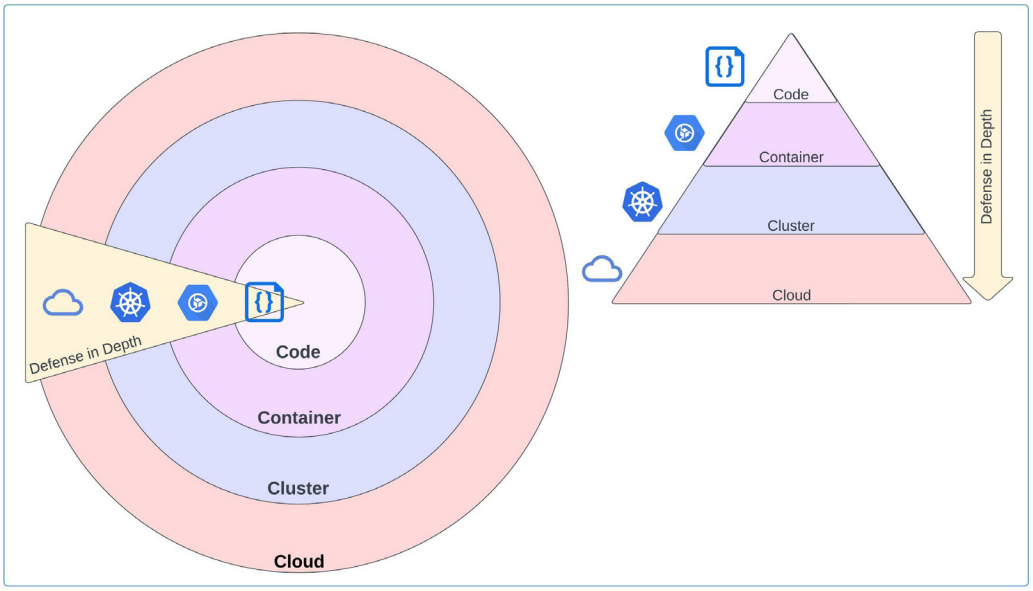

***Figure 10.1**: Cloud Native Security 4C Model*

This model outlines four concentric layers: **cloud**, **cluster**, **container**, and **code**, as illustrated in *Figure 10.1, Cloud Native Security 4C Model*, and how they are an extension of the defense in depth concept. The layered approach defined here highlights the security independence of each layer, and each layer has its own security characteristics that should be addressed, but they together contribute to the overall security of the platform.

At the lowest layer of the cloud service provider, there is the infrastructure the cluster relies upon, the outermost exposed layer that sets the security baseline not only for the cluster but for the entire cloud footprint. The other layers, even if secured, cannot benefit a vulnerable or unsecured configured infrastructure, and vice versa. The relationship is polyvalent among all the four layers. Each layer of the Cloud Native Security model can be mapped to one or more specific chapter or section of this book, except for the cloud layer, which can be addressed only when looking at cloud-specific security measures. A brief recap of the previous chapters in relation to *Figure 10.1, Cloud Native Security 4C Model*, is mapped in *Table 10.1, 4C Model Mapping*:

https://t.me/learningnets

Type	Description
Code	• Chapter 4 – Securing Container Images and Registries • Chapter 5 – Application Container Security
Container	• Chapter 3 – Container Stack Security • Chapter 4 – Securing Container Images and Registries
Cluster	• Chapter 7 – Kubernetes Hardening • Chapter 8 – Kubernetes Orchestration Security • Chapter 9 – Kubernetes Governance
Cloud	• Chapter 10 – Kubernetes Cloud Security

***Table 10.1**: 4C Model Mapping*

The 4C security model must not be seen as the cloud Kubernetes security reference best practice as it does not provide sufficient protection. It does offer some general guidelines though. It is fundamental to verify the specific cloud provider security best practices because they are the baseline security of the cluster environment and may differ quite significantly between providers.

For the container and code sections, in previous chapters of this book, we identified some security arguments that may fit the specific cases. Lastly, the cluster topic has been discussed from an unmanaged perspective, while in cloud environments, the Kubernetes service is often a managed service, which poses some limitations to the topics we have discussed so far. Before discussing in detail some of the Kubernetes cloud services, it is necessary to clarify how the cloud providers implement the Kubernetes cluster as a cloud service. In *Chapter 7, Kubernetes Hardening*, we introduced the control plane in *Figure 7.1, Kubernetes Architecture*, where we intentionally did not discuss one of the components: the **Cloud Controller Manager** or **CCM**.

https://t.me/learningnets

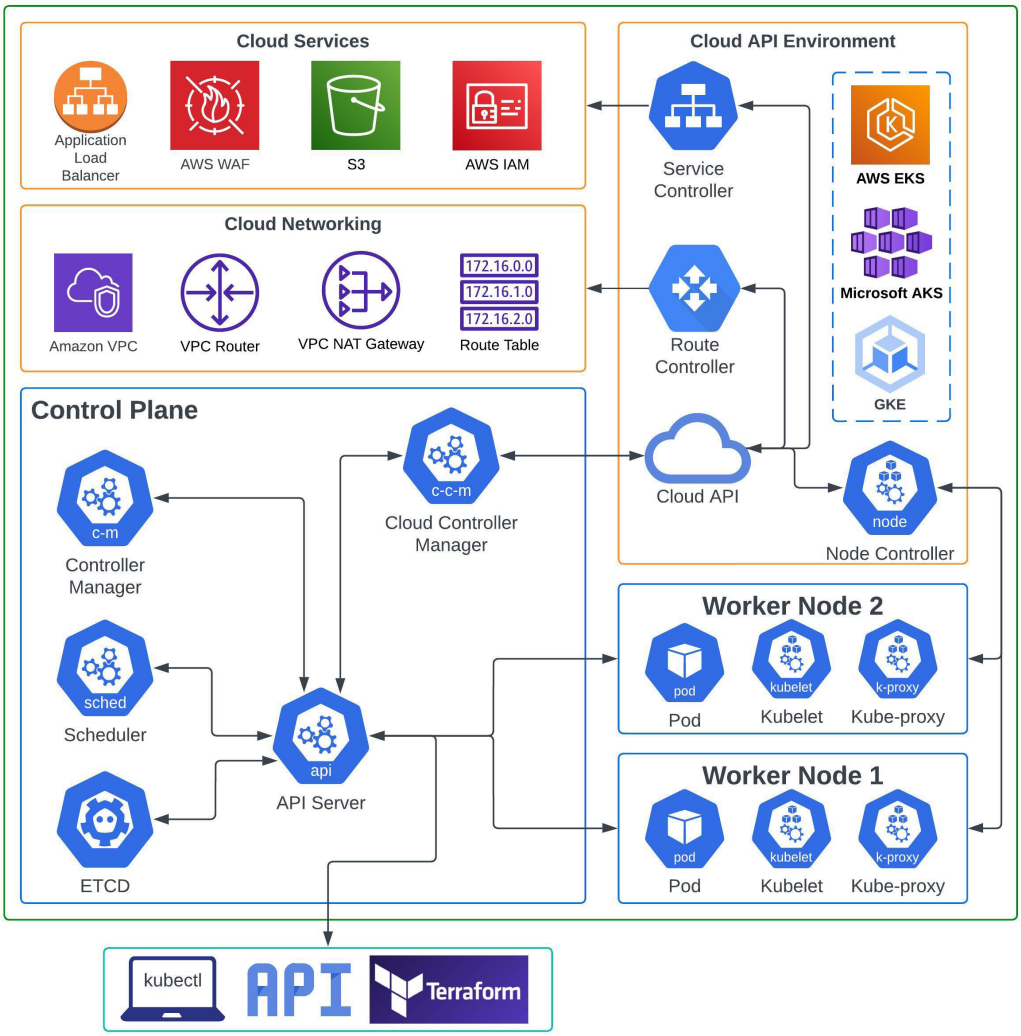

***Figure 10.2**: Cloud Controller Manager*

Recalling *Figure 7.1, Kubernetes Architecture*, and adapting its content with the introduction of the Cloud Controller Manager, the *Figure 10.2, Cloud Controller Manager* provides a visual overview of the control plane and data plane interaction with underlying cloud infrastructure. It is evident how the cloud implementation of the Kubernetes cluster adds another layer of difficulty to a system that already has a high degree of complexity. The Cloud Controller Manager is not a standalone system; it is composed of additional modules specific to the cloud Kubernetes service, which are needed to fulfil cloud requirements to interact with specific infrastructure of the cloud provider, as per *Table 10.2, Cloud Components*:

https://t.me/learningnets

Type	Description
Node Controller	Adds or removes node objects to or from the Kubernetes cloud service. The node information and type are specific to the cluster: • Computer resource. • Location (region, network, availability zone, and so on). • Health check and status retrieving.
Service Controller	Service provided via the cloud system in use that can integrate with the cluster, such as the following: • Load balancer. • Web Application Firewall. • Packet filtering. • Encryption in transit.
Route Controller	• Manages the network routes inside the cluster to redirect network traffic to PODs and containers. • Manages the underlying cluster network infrastructure.

***Table 10.2**: Cloud Components*

The description of how Kubernetes is implemented at the cloud level is of greatest importance; it helps understand the security implications of various implementation models. Although an effort can still be produced to apply security best practices to the cluster, as we have seen so far, a good chunk of the security requirements will need to be shifted to the cloud provider and implemented in accordance with the specific cloud system security methodologies.

Amazon elastic Kubernetes service

AWS does not need any introduction; it is a well-established and popular cloud service provider. It considers security for managed services like **Elastic Kubernetes Service** (**EKS**) applied via the shared responsibility model. In general, AWS is responsible for managing the Kubernetes control plane and all its components, while the customer is responsible for managing the worker nodes plus everything running on top of them.

Figure 10.3, EKS Shared Responsibility Model, shows in blue the topics that are always customers' responsibility, while the elements in orange are the ones that are always the responsibility of AWS. There are a few elements that can shift according to the sub-service the customer chooses; these are the worker node scaling option and the set of operating system, kubelet, container runtime interface and image configuration.

https://t.me/learningnets

Refer to the following figure:

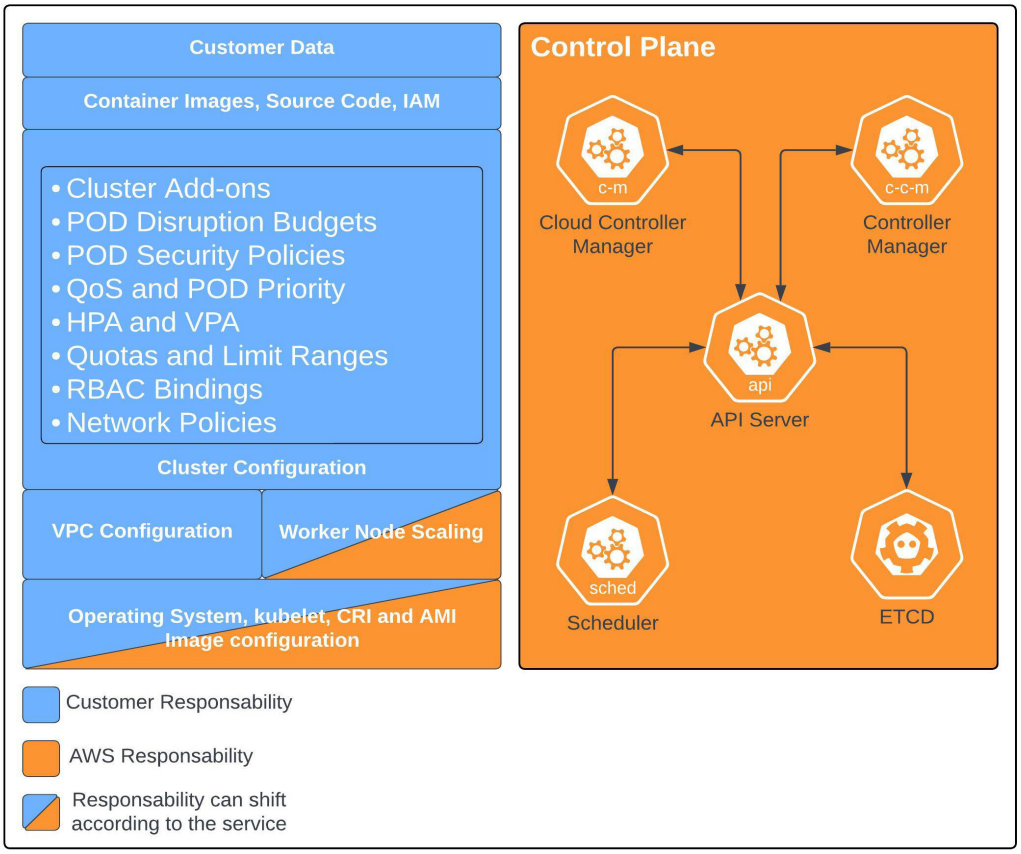

***Figure 10.3**: EKS Shared Responsibility Model*

Note: The "AMI image configuration" in *Figure 10.3, EKS Shared Responsibility Model*, is intended as Amazon Machine Image (AMI) configuration.

AWS EKS comes with three tiers, always in relation to how the worker nodes are managed, as per *Table 10.3, EKS Shifting Responsibility*:

Service	Customer Responsibility	AWS Responsibility
Self-Managed Workers	Worker Node Autoscaling OS, kubelet, CRI, AMI Image	N/A
Managed Node Groups	OS, kubelet, CRI, AMI Image	Worker Node Autoscaling
EKS Fargate	N/A	Worker Node Autoscaling OS, kubelet, CRI, AMI Image

***Table 10.3**: EKS Shifting Responsibility*

https://t.me/learningnets

Many aspects of AWS EKS security best practices can be mapped to the topics we have discussed in the previous chapters of this book, but some of them, which are not only cloud specific but are also bound to the AWS cloud provider, are going to be the arguments of the next few discussions:

- Infrastructure security
- Identity and Access Management (IAM)
- Detective controls
- Incident response

One of the key elements to secure the cloud version of Kubernetes running in AWS is to consider securing the **infrastructure** that runs it. AWS recommends adopting the infrastructure security guidelines for EKS in connection with the runtime security topic; refer to *Chapter 7, Kubernetes Hardening*, the *Securing Container Runtime Interface* section. *Table 10.4, EKS Infrastructure Security*, describes the security best practices recommendation:

Type	Description
Use a private network	Managed Node Group prior April 2020 was automatically provisioned with public IPs, but this is not the case anymore; it is worth verifying whether or not nodes have been deployed in a subnet with public exposure.
Container-specific OS	AWS recommends special purpose OS, such as Project Atomic, Flatcar Linux, Bottleroocket or EKS optimized AMI. These Linux distros reduce the attack surface of the host as per the discussion in *Chapter 1, Containers and Kubernetes Risk Analysis*, the *Attack Surface* section.
Node access	SSH access is not recommended; instead, AWS **System Manager Session Manager** (**SSM**) is preferred. There is no need to handle SSH key pairs because SSM is integrated with IAM. A custom IAM example policy is provided below.
Node update	The special purpose OS recommended above are subjected to update as any other Linux distro, so it is recommended to automate rolling patch updates, as minimal security measure.
Immutability	Rather than treating the infrastructure as upgradable, it is recommended to consider it to be immutable, which means upgrades are not necessary because when a new update is available, the worker node is replaced by a new one. This takes a bit of DevSecOps work, establishing a deployment methodology to safeguard the integrity of the cluster.

https://t.me/learningnets

Auditing	AWS recommends **kube-bench** as auditing tool; refer to *Chapter 7, Kubernetes Hardening*, the *Hardening Tools* section.
Vulnerability Management	AWS recommends using Amazon Inspector to check the EKS cluster for **Common Vulnerabilities and Exposures** (**CVE**). In order to achieve this, SSM Agent must be installed.

***Table 10.4**: EKS Infrastructure Security*

The default AWS managed policy **`AmazonSSMManagedInstanceCore`** cited in the Node Access of *Table 10.4* contains permissions not strictly required to access the worker. The following is a minimalist example code:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EnableSSMRunCommand",
      "Effect": "Allow",
      "Action": [
        "ec2messages:AcknowledgeMessage",
        "ec2messages:DeleteMessage",
        "ec2messages:FailMessage",
        "ec2messages:GetEndpoint",
        "ec2messages:GetMessages",
        "ec2messages:SendReply",
        "ssm:UpdateInstanceInformation"
      ],
      "Resource": "*"
    },
    {
      "Sid": "EnableAccessViaSSMSessionManager",
      "Effect": "Allow",
      "Action": [
        "ssmmessages:CreateControlChannel",
        "ssmmessages:CreateDataChannel",
```

https://t.me/learningnets

```
        "ssmmessages:OpenControlChannel",
        "ssmmessages:OpenDataChannel",
        "ssm:UpdateInstanceInformation"
      ],
      "Resource": "*"
    }
  ]
}
```

The SSM system leveraging the preceding policy will allow to access the worker node by simply typing the following command:

```
aws ssm start-session --target [EKS_NODE_INSTANCE_ID]
```

The preceding example is just one of the many considerations that governs **Identity and Access Management** (**IAM**) for an EKS cluster, and that implies not only access to the worker nodes but also to PODs. *Figure 10.2, Cloud Controller Manager*, illustrates how the Cloud API governs the communication between AWS native services like the AWS **Application Load Balancer** (**ALB**) and the worker nodes on which the PODs are running.

The ALB needs to be able to invoke AWS APIs and list service endpoints. Kubernetes manages access to PODs via RBAC roles; these are assigned automatically by the cluster to namespaces via a **default service account**. The security concern is that upon cluster creation, the API endpoint is public to the internet by default; even if the authentication would require a combination of RBAC and AWS IAM, it represents needless exposure. The recommendation is to limit the exposure to the least or disable the API public access completely. To achieve this, the endpoint private access must be enabled to keep the communication with the cluster API server within the boundaries of the AWS VPC. This will create a private hosted zone in AWS Route 53 linked to the VPC in which the cluster has been deployed. Furthermore, to properly route network traffic to the **kube-apiserver**, the VPC must have the **AmazonProvidedDNS** option set into the DHCP, and both **enableDnsHostnames** and **enableDnsSupport** should be set with **true** as value.

In *Chapter 9, Kubernetes Governance*, we discussed how policy agents can apply security requirements at scale; specifically, the Kyverno engine provides an example of how to enforce IRSA onto the cluster. IRSA stands for IAM Role for Service Account, a feature of the AWS Identity and Access Management service that allows us to assign an AWS IAM Role to a Kubernetes service account. Therefore, when PODs have assigned a service account that is connected to an IAM role, the kube-apiserver will invoke the **Open ID Connect** (**OIDC**) endpoint that has been configured as **Identity Provider** (**IDP**).

https://t.me/learningnets

The OIDC endpoint signs the token issued by the Kubernetes API server via cryptography, allowing the POD to communicate with the IAM role. According to the policy assigned to the IAM role, the POD will be able to communicate with the service listed, for example, with an Application Load Balancer or an AWS S3 bucket. To provide an example of token structure, the following code is an example of JWT token for IRSA:

```
{
  "aud": [
    "sts.amazonaws.com"
  ],
  "exp": 1675532114,
  "iat": 1675445714,
  "iss": "https://oidc.eks.us-east-1.amazonaws.com/id/
  T34DE27G27A865933133EA00A26FB256",
  "ubernetes.io": {
    "namespace": "default",
    "pod": {
      "name": "alpine-44b5774646-af911",
      "uid": "4a20e883-1407-12ia-a75c-0e72b7e4f463"
    },
    "serviceaccount": {
      "name": "s3-read-only",
      "uid": "b270bb5c-4406-91ea-3898-123b45b60dfg"
    }
  },
  "nbf": 1675445714,
  "sub": "system:serviceaccount:default:s3-read-only"
}
```

As per part of the shared responsibility model showed in *Figure 10.3, EKS Shared Responsibility Model*, the control plane is managed by AWS. The control plane logs must be enabled to achieve visibility into the logs generated by the **kube-scheduler**, the **`kube-apiserver`**, the **etcd** data store and the **kube-controller-manager**.

https://t.me/learningnets

Note: AWS warns that enabling control plane logging will incur extra costs. Readers are encouraged to verify those costs and understand if enabling control plane logs meet their requirements.

Table 10.5, EKS Detective Controls, provides a quick overview of logging elements that should be monitored to gain visibility in the Kubernetes cluster:

Type	Description
Metadata	Log and verify API calls authorization requests and the reason why authorization was granted by inspecting the following two annotations: • authorization.k8s.io/decision • authorization.k8s.io/reason
Alerts	Leverage the **host**, **sourceIPs**, and **k8s_user.username** attributes to generate alerts where there is an abnormal increase in the following response: • 403 Forbidden • 401 Unauthorized
Log Insights	Monitor the following RBAC objects with CloudWatch Log Insights: • Roles • RoleBindings • ClusterRoles • ClusterRolesBindings
CloudTrail	Logs IRSA API calls; when CloudTrail logs show unauthorized service accounts, it may be sign of a policy misconfiguration.
CloudTrail Insights	CloudTrail Insights identifies anomalies in API request volumes.

***Table 10.5**: EKS Detective Controls*

When an anomaly is detected or an alert is generated, the log should be treated as a security event. A security event is a drift from a system's expected behavior, which should be investigated to determine whether an incident actually occurred. AWS provides a **Security Incident Response** (**IR**) Guide as part of the AWS Whitepapers. The **AWS Security Incident Response Guide**, available at **https://docs.aws.amazon.com/whitepapers/latest/aws-security-incident-response-guide/aws-security-incident-response-guide.html**, follows the **National Institute of Standards and Technology** (**NIST**) security best practices formalized in the **Computer Security**

https://t.me/learningnets

Incident Handling Guide Special Publication (**SP**) 800-61 r2. The NIST SP 800-61 r2 provides the following incident response phases:

- Preparation
- Operations
    - Detection
    - Analysis
    - Containment
    - Eradication
    - Recovery
- Post-Incident Activity

The various steps of the Operations phase can be mapped to the AWS EKS Incident Response, as shown in *Table 10.6, EKS Incident Response*:

Type	Description
Detection	This first phase of the IR Operations is fulfilled by implementing the detective controls explained earlier. Monitoring and visibility are important topics of any Incident Response plan.
Analysis	AWS recommends the following identification methodologies for PODs and worker nodes: • Workload analysis • Service account analysis • Vulnerable or compromised images
Containment	Containment is applied by means of the following: • Isolating the POD • Detaching the IAM role or policy • Cordoning the node • Disabling POD termination
Eradication	In relation to the analysis and containment phases, eradication is the removal of all the unauthorized resources by mitigating all the exploited vulnerabilities and eliminating malware software.
Recovery	Deploy a new workload that contains the fixes or patches to the exploited vulnerabilities to kill the affected PODs or worker nodes, bringing up a new set of patches systems.

***Table 10.6**: EKS Incident Response*

https://t.me/learningnets

The **analysis** steps described in Table 10.6 can be expanded by providing a few quick examples in the form of code that can be utilized to identify the allegedly compromised systems by looping into the workloads. For instance, workloads are one or more applications running on the cluster, in Kubernetes term this can also be intended as one or more PODs created to run the applications. The following code is an example of how to identify worker nodes running compromised PODs, in essence a simple example of workload analysis:

```
$ kubectl get pods <compromised-pod-name> --namespace <namespace> \
-o=jsonpath='{.spec.nodeName}{"\n"}'
```

If the POD is part of a workload resource like a Deployment, all the PODs running in the same workload are likely to be exposed. To list all the PODs of the compromised workload, use the following code:

```
$ deployment=$(kubectl get deployments <compromised-deployment-
name> \
  --namespace <namespace> -o json | jq -j \
'.spec.selector.matchLabels | to_entries | .[] | "\(.key)=\(.value)"')

$ kubectl get pods --namespace <namespace> --selector=$deployment \
-o json | jq -r '.items[] | "\(.metadata.name) \(.spec.nodeName)"'
```

Similarly, when the service account is compromised, all the PODs running with that service account are likely to be compromised. We can identify the PODs using the compromised service account by running the following command:

```
$ kubectl get pods --namespace <suspicious-namespace> -o json | \
    jq -r '.items[] |
    select(.spec.serviceAccount == "<compromised-service-account-
name>") |
    "\(.metadata.name) \(.spec.nodeName)"'
```

In complex deployment environments, with high container interaction, sometimes it is difficult to keep track of all the container images stored in the registry. When an image is suspected to have exploitable vulnerabilities, we can filter all the running PODs on that specific container image by running the following command:

```
image=<vulnerable-image/tag>

$ kubectl get pods --all-namespaces -o json | \
```

https://t.me/learningnets

```
4.      jq -r --arg image "$image" '.items[] |
5.      select(.spec.containers[] | .image == $image) |
6.      "\(.metadata.name) \(.metadata.namespace) \(.spec.nodeName)"›
```

The **containment** steps described in *Table 10.6* can be analyzed further by providing the following example code that can be utilized to set a contained perimeter around the compromised resources, stopping them from spreading the damage. We can isolate a POD using a network policy like the one described in *Chapter 9, Kubernetes Governance*, section *Network Policies*:

```
1. apiVersion: networking.k8s.io/v1
2. kind: NetworkPolicy
3. metadata:
4.   name: deny-by-default
5. spec:
6.   podSelector:
7.     matchLabels:
8.       app: compromised-pod-label
9.   policyTypes:
10.  - Egress
11.  - Ingress
```

Worker nodes, in terms of host systems, are not affected by the preceding policies. If the suspected compromised resources are the nodes, it is more effective to use the AWS Security Groups to achieve isolation. In an IRSA scenario where the POD communicates with other AWS resources, such as **AWS Elastic Load Balancer** or **AWS Simple Storage Service** (**S3**), using an **OIDC** token, detaching the policy that allows the POD to communicate with the service would prevent any further network communication. Another way to achieve worker node isolation is to instruct Kubernetes to exclude the compromised node from the cluster. With the **cordon** command the **kube-scheduler** will mark the node as **unschedulable=true** and assign the workload to the remaining nodes. Following an example of **cordon** command:

```
1. $ kubectl cordon <compromised-node-name>
```

It is also recommended to enable termination protection to stop an attacker's attempt to cover their traces by terminating the PODs. That's why the isolation technique is preferred.

https://t.me/learningnets

Azure Kubernetes Service

The Microsoft **Azure Kubernetes Service** (**AKS**) was originally only one of the three services included in what was known as **Azure Container Service** (**ACS**), along with Docker Swarm and Apache Mesos. Due to the high increase in demand and popularity of the Kubernetes system, in mid 2018, the Azure team decided to extract Kubernetes as a standalone service from ACS, creating the Azure Kubernetes Service. There is no specific shared responsibility model directly referencing the Azure Kubernetes Service; instead, there is a broader representation of the responsibility according to stack deployment typology. The responsibility shifts from the customer, running a full on-prem environment progressively to Microsoft in a Software-as-a-Service environment. We can deduce that for AKS, Microsoft's responsibility is to secure the cluster infrastructure, and the customer's responsibility is to secure the cluster itself and the PODs and containers deployed on top of it.

Microsoft Azure's approach to security is based on the **Microsoft Cloud Security Benchmark** (**MCSB**). **MCSB** is a set of security controls or recommendations that are referenced by the Azure services.

Note: MCSB is the successor of the Azure Security Benchmark (ASB), a rebrand Microsoft pursued in October 2022. The ASB is still an applicable security framework and has provided the baseline for the MCSB.

The step forward from the **Azure Security Benchmark** (**ASB**) to the MCSB is due to the continuous increase in cloud services offered by Azure and the fast-paced development environment with new features released constantly. The Azure Security Benchmark has three active versions (v1, v2, and v3), and the v3 has provided the baseline for the first version of MCSB.

It is also worth noting that the DevOps security concept appears only from the third version of the Azure Security Benchmark onward, an update needed to implement DevSecOps solutions in the cloud.

Although Azure Security Benchmark v3 and MCSB v1 seem the same at a first look, they do provide one key differentiator: the AWS guidance.

https://t.me/learningnets

The MCSB **AWS guidance** maps the Azure security controls with the popular Amazon Web Services platform to enable multi-cloud capabilities. This section can reveal itself useful in migration scenarios when organizations are willing to move from the popular cloud provider to Azure.

Figure 10.4, ASB vs MCSB, illustrates the evolution of the Microsoft Azure security recommendations over time, from version 2 onward the framework has been pretty stable with minor adjustment. Of all the security recommendations, only network security, data protection and incident response have been unaltered, with the others following the natural evolution of the cloud platform, and therefore, the risks associated with it. Refer to the following figure:

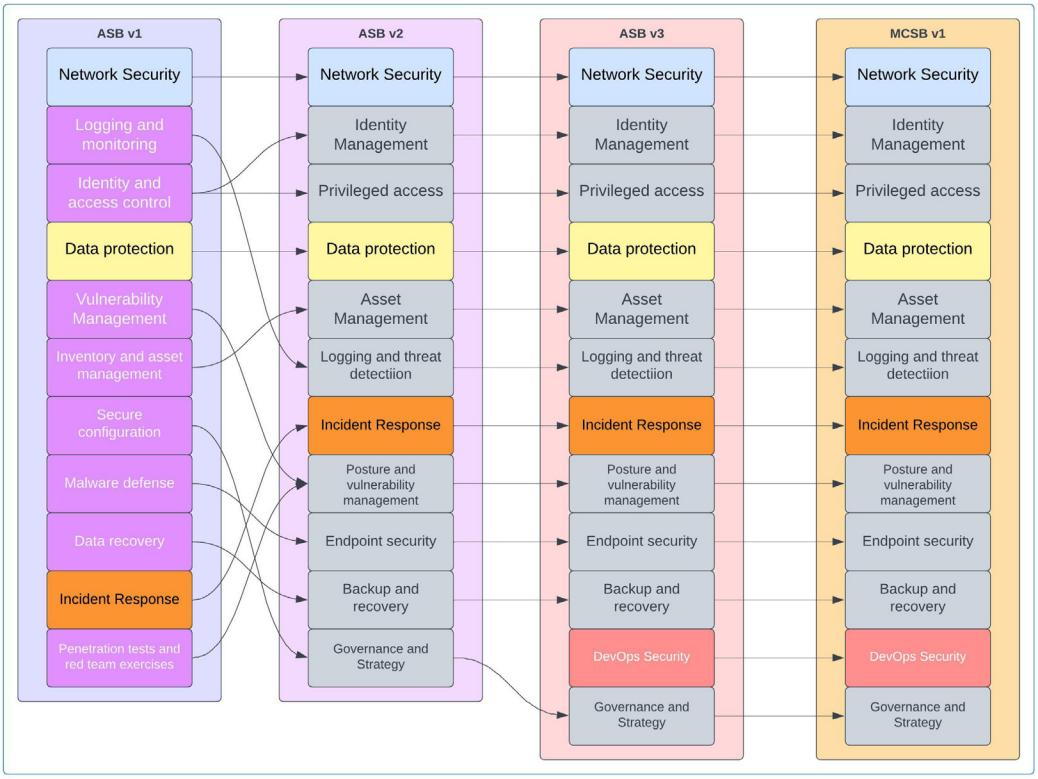

***Figure 10.4**: ASB vs MCSB*

https://t.me/learningnets

The MCSB security framework is based on two key aspects of the security spectrum, as per *Table 10.7, Microsoft Cloud Security Benchmark*:

Type	Description
Security Controls	The security controls span the entire cloud platform and are not service specific. This is the case of either the three versions of the Azure Security Benchmark or the Microsoft Cloud Security Benchmark. Readers can consider these as security best practices in the cloud to define the following: • The **security principle** behind the recommendation. • The **Azure guidance** in how to achieve the intended security control. • The Azure implementation and additional context, with links to the guidelines of the various aspects involved in the security controls. • Customer security stakeholders, typically involved in the various phases of the implementation plan. • **AWS guidance**, a service mirror implementation solution with the Amazon services.
Security Baseline	The security baseline applies the security controls from the cloud service standpoint. It extracts, from the list of security controls, only those that are relevant to the specific Azure service.

***Table 10.7**: Microsoft Cloud Security Benchmark*

Note: Not all the Azure services are using the latest security benchmark, specifically the MCSB v1. On the contrary, many Azure cloud services still apply different security benchmark versions.

The AKS security is based on the very first version of the Azure Security Benchmark, and it follows the implementation logic expressed in *Figure 10.4, ASB vs MCSB*, in the **ASB v1** section. Due to its complexity, all the aspects listed in the ASB version 1 are in scope for the Kubernetes cluster. The listing of such baseline controls would result in a tedious copy and paste exercise, which would not benefit the scope of the chapter and overall, of the book.

For the reason mentioned and also in consideration of the vast coverage the previous chapters have been able to provide about the Kubernetes system, this section will focus on three key security aspects of the AKS service: **cluster security**, **node security** and **Microsoft Defender for Containers**. **Cluster security** for Azure Kubernetes Service is specific to this service, and it considers the security controls discussed in the Azure Security Benchmark version 1. In relation to the Azure service that contributes to the overall security of the Kubernetes cluster in the cloud, *Table 10.8, AKS Cluster Security* provides an overview of the topics of interest:

https://t.me/learningnets

Type	Description
Identity	The cluster needs an identity to leverage other Azure resources like storage, vault, and application gateway. When a new cluster is created, Azure assigns a **managed identity** to it, which is handled by the cloud system via certificate-based authentication. To achieve control on the cluster identity, we can replace the managed identity with a **service principal**. In this case, the identity must be managed by the user, including credential provisioning and rotation.
Credentials	Kubernetes Clusters with service principals as identity have a 1-year expiration time. Credentials must either be extended or rotated. This is mandatory, or the cluster will just stop working. Credentials rotation or update can be executed either manually or via a predefined security policy, or in case where **Azure Active Directory** (**AD**) is the authentication authority, two more identities would need reset or rotation: • AD Client App • AD Server App
Configuration file	Access to the **~/.kube/config** file can be restricted via Azure **Role-Based Access Control** (**RBAC**) by defining who can access the Kubernetes config file. Azure offers two built-in roles: • AKS Cluster User Role • AKS Cluster Admin Role The preferred way is to define user permissions via Azure AD with the **clusterUser** Role. This option allows much more granularity control, and it is based on custom authentication permissions.
kube-apiserver	Restrict access to the Kubernetes API server by controlling the IPs that are allowed to communicate with the kube-apiserver. This can be achieved during the cluster creation by passing the --**api-server-authorized-ip-ranges** parameter.
etcd	In order to achieve encryption at rest on the Azure cloud platform, AKS utilizes the Key Management Service (KMS) system known as **Azure Vault**. All the secrets stored in etcd will benefit from the encryption at rest.
RBAC	As per discussion in the identity section at the beginning of this table, leveraging Azure AD will enable you to configure access (authentication and authorization) to the cluster using Azure AD users, groups and service principals. After Azure AD authentication, the built-in Kubernetes RBAC will manage access to the cluster resources, including namespaces.

https://t.me/learningnets

Type	Description
Certificate	AKS uses certificates for secure internal communication. With a cluster using RBAC deployed after March 2022, the certificates are rotated automatically by Azure. Before March 2022, this feature was not available, and if the cluster needs certificate rotation in place, for compliance reasons for example, the rotation is executed manually.
Azure Policy	The Azure Policy system leverages the Gatekeeper admission controller to implement the Open Policy Agent (OPA) system as described in *Chapter 9, Kubernetes Governance*, the *Policy Engines* section.
ImageCleaner	At the time of writing this chapter this feature is in **preview** state, meaning that has not been fully implemented by Azure. It enables the AKS cluster to define rules to delete old or stale container images that can present security issues due to vulnerabilities.

***Table 10.8**: AKS Cluster Security*

As system counterpart of the Kubernetes control plane, the worker nodes are an element of security concern because they materially execute the container workload. That is why node security is an important topic of the overall security of the cluster. These security controls are not intended to supersede the node security measures discussed in the previous chapter, but they are willing to add a new layer of security for the recommendations proposed by the Azure platform. *Table 10.9, AKS Node Security*, details the security measures that AKS can provide to secure the worker nodes:

Type	Description
Host	One of the security measures adopted to enhance worker node security is to use the Azure Dedicated Host service. This enables the cluster to use dedicated physical machines that are uniquely allocated the AKS system.
CVM	Azure provides the **Confidential Virtual Machine** (CVM) service. This security feature enables the worker nodes to use encrypted RAM, protecting the host memory processing. It is based on **Advanced Micro Devices** (AMD) SEV-SNP technology which stands for **Secure Encrypted Virtualization-Secure Nested Paging**.
Encryption	AKS offers the **host-based encryption** service where the virtual machine used as worker node can be encrypted at rest, and the data communication with the storage system of the same virtual machine is encrypted, achieving encryption in transit with the backend storage system as well.
BYOK	The previous host-based encryption service can optionally utilize a custom key through the **Bring Your Own Key** service. In this case, the encryption keys are managed by the user or cluster admin.

https://t.me/learningnets

FIPS	Although not common, AKS can provide the **Federal Information Process Standard** (FIPS), a security framework requested by the US government for organizations that need to elevate their security standards in order to be compliant with US federal bodies security requirements.

***Table 10.9**: AKS Node Security*

Microsoft Windows users would likely be familiar with the **Microsoft Defender** system. Like many other Microsoft host-based systems, the Defender platform has evolved over the years; today, it is the Azure cloud-native solution for containers. **Microsoft Defender for Containers** is the cloud-based service that Microsoft offers to monitor, maintain and improve the security of the AKS cluster, and broadly, of the containers and applications running in the cloud environment. This service provides three key security aspects, as listed in *Table 10.10, Microsoft Defender for Containers features*:

Type	Description
Environment Hardening	This feature is not intended as an application of a hardening framework like the Center for Internet Security (CIS) Benchmark or in relation to the solution discussed in *Chapter 7, Kubernetes Hardening*; rather, it is a clever system that surfaces gaps in cluster configuration and proposes fixes.
Vulnerability assessment	Microsoft Defender scans the Azure Container Registry to identify the container images that contain known vulnerabilities. The scans provide visibility into the container workloads and guidance to protect and remediate security issue that might affect the containerized environment.
Threat Protection	Real-time threat protection is offered out of the box by Azure Microsoft Defender for Containers. The platform is designed to generate alerts for any security event detected in the container workloads and can be enabled also at the host layer. This feature is extremely useful in POD Escaping scenarios. The service is based on the MITRE ATT&CK matrix for Containers.

***Table 10.10**: Microsoft Defender for Containers features*

Azure Kubernetes Service provides a few more security aspects that must be noted, but not discussed, as their specifics have been covered in the previous chapters of this book: **build security**, **registry security** and **application security**.

Google Kubernetes Engine

Google Kubernetes Engine (**GKE**) is the oldest of the cloud Kubernetes service, and it is also the quickest to adopt new features, evolving its infrastructure and service

https://t.me/learningnets

offering constantly. After all, before becoming a graduated Cloud Native Computing Foundation project, Kubernetes was born at Google and was later donated to the CNCF community.

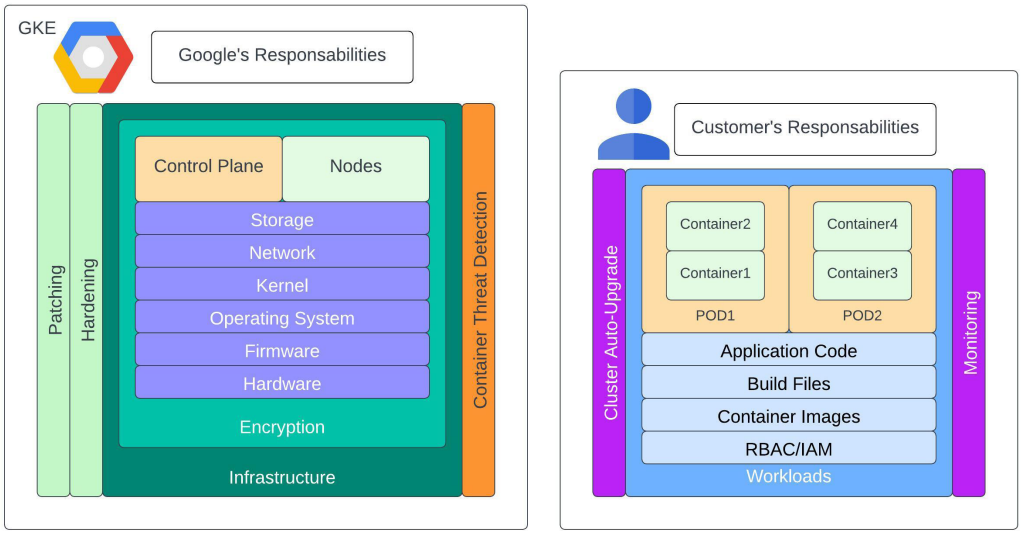

***Figure 10.5**: GKE Shared Responsibility Model*

Google GKE Shared Responsibility Model is slightly different from the previous cloud providers, and it is somehow simpler because there is a clear separation of duties between Google and the customer. With reference to the Cloud Native Security 4C Model described at the beginning of this chapter, Google's responsibilities are clearly defined around the Cloud and Cluster, while the customer's responsibilities are bound to the Container and the Code, as illustrated in *Figure 10.5, GKE Shared Responsibility Model*. In essence, Google's responsibilities include everything except cluster upgrades and workload management. Monitoring cluster and worker nodes activities is also considered, naturally, a customer's responsibility, although this can be achieved with specific tools: the **Google Cloud Operations Suite** and the interesting **Security Posture Dashboard**. The latter has a footprint similar to the Microsoft Defender for Containers in terms of features, but it is still in preview at the time of writing this book.

Google praises itself for custom designing the network equipment and server boards used either in their data centers or in the **Google Cloud Platform** (**GCP**). Among those custom designed hardware solutions, there is **Titan**, a hardware security chip deployed on Google's server, which enables cryptographic signature at the hardware layer to ensure that operating systems are booting securely.

https://t.me/learningnets

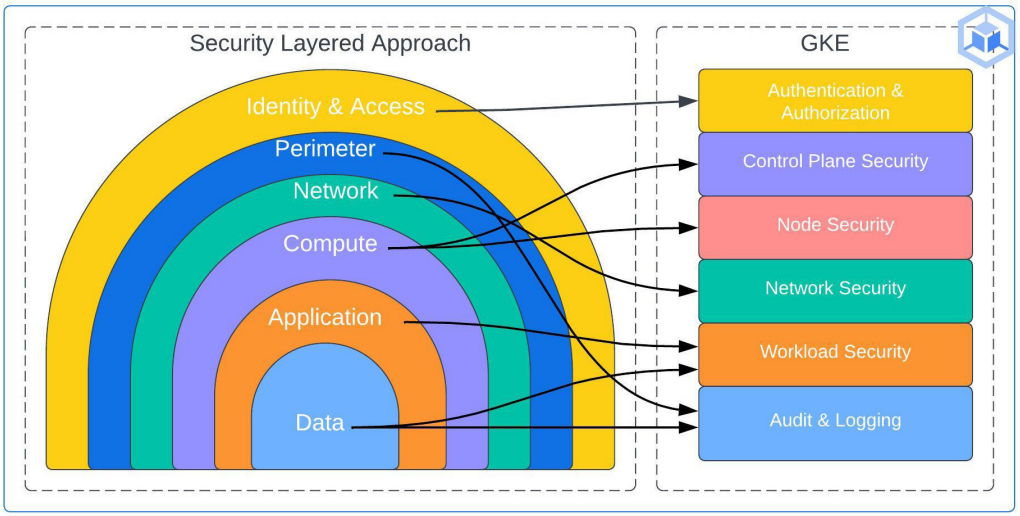

***Figure 10.6**: Layered approach*

GKE provides different ways to secure the cluster and the workloads, but the recommended one is the security layered approach. The traditional security layered approach is to look at a platform from the inside out. *Figure 10.6, Layered approach*, illustrates this on the left side, in the bottom-up direction. Therefore, the security layered approach moves outward starting with the **Data** layer at the inner center of the diagram. The outward direction indicates the increased level of exposure terminating with the **Identity & Access** as the outer layer. GCP maps the security layered approach with the Google Kubernetes Engine security controls, selecting the security measures most relevant to the Kubernetes cluster and its deployment in a cloud environment. In a GKE cluster, the users are managed by the Google Cloud Platform, and in order to clarify their purpose in contrast with the traditional Kubernetes users, *Table 10.11, GKE User Types*, describes the subtle difference between them:

Type	Description
User Account	Although Kubernetes has knowledge about these accounts, it does not manage them.
Service Account	These are accounts managed by Kubernetes only, and they can used only by Kubernetes resources.
Google Account	A Google Account grants access to Google-based services, for example, Gmail.
Google Cloud Service Account	The GCP Service Accounts are Google Cloud identities, and they manage authentication and authorization between GKE and other GCP services.

***Table 10.11**: GKE User Types*

https://t.me/learningnets

GKE leverages the Role-Based Access Control mechanism to extend control and provide access granularity to the Google Cloud project or the Kubernetes cluster itself. When RBAC is used, readers should ensure that they disable the legacy "Attribute Based Access Control" feature that is still present in the GKE system. For a quick recap of what ABAC is, refer to *Chapter 8, Kubernetes Orchestrator*, the *RBAC vs ABAC* section. Google's approach to **control pane security** is not different from the other two big competitors; similarly, it's not different from most other cloud Kubernetes providers for that matter, which include in scope kube-apiserver, controller manager, scheduler and etcd database. *Table 10.12, GKE Control Plane Security*, summarizes the main element of the security spectrum used by GKE:

Type	Description
Hardening	The hardening concept in Google is different from what has been discussed in *Chapter 7, Kubernetes Hardening*; it is based on the adoption of what GCP calls the **Container-Optimized OS**. This operating system is a hardened version of **Chromium OS**, implementing the various characteristics of a typical Linux system, such as Seccomp and AppArmor, but also some peculiar features of the Linux kernel, such as **Integrity Measurement Architecture** (IMA) and **Kernel Page Table Isolation** (KPTI).
Cluster Trust	A control plane is an ensemble of many moving parts, and these parts communicate between themselves and externally, for example, with worker nodes, PODs, and vice versa. Requests and responses are sent over SSH tunnels or leveraging the Konnectivity Service (reference *Chapter 7, Kubernetes Hardening*, section *Securing Network Communication*) over Transport Layer Security (TLS). On the most recent clusters, TLS version 1.3 is the default.
Authorized Network	Specify CIDR ranges that are allowed to access the GKE cluster over HTTPS. As per the Cluster Trust topic, communication is secured via TLS. All the other communication requests are dropped.

***Table 10.12**: GKE Control Plane Security*

Within the **control plane security** topic, it is worth noting the **private cluster** concept. In a typical user scenario, being the cluster in the cloud, the user must have internet access to communicate remotely with Kubernetes. In enterprise environments, it is common to see users accessing cloud resources via a **Virtual Private Network** (**VPN**) connection that secures the communication between the two endpoints, but that is not always the case.

https://t.me/learningnets

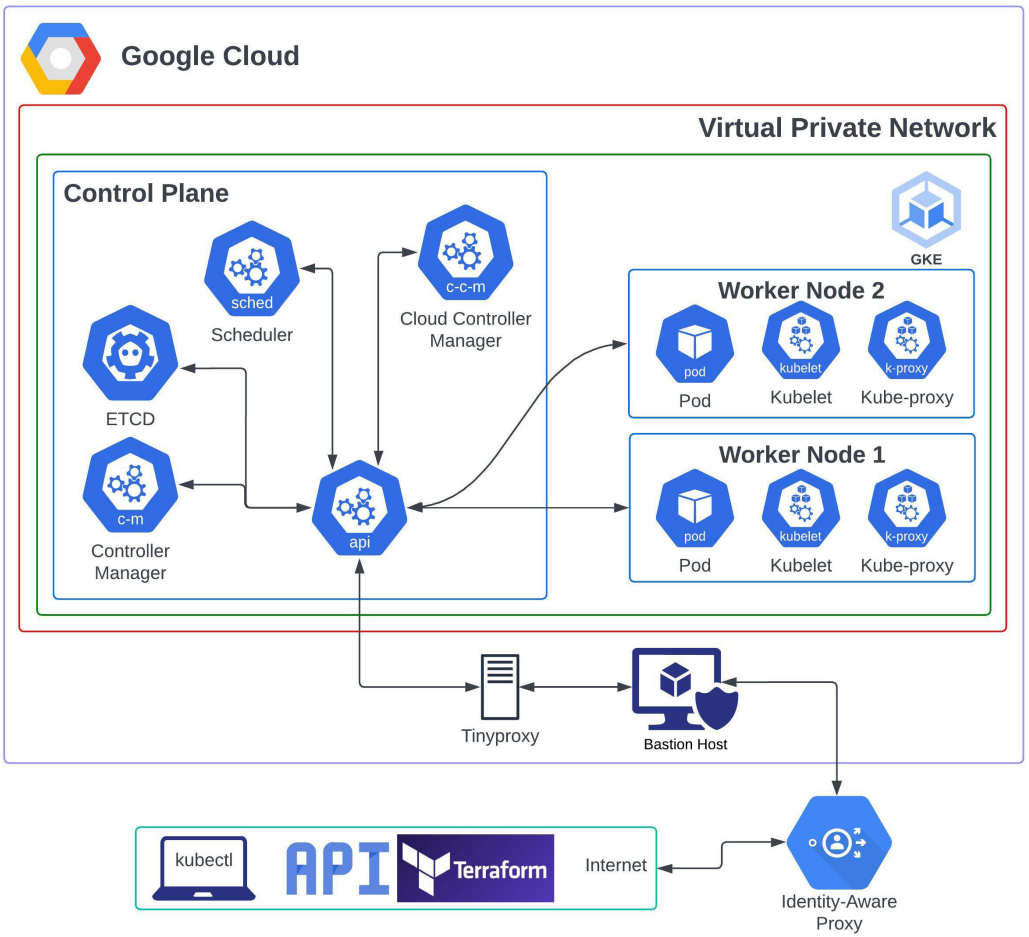

***Figure 10.7**: GKE Private Cluster*

A **GKE Private Cluster** is deployed in virtual private networks where worker nodes, services, PODs and containers have been assigned only internal IP addresses, as shown in *Figure 10.7, GKE Private Cluster*. Instead, the control plane has both public and private endpoints, and the public endpoint can be disabled. To be able to remotely access a cluster running in a completely private network, Google Cloud uses the **bastion host** access mechanism. The remote access is handled via the **Identity Aware Proxy** (**IAP**), which must be enabled for the bastion host instance; then, the request is redirected against the Google user authentication system, which can also be attached to a role. If authentication and authorization to the bastion host are granted, the request is proxied to the GKE cluster through the **Tinyproxy** service. A code example of how to create a GKE private cluster is shown below; consider the **/28** subnet mask requirement:

https://t.me/learningnets

```
$ gcloud container clusters create my-private-cluster \
    --create-subnetwork name=my-private-subnet \
    --enable-private-nodes \
    --enable-private-endpoint \
    --enable-master-authorized-networks \
    --enable-ip-alias \
    --master-ipv4-cidr 172.12.16.32/28
```

When the **`--enable-private-endpoint`** option is declared the private cluster runs completely on a private network with private IP addresses, but it can still communicate with Google Cloud external services by exposing Kubernetes services, such as a load balancer. One of the most interesting applications of **node security** in the security layered approach at Google is the **Shielded GKE Nodes**. A Shielded GKE node is what Google calls a **Shielded VM (Virtual Machine),** addressed specifically to work as cluster worker nodes. A Shielded VM is a type of virtual machine that has achieved a higher level of security due to the following characteristics:

- Secure Boot
- Virtual Trusted Platform Module (vTPM)
- Integrity Monitoring

The first two topics can be reviewed in *Chapter 2, Hardware and Host OS Security*, while the Integrity Monitoring feature is a monitoring service that ingests data about the Secure Boot and vTPM events logging. GKE uses instance metadata to configure virtual machines as worker nodes, but the metadata extracted from the VMs contains information that PODs or containers do not need access to, for example, the service account key. GKE provides two ways to secure instance metadata from unwanted exposure, as listed *Table 10.13, Securing instance metadata*:

Type	Description
Metadata Concealment	The Metadata Concealment is a not recommend approach, as it is due to be deprecated; but in principle, it protects access to kube-env by firewalling traffic from PODs that are not running on HostNetwork to the cluster metadata server.
Workload Identity	It replaces Metadata Concealment by creating an access approach similar to the AWS IRSA solution. By applying Identify and Access Management (IAM) policies to the workloads, we can tailor granular access permissions and therefore, stop PODs and containers from being able to access instance metadata.

***Table 10.13**: Securing instance metadata*

https://t.me/learningnets

The elements of the Private Cluster solution are also elements of network security, which does not end at securing communication between users, control plane and worker nodes; it goes further, for the two considerations expressed in *Table 10.14, GKE Network Security*:

Type	Description
POD-to-POD	This is a concept already expressed in previous consideration, but it is worth considering that Google Kubernetes Engine applies a similar solution because it strengthens the effectiveness of using network policies to govern network traffic between PODs. As a reminder with no policies in place, all traffic is allowed.
Load Balancer Filter	A **LoadBalancer** service type that matches PODs' labels can filter traffic to ensure that only authorized traffic reaches the PODs. The **loadBalancerSourceRanges** parameter can help achieve filtering by specifying the allowed CIDRs.

***Table 10.14**: GKE Network Security*

The workload security layer in GKE has two main preventive measures, as listed in *Table 10.15, GKE Workload Security*:

Type	Description
GKE Sandbox	Workload isolation helps in preventing POD escaping scenarios by segregating the container kernel from the host kernel. In essence, the sandboxing stops the POD from sending syscall to the host kernel leveraging on the **gVisor** solution. As reference, you can revisit *Chapter 8, Kubernetes Orchestrator*, the *Cluster Isolation* section.
Binary Authentication	The purpose of this Google Cloud service is to provide visibility in software supply chain issues. Binary Authentication is based on the **Kritis** system, which enforces container image verification against the policy defined, and it is suitable for DevSecOps implementation. If the image does not satisfy the policy requirements in terms of software, libraries, and their versions, the deployment will fail.

***Table 10.15**: GKE Workload Security*

The following example code illustrates a **MySQL** deployment sandboxed via **gVisor**:

```
1. # gVisor.yaml
2. apiVersion: apps/v1
3. kind: Deployment
4. metadata:
5.   name: mysql
```

https://t.me/learningnets

```
  labels:
    app: mysql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      runtimeClassName: gvisor
      containers:
      - name: mysql
        image: mysql
```

As mentioned at the beginning of this section, Google Cloud provides a new feature for the **audit and logging** layer; it is called the **Security Posture Dashboard**. The dashboard scans clusters and container workloads against security industry best practices to determine whether any vulnerability or configuration issue is affecting the system. Lastly, Google recommends using **GKE Autopilot**, an automated Google security framework that applies most of the security measures described, and more. This is a very simple way to deploy a Kubernetes cluster in Google Cloud with a baseline of already active security requirements.

Red Hat OpenShift

OpenShift is the Kubernetes solution offered by Red Hat. OpenShift is mainly focused on hybrid cloud enterprise solutions, trying to unload organizations from the burden of managing complex infrastructures. The base OpenShift solution is the **Red Hat OpenShift Kubernetes Engine**, which represents the infrastructure and orchestration Kubernetes system. It has been tailored to work on Red Hat Enterprise Linux and Red Hat Enterprise Linux CoreOS.

The second tier is the **Red Hat OpenShift Container Platform**, where OpenShift brings security into the Kubernetes cluster by implementing the DevSecOps approach based on three key concepts: build, deploy, and run. To **build** security into

https://t.me/learningnets

the applications, OpenShift recommends the key elements described in *Table 10.16*, *OpenShift Build*:

Type	Description
Trust	Gaining control on the container's content by creating trusted Linux systems, the **Red Hat Universal Base Images**. The images are constantly monitored via the **Container Health Index**, which provides insights into the health status of the container images. The container images are rebuilt when a new version is released.
Registry	OpenShift provides out-of-the-box Red Hat Quay, a private registry that has built-in vulnerability scanning capabilities.
Building	OpenShift leverages the **Source-to-image** (S2I) framework to combine custom source code of the application with container base images.
Pipeline	OpenShift spans security across the CI/CD pipeline: • Scans container images into the registry • Implements vulnerability checks and remediations into the integrated development environment (IDE) through the **Red Hat Dependency Analytics** • Ensures real-time assessment integration with **Jenkins** and **Tekton**; Tekton specifically address the need of CI/CD pipelines with Kubernetes Operators

***Table 10.16**: OpenShift Build*

The approach to the **deployment** concept is achieved with the elements described in *Table 10.17, OpenShift Deploy*:

Type	Description
Platform	Implements **Kubernetes Operators**, a feature that packages an application into a single deployment component, regardless of its complexity. This simplifies the management life cycle of the application.
IAM	Identity and Access Management is achieved by the OpenShift control plane through the built-in **Cluster Authentication Operator** (CAO). CAO is deployed as a Kubernetes Operator, which simplifies the system management life cycle and provides **OAuth** access tokens as authentication mechanisms for both users and service accounts.
Data	OpenShift provides encryption in transit by default, with the options common to other Kubernetes cloud providers. It also provides encryption at rest for the Linux Core OS storages and for the etcd database.

https://t.me/learningnets

Type	Description
Policy	OpenShift has a rich set of policy-based options, and they can be grouped into the following: • Container policies • Cluster policies Container policies use the **Security Context Constraints** admission controller to manage and govern deployment securely. Cluster Policies are managed via the **Red Hat Cluster Management** system.

***Table 10.17**: OpenShift Deploy*

To protect applications running into the Kubernetes cluster, OpenShift relies on the key elements described in *Table 10.18, OpenShift Protect*:

Type	Description
Container Isolation	Container isolation is achieved following the recommendations of the NIST SP 800-190, including Linux namespaces, Cgroups, SELinux, Capabilities and Seccomp.
Application Isolation	Application isolation is mainly achieved with the following: • OpenShift projects, SELinux namespaces • Security Context Constraints (SCCs)
Network Isolation	OpenShift uses **HAProxy** to handle ingress traffic control and **CoreDNS** for PODs name resolution. Both systems implement **reencrypt**, a solution to decrypts HTTP traffic and reencrypt it without TLS termination.
Access Security	Red Hat OpenShift supports Single Sign-On (SSO) out-of-the-box with Security Assertion Markup Language (SAML) 2.0 or OpenID Connect (OIDC). Access federation is achieved through the **Keycloak** project implementation, an open-source IAM solution.
Observability	Prometheus is the default built-in monitoring solution.

***Table 10.18**: OpenShift Protect*

Red Hat OpenShift Container Platform can run as a cloud-managed service in AWS, Azure, GCP or IBM Cloud, but it can also run on a dedicated OpenShift platform. The third and last tier offered by Red Hat is the **Red Hat OpenShift Platform Plus**, which adds the **Red Hat Advanced Cluster Security for Kubernetes** option on top of the previously described security solutions. This option adds the following features:

- Securing software supply chain
- Kubernetes Security Posture Management (KSPM)

https://t.me/learningnets

- Workload protection

It also enabled compliance capabilities against the most popular security frameworks, such as CSI Benchmark, PCI-DSS and HIPAA; deployments risk profiling to enhance remediation solutions for exposed workloads; a visual representation of the network traffic workflow; a vulnerability management feature for both the cluster and the container images; and an incident response mechanism.

Rancher

Rancher is a Kubernetes orchestrator tool, a multi-clusters management solution with the mission to manage Kubernetes clusters at scale. It was acquired by SUSE in 2020 in an effort to bring to the market **Rancher Prime**, a secure multi-cluster Kubernetes solution. Rancher supports both cloud deployment with AWS, GCP, and Azure, and on-premises deployment with **VMWare vSphere** implementation. The security section is not particularly rich in terms of topics, but Rancher does reference the Kubernetes security best practices and the CIS Benchmark hardening framework. Similarly, like many other Kubernetes distributions, Rancher integrates Open Policy Agent (OPA) via Gatekeeper in the full sense of the **Constraint Framework** discussed in *Chapter 9, Kubernetes Governance*, in the *Policy Engines* section.

The interesting feature that Rancher brings to the discussion, and the main reason why it is part of this chapter, is due to out-of-the-box integration with **NeuVector**. NeuVector is an open-source security framework that runs container operations by providing real-time network visibility, as illustrated in *Figure 10.8, NeuVector*:

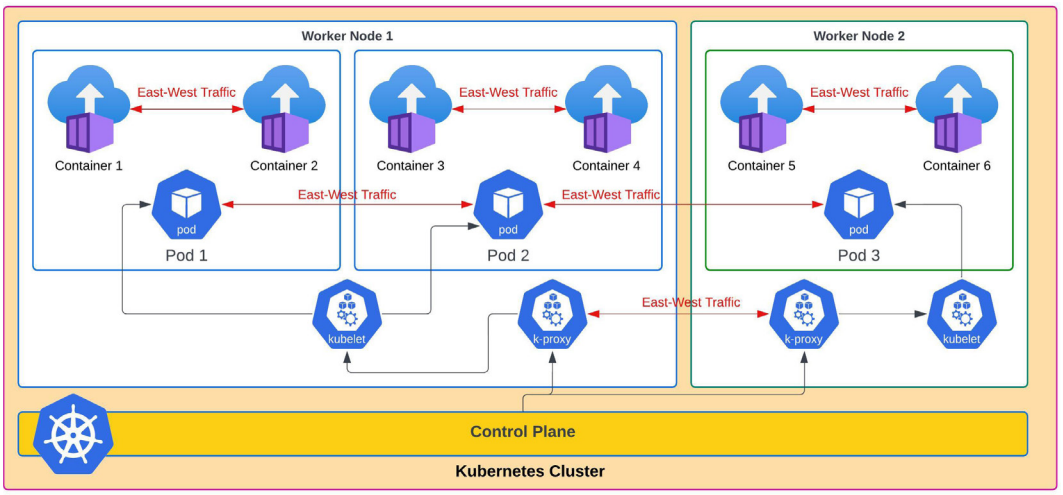

***Figure 10.8**: NeuVector*

https://t.me/learningnets

NeuVector can run runtime security by inspecting **east-west network traffic**; refer to *Chapter 6, Monitoring Container and Security*, *Figure 6.1, Network Traffic*. This enables NeuVector to execute deep packet inspection, a typical firewall feature, by monitoring all the network traffic container-to-container, POD-to-POD, and across worker nodes. Particularly, NeuVector enhances runtime security within the cluster by enabling the features listed in *Table 10.19, NeuVector characteristics*:

Type	Description
DPI	Deep Packet Inspection, also known as DPI (patented by NeuVector) on: • OSI Layers 3 and 4 Port (network and transport layers) • OSI Layer 7 Protocol (application layer) • Processes
Application Behavior	Uses the DPI system to identify expected behaviors and drifts from expected behaviors; NeuVector is capable of automating policy generation based on the network traffic inspection, packet capture and user interaction with the system
Vulnerability Management	The NeuVector Vulnerability Management is compelling: • Scans container image into the registry • Scans container image within the CI/CD pipeline • Inspects containers at runtime • Applies the CIS Benchmark hardening compliance framework against containers, Docker runtime, and the Kubernetes cluster
Zero Trust	Zero Trust is achieved by the integration of the following two features: • **Data Loss Prevention** (DLP) is implemented via the Deep Packet Inspection feature; due to its visibility into the network traffic, NeuVector can detect sensitive data transmission • **Web Application Firewall** (WAF), by leveraging network traffic at the OSI Layer 7

***Table 10.19**: NeuVector characteristics*

Due to the Rancher multi-deployment and managing nature, the focus is on maintaining a consistent security infrastructure across clusters via the integration of the OPA system, which leverages a centralized security policy management system. The Rancher system can handle security in the same web interface for clusters deployed either in the cloud, with services like EKS, AKS, GKE or on-premises, applying a comprehensive multi-cloud approach.

https://t.me/learningnets

Tanzu

Tanzu is the Kubernetes solution created by VMware. It is the answer to the need for unifying Kubernetes clusters management, regardless of where these are deployed. With **Tanzu Mission Control**, users can consistently manage Kubernetes clusters deployed into the cloud, multi-cloud, on-premises or in hybrid scenarios, and it works with vSphere out-of-the-box to build and deliver cloud native applications. **Tanzu Application Platform** has simplified the approach to securing software supply chains by providing an interesting point of view on the matter, as described in *Table 10.20, Tanzu Software Supply Chain*, as part of the **shifting security left** process:

Type	Description
Shift left	Once the **Software Bill of Material** (SBOM) is defined and trusted, it becomes part of the building process; refer to *Chapter 5, Application Container Security*, the *Third-party components* section.
Automation	Achieving full automation of manual activities around dependencies such as the following: • Scanning • Patching • Inventorying • Signing
Guardrails	Identify unsinged container images or vulnerabilities by stopping them from running.

***Table 10.20**: Tanzu Software Supply Chain*

The vulnerability scanning system implemented by Tanzu is based on **Grype**, the **Anchore** unit that scans container images and filesystems for SBOM vulnerabilities; refer to *Chapter 4, Securing Container Images and Registries*, the *Scan and Verify Images* section, for a quick overview of the tool and its use cases. The build process is secured and shielded from the supply chain attack vector by **VMWare Tanzu Build Service**, through the adoption of **Buildpacks**, the **Cloud Native Compute Foundation** (**CNCF**) reference tool for SBOM, which automatically generates inventories for Node.js- and Java-based projects.

Tanzu integrates a system to cryptographically sign container images. This is implemented through the adoption of the **kpack/cosign** tool. **Kpack** is the Kubernetes Native Container Build Service system created by Pivotal Software, which was acquired by VMware in 2019. **Cosign** is a tool used to generate a key pair attributable to a Docker image that works with Kubernetes. The following example code shows how to create a key pair with the **cosign** command (the command will ask users to insert a password):

https://t.me/learningnets

```
$ cosign generate-key-pair k8s://default/my-key-pair
Successfully created secret my-key-pair in namespace default
Public key written to cosign.pub
```

The private key is stored in Kubernetes, and the public key is saved in the same folder where the command was executed. To create a secret out of the cosign key pair, use the following:

```
apiVersion: v1
kind: Secret
type: Opaque
metadata:
  name: my-key-pair
  namespace: default
  annotations:
    kpack.io/cosign.docker-media-types: "1"
data:
  cosign.key: <insert the content of the private key>
  cosign.password: <password used to generate the key>
  cosign.pub: < insert the content of the public key>
```

The **`kpack.io/cosign.docker-media-types: "1"`** annotation is for registries that do not support the **Open Container Initiative** (**OCI**) format. Let us create a service account with the following example code that will be used by the signed image:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: signed-service-account
  namespace: default
secrets:
- name: my-registry-credentials
- name: my-key-pair
imagePullSecrets:
- name: my-registry-credentials
```

https://t.me/learningnets

Based on the preceding information, we can build a container image using the service account created previously, by executing the following code:

```
apiVersion: kpack.io/v1alpha2
kind: Image
metadata:
  name: my-signed-image
  namespace: default
spec:
  tag: <image tag>
  serviceAccountName: signed-service-account
  builder:
    name: my-builder
    kind: Builder
  source:
    registry:
      image: “source-image”
```

Upon creation, to verify that the image built was correctly signed, we can verify the signature with the following command:

```
$ cosign verify --key cosign.pub <image-digest>
```

The example procedure shown in the previous code is automated by Tanzu, which provides users with the capabilities to enforce container images signature across the Kubernetes.

Conclusion

In this chapter, we discussed the most prominent and popular Kubernetes cloud services. We started by discussing the Cloud Native Security Model, with the 4C approach, code, container, cluster and cloud, and then we looked at how those are related to the topics we have already addressed in the previous chapters of this book.

We then analyzed the AWS shared responsibility model and how it impacts the different kinds of **Elastic Kubernetes Service** (**EKS**) services. On the EKS system, we described and explained the four main security topics: infrastructure security, identity and access management, detective controls and incident response.

https://t.me/learningnets

Among AWS cloud competitors, **Microsoft Azure Kubernetes Service** is one of most utilized in the cloud environment. Microsoft's effort to standardize the security requirements as a cloud provider and to allocate those requirements to any Azure service let us define the best course of action to secure the Kubernetes cluster according to Microsoft infrastructure. **Google Kubernetes Engine** (**GKE**) completes the Kubernetes cloud offer by the big three cloud providers, with its focus on hardening and isolation, and the various security mechanisms to strengthen the Linux systems.

Lastly, we discussed how Red Hat OpenShift, Rancher and VMWare Tanzu apply interesting and particular security controls, widening the footprint of their systems to include private and public cloud providers, and hybrid and on-premises solutions.

In the next chapter, we will learn how to secure Helm, the most popular Kubernetes package system.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

https://t.me/learningnets

Chapter 11
Helm Chart Security

Introduction

Helm is similar to **apt** for Linux Ubuntu or **yum** for CentOS, a package manager that can organize Kubernetes objects in a packaged application, such an rpm file, that can be downloaded, redistributed, installed and configured. The equivalent of a Linux **rpm** file in Helm is a **chart**. The main purpose of the Helm package manager is to help in making Kubernetes easier to handle by reducing the intrinsic complexity of the cluster. Any application in Kubernetes can be deployed by defining a set of instructions in a YAML file. With complex systems, the YAML file becomes complex, and deploying the desired application becomes a difficult challenge. With Helm it is possible to deploy an application into the Kubernetes cluster by simply using few command line **helm** commands, as we would do with **yum** or **apt**. We have discussed security solutions like Open Policy Agent or Kyverno in the previous chapters, or system like Jenkins, Harbor, Istio, Falco, HashiCorp Vault, Prometheus or Grafana. Like many others, they can be installed in the Kubernetes cluster by leveraging the related helm charts with a few simple commands.

A command line command does not offer powerful research options and it is visually poor. In 2019 the **ArtifactHUB** was launched, an online Helm Chart repository, just like DockerHub is to Docker. The issue with external package repositories is that software is often distributed or updated to fulfil missing functionalities than to address security issues; this concerns Kubernetes because it introduces vulnerabilities and increases the attack surface of the cluster.

https://t.me/learningnets

Structure

In this chapter, we will discuss the following topics:

- Helm
- Tiller
- Integrity
- IaC Trust
- Chart Scanner
- Dependencies

Objectives

This chapter aims to describe the security concerns around the Helm Chart system, including its various components and their interaction with the Kubernetes cluster. This visibility helps in reducing the security risk concerning expanding Kubernetes' capabilities and utilizing external packages from verified sources.

Helm

Helm charts are files containing declarative Kubernetes resources that are required to install an application in the cluster. If the application brings dependencies, they can be declared and resolved automatically by the system to let the application run properly. Refer to the following figure:

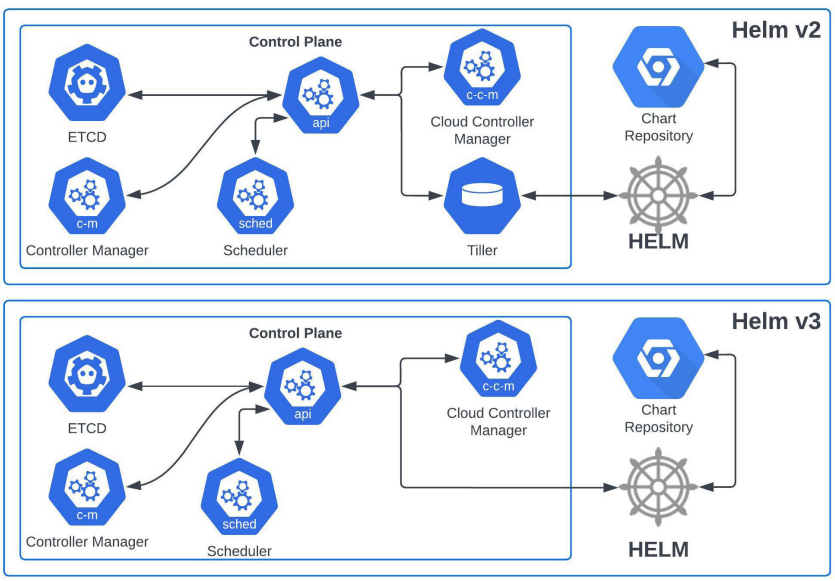

***Figure 11.1**: Helm Architecture*

https://t.me/learningnets

Figure 11.1, Helm Architecture, shows the basic architecture of the system and illustrates the main difference between Helm v2 and v3. In Helm version three, **Tiller** has been deprecated, moving the previous client-server architecture to a client-only architecture. The default Helm installation has no security configurations in place. Although this can be appropriate in certain home lab or local development environments, it is not the recommended approach. With Helm version 2, most security requirements are related to the **Tiller** backend system. In Helm version 3, the communication and security requirements are essentially related to the **kube-apiserver** interaction. Tiller will be discussed in an upcoming section in this chapter.

Helm, in essence, deploys Kubernetes applications as templates via **Yet Another Markup Language** (**YAML**) file manifests containing a set of instructions, including the referred container images and a set of parameters. When considering using ArtifactHub as a chart repository, it is recommended to pay attention to the following three key elements:

- The chart is published by a **Verified Publisher**.
- The chart is tagged as an **Official** release.
- The **Images Security Rating** is a **C** or above.

ArtifactHub provides a vulnerability scanner mechanism integrated in their platform that scans the container images. This is based on **Trivy** (refer to the *Scanning and Verifying Images* section in *Chapter 4, Securing Container Images and Registries*), a tool developed by Aqua Security to generate security vulnerability reports for each chart uploaded. The report has a security rating determined by the scoring system illustrated in *Table 11.1, ArtifactHub Images Security Rating*. Note the missing **E**; this rating is not present in the original scoring model:

Rating	Description
A	No vulnerabilities reported
B	Reported LOW vulnerabilities
C	Reported MEDIUM vulnerabilities
D	Reported HIGH vulnerabilities
F	Reported CRITICAL vulnerabilities
-	Reported vulnerabilities of unknown severity

***Table 11.1**: ArtifactHub Images Security Rating*

Reports are generated periodically and specifically daily for the latest chart version and weekly for previous chart versions. The report could contain multiple scanned images depending on the complexity of the chart and also depending on any dependencies the helm chart could have on external images. Trivy scans specific

https://t.me/learningnets

application dependencies by inspecting the manifest file where the dependencies are declared, with a wide footprint in terms of programming languages supported, including Ruby, Python, PHP, Node.js, .Net, Java, Go, Rust, C/C++, and Elixir. Remember that ArtifactHub is a public container registry, so it should be considered a non-trusted entity as per the topics discussed in *Chapter 4, Securing Container Images and Registries*, in the *Private and Public Registries* section. In this regard, ArtifactHub has released its own chart **artifact-hub** that can be deployed on a private Kubernetes cluster and configured as local private repository. A Helm chart deployment YAML file may contain references to container images either optional or required for the application to start. Controlling the extension of a single deployment can be challenging, for example, the **Istio** chart published by IBM for the IBM Cloud platform contains 14 dependencies, including charts like Prometheus, Grafana, Jaeger, Kiali, and others, which contain other dependencies. As all these are deployed as containers, it is not uncommon to install systems that potentially have no quotas set, and no security contexts or Linux capabilities enabled. It is possible to override the default values by editing the **`values.yaml`** file.

Tiller

Tiller is the back-end system installed by default in the **`kube-system`** namespace, and it listens for requests made by the Helm system. From Helm version 3 onward, the Tiller server is removed and no longer needed; nevertheless, to complete the security spectrum of specific use cases, it is worth discussing the security requirements of such a solution. If installed through **`helm init`**, the options in *Table 11.2, Tiller parameters* should be considered:

Type	Description
--canary-image	This option allows the use of a Canary image; not recommended for production environments, as they can be not stable.
--kube-context	With this parameter enabled, it is possible to specify an alternative cluster.
--tiller-namespace	With this parameter enabled, it is possible to specify an alternative namespace.
--service-account	It enables us to specify the service account for the Tiller system in RBAC clusters.

***Table 11.2**: Tiller parameters*

https://t.me/learningnets

The previous table does not show the element of encryption that will satisfy a secure communication channel between Helm and Tiller. The following installation command sets a **Transport Layer Security** (**TLS**) communication security requirement in place:

```
$ helm init \
--override 'spec.template.spec.containers[0].command'='{/tiller,--storage=secret}' \
--service-account=helm-service-account \
–storage=secret \
--tls-ca-cert=ca.pem \
--tiller-tls \
--tiller-tls-verify \
--tiller-tls-key=key.pem \
--tiller-tls-cert=cert.pem
```

Role-Based Access Control (RBAC) is the default authentication mechanism used by Helm for deployments, as it is also the default for Kubernetes. Kubernetes' RBAC has been discussed in *Chapter 8, Kubernetes Orchestration Security*, and the role types have been described in *Table 8.5, RBAC vs ABAC*. The service account is considered an RBAC resource in Kubernetes, but it should be declared separately from the authentication standpoint:

```
rbac:
  create: true
Component1:
  serviceAccount:
    create: true
    name: component1
Component2:
  serviceAccount:
    create: true
    name: component2
```

https://t.me/learningnets

Tiller works with the permission granted by its service account, and being deployed by default in **`kube-system`**, it is certainly a source of concern.

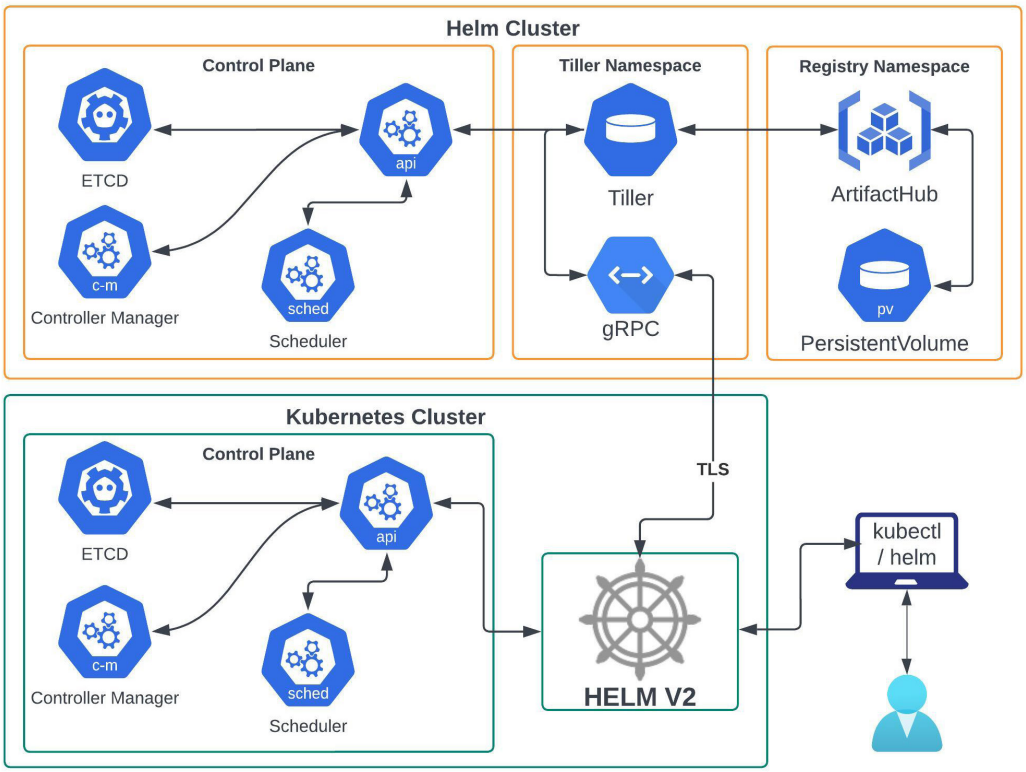

***Figure 11.2**: Tiller Secure Architecture*

The limitations provided by the RBAC mechanism are sufficient to constrain Tiller within the authorizations of the Kubernetes Role and RoleBindings roles, but to enhance security, it is sensible to deploy the Helm server-side system in a different namespace or even better, in a different cluster, via the **`--kube-context`** and **`--tiller-namespace`** parameters. When a new software is needed in the cluster, the installation request is sourced from another cluster where the user is connected and is running helm command via TLS secure communication, as illustrated in *Figure 11.2, Tiller Secure Architecture*. It is very important to enable the TLS protocol during the Helm installation because the gRPC endpoint, which is the Tiller server exposed endpoint, runs without authentication in the first place. No authentication means any process can perform any operation in the kube-system when the endpoint is known. By enabling TLS, Helm is authenticated by the means of the cluster in which it resides, and it reaches the gRPC endpoint over secure communication. Lastly, it is advisable to change the default information storage mechanism from **`ConfigMaps`** to **Secrets**. This involves adding the **`--storage=secret`** parameter to the helm installation command.

https://t.me/learningnets

Integrity

In Helm v3, a new mechanism has been implemented to verify the origin and the integrity of a chart package. This system implements tools like **GnuPG** or Public Key Infrastructure (**PKI**). **Pretty Good Privacy** (**PGP**) is an encryption tool released at the beginning of the '90s that provides cryptographic privacy for data communication. At the time, it was designed to protect email communication. The issue with PGP was the patenting concern around the algorithm used for the encryption mechanism. Based on that experience, PGP Inc. proposed an open version to the community, OpenPGP, which defines cryptographic standards to secure communications. In 2010, Symantec acquired PGP to implement its cryptographic mechanism as part its Enterprise Security Group. Late in 2019, this was acquired by Broadcom.

The Free Software Foundation created in 1999 an open-source version of the encryption tool called **GNU Privacy Guard** (**GNU-PG)** based on the OpenPGP standard. The GNU-PC technical requirements were defined in the RFC4880 document. An RFC is a technical document published by the **Internet Engineer Task Force** (**IETF**). The GNU suffix signifies the open-source character of the software, like many other applications distributed today under the GNU General Public License. In order to verify integrity, Helm charts generate **provenance** record with the **.prov** extension, which are stored alongside the charts. For instance, a chart file like **chartXYZ-0.1.0.tgz** will have a corresponded provenance record, such as **chartXYZ-0.1.0.tgz.prov**. The provenance records are generated at the chart creation with the **–sign** parameter. The **.prov** file is a YAML file manifest that contains information about the chart and cryptographic data, as per *Table 11.3, Provenance file*:

Type	Description
Chart	The chart.yaml document that includes information about the software packages
Signature	The SHA256 signature on the chart file
Body	The body is encrypted with the GnuPG system

***Table 11.3**: Provenance file*

The key for signature must be already in place; for Gnu Privacy Guard users, this should be located in **`~/.gnupg/secring.gpg`**.

Note: GnuPG v2 has the secret moved to the kbx format, so the location would be ~/.gnupg/pubring.kbx. In this scenario, it is necessary to convert the new keyring format kbx to the former keyring format gpg.

https://t.me/learningnets

To apply a signature to an example chart, run the following command:

```
$ helm package --sign --key 'Luigi Aversa' --keyring ~/.gnupg/secring.gpg chartXYZ
```

The preceding command will generate both the **.tgz** and **.prov** files. An example of the **.prov** file is provided as follows:

```
Hash: SHA512

apiVersion: v2
appVersion: "0.21.1"
description: CHartXYZ
name: chartXYZ
type: application
version: 0.1.0

...
files:
  chartXYZ-0.1.0.tgz: sha256:jjggj39g9935fggpge98g095g9ug3g09ugu3f-gug9u0gu05ug8u5u530u8035u08
-----BEGIN PGP SIGNATURE-----

goepjrog££$rref;e;kkb[teyhpo4oy-6{[6-hhl[h[pkht6hkthrbbk0kkkk/Kd
nvHFBNps7hXqSocsg0a9Fi1LRAc3OpVh3knjPfHNGOy8+xOdhbqpdnB+5ty8YopI
mYMWp6cP/Mwpkt7//wTIaLKiWkv2rSfty4yZUs7DJcYB8Udi1prnWf8fgfer658k
pkht6hkthrbbk0kkkk/gP1ecWFMenvHFBNps7hXqSocsg0a9R/c3OpVh3knjPfHN
vicbaFH5AmJCBihBaKJE4R1IX49/+JojnOLIhI3Psd0HBD2bTlsm/rSfty4yZUsD
nvHFBNps7hXqSocsg0a9Vh3knjPfHNGOy8+u9/eyLR0+JcYB8Udi1prnWf8=jtrt
=bfds
-----END PGP SIGNATURE-----
```

To verify the integrity of the chart by the mean of cryptographic signature, run the following:

```
$ helm verify chartXYZ-0.1.0.tgz
```

https://t.me/learningnets

The integrity verification can be also performed at the deployment phase with a command similar to the following:

```
1. $ helm install -generate-name -keyring ~/.gnupg/secring.gpg -verify
   chartXYZ-0.1.0.tgz
```

If the verification fails with a "**sha256 do not match**", the installation is aborted. The preceding command includes the path to the default keyring location.

IaC trust

The previous section has described the signature mechanism provided by the Helm system with the implementation of the GnuPG software for local charts. In this section, we will expand on the topic to achieve package signature verification during the deployment phase. A simple approach to integrity verification is provided by a service called **Keybase.io**, which performs the same kind of steps and applies the same logic discussed in the previous section. For this to work properly, two prerequisites are to be considered:

- An account with Keybase.io
- The **keybase** command-line tool installed

Keybase introduces an easy way to retrieve an organization's PGP public key by querying a web interface: the purpose is to establish a cryptographic identity **chain of trust** for anyone with an internet presence. The concept of retrieving online public keys is not new to computer science; it is what a **keyserver** is supposed to fulfil in computer security. A keyserver is a system that stores and serves public cryptographic keys together with the information about the entity that released the PGP key publicly. The public key can also be stored in the X.509 certificate format. **Keybase.io** is a keyserver in essence. Here are some of the most popular keyservers:

- keyserver.ubuntu.com
- pgp.mit.edu
- keys.openpgp.org

The keyserver allows us to search for a specific identify by email address, key, or fingerprint (a shorter string of bits mapped to a larger set of data), if known. However, it is also possible to add the keyserver to the GPG tool locally and execute command-line queries to retrieve the PGP key. The example we are going to illustrate here goes beyond the simple application of the signature to a helm package; instead, it aims to achieve helm chart package signature verification for any chart that has a PGP public key ascribable to an identity. Grafana is one of the most popular open-source web visualization tools and works beautifully with Prometheus, the metric collector and monitoring system discussed in *Chapter 6, Monitoring Container and*

https://t.me/learningnets

Security, in the *Container Activity* section. Grafana provides a helm chart, and it can be downloaded and installed in Kubernetes with few simple commands:

```
$ helm repo add grafana https://grafana.github.io/helm-charts
$ helm repo update
$ helm install desired-version grafana/grafana
```

Note: We are not going to execute the install command at this stage. It is only to illustrate the simplicity of the helm chart deployment.

Searching Grafana on the Ubuntu keyserver returns a few results; the most recent one has key ID:

```
0E22EB88E39E12277A7760AE9E439B102CF3C0C6
```

In order to utilize the key, GPG has to acknowledge where it is published, so we add the Ubuntu keyserver to the local keyring and request Grafana's key with the following command:

```
$ gpg --keyserver keyserver.ubuntu.com \
--recv-keys 0E22EB88E39E12277A7760AE9E439B102CF3C0C6
gpg: key 9E439B102CF3C0C6: public key "Grafana Labs <engineering@grafana.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1
```

Once the key has been imported in the local keyring, we can export it to a **.gpg** file; we can also skip overwriting any previous exported key if the key is already present in the original path:

```
$ gpg --output previous-key.
gpg --export 0E22EB88E39E12277A7760AE9E439B102CF3C0C6
File 'previous-key.gpg' exists. Overwrite? (y/N) N
Enter new filename: grafana.gpg
```

And verify the integrity of the helm chart by typing the following:

```
$ helm fetch --verify grafana/
grafana --version 6.50.7 --keyring grafana.gpg
Error: failed to fetch provenance "https://github.com/grafana/helm-charts/releases/download/grafana-6.50.7/grafana-6.50.7.tgz.prov"
```

https://t.me/learningnets

The error in the previous command highlights the fact that the **`.prov`** file does not exist, so the Grafana chart has not been signed for verification and cannot be verified. Let's repeat the exercise, adding the HashiCorp repo to helm:

```
$ helm repo add hashicorp https://helm.releases.hashicorp.com
"hashicorp" has been added to your repositories
$ helm search repo hashicorp
NAME                    CHART VERSION APP VERSION DESCRIPTION
hashicorp/consul        1.0.4         1.14.4      Official HashiCorp Consul Chart
hashicorp/terraform     1.1.2                     Install and configure Terraform Cloud Operator ...
hashicorp/vault         0.23.0        1.12.1      Official HashiCorp Vault Chart
hashicorp/waypoint      0.1.17        0.10.5      Official Helm Chart for HashiCorp Waypoint
```

The HashiCorp PGP public key is as follows:

```
c874011f0ab405110d02105534365d9472d7468f
```

To add the HashiCorp public key to the local keyring and export it to a **`.gpg`** file, use the following commands:

```
$ gpg --keyserver keyserver.ubuntu.com \
--recv-keys c874011f0ab405110d02105534365d9472d7468f
    gpg: key 34365D9472D7468F: public key "HashiCorp Security (hashicorp.com/security) <security@hashicorp.com>" imported
    gpg: Total number processed: 1
    gpg:               imported: 1
```

PGP Keys actually present in the keyring can be listed with the following:

```
$ gpg --list-keys
/Users/luigiaversa/.gnupg/pubring.kbx
------------------------------------
pub   rsa3072 2023-01-06 [SC] [expires: 2025-01-05]
      0E22EB88E39E12277A7760AE9E439B102CF3C0C6
uid           [ unknown] Grafana Labs <engineering@grafana.com>
```

https://t.me/learningnets

```
7. sub   rsa3072 2023-01-06 [E] [expires: 2025-01-05]
8.
9. pub   rsa4096 2021-04-19 [SC] [expires: 2026-04-18]
10.      C874011F0AB405110D02105534365D9472D7468F
11. uid                 [ unknown] HashiCorp Security (hashicorp.com/
    security) <security@hashicorp.com>
12. sub   rsa4096 2021-04-21 [S] [expires: 2026-04-20]
13. sub   rsa4096 2021-04-19 [E] [expires: 2026-04-18]
```

To export the Public PGP key to a GPG file, we can use the same command used with the Grafana case but while naming the output file appropriately to skip the overwrite task:

```
1. $ gpg --output hashicorp.gpg --export c874011f0ab405110d02105534365d
   9472d7468f
```

Once the HashiCorp GPG key has been exported from the HashiCorp PGP key, we can verify any HashiCorp helm chart present in the repo:

```
1. $ helm fetch --verify hashicorp/
   terraform --version 1.1.2 --keyring hashicorp.gpg
2. Signed by: HashiCorp Security (hashicorp.com/security) <security@
   hashicorp.com>
3. Using Key With Fingerprint: C874011F0AB405110D02105534365D9472D7468F
4. Chart Hash Verified: sha256:9115fecf667f3d41a5a1c5833de948ef-
   699321a6165be7bebe4976a7279702d5
```

Successful verification indicates that HashiCorp has signed the **Terraform** helm chart at the creation phase, as illustrated in the previous section of this chapter, so the repo hosts not only the **`.tgz`** file but also the **`.prov`** file. Readers familiar with tools like Terraform will understand that the popular HashiCorp tool satisfies the implementation of a **Infrastructure as Code** (**IaC**) solutions. Infrastructure as Code is the provisioning of infrastructure using configuration files that contain the IT infrastructure specifications either on-premises or in the cloud. To help manage the various types of the IT infrastructures, Terraform uses a mechanism called providers. A provider is a set of resources assigned to a specific vendor, which details modules and parameters for any service or resource that can be deployed. For example, the AWS provider has modules for the **Virtual Private Cloud** (**VPC**) service, the **Security Group** (**SG**) service, and the **Elastic Kubernetes Service** (**EKS**); likewise, Microsoft Azure, Google Cloud, and Kubernetes have their own set of modules that can be leveraged to deploy the related services. The Terraform Registry, where the

https://t.me/learningnets

providers are stored, also publishes vendor- or application-specific providers, such JFrog Artifactory, Aqua Security, Cisco, Cloudflare, and the Elastic stack.

In 2017, HashiCorp released the first version of the **`terraform-provider-helm`**, which is on release 2.8.0 today. The description of the various configurations and parameters is out of scope of this book, but it is worth noting the options as per *Table 11.4:*

Type	Description
verify	If set to true, this optional parameter will enable chart integrity verification, and therefore, is expecting to find the **.prov** file alongside the **.tgz** package.
keyring	This is the path to the public key **.gpg** file used to execute integrity verification on the chart; this option can be used only if the previous option is true.

***Table 11.4**: Terraform Helm Parameters*

Therefore, the following example code will automate helm chart infrastructure as code deployment, also providing chart integrity verification by the means of PGP Public Key:

```
1. resource "helm_release" "vault" {
2.   name             = "vault"
3.   create_namespace = true
4.   namespace        = "the-assigned-namespace"
5.   repository       = "https://helm.releases.hashicorp.com"
6.   chart            = "vault"
7.   version          = "0.23.0"
8.   verify           = true
9.   keyring          = "hashicorp.gpg"
10.}
```

When thinking of a DevSecOps scenario and **Secure Software Development Life Cycle** (**SSDLC**), which entails Kubernetes clusters and additional software to be managed on top of the orchestrator via CI/CD pipelines, the option to add a layer of security via package signature verification enhances the security posture and simultaneously reduces the attack surface, enabling the Web of Trust.

https://t.me/learningnets

Chart scanner

In *Chapter 7, Kubernetes Hardening*, in *Table 7.15, External Hardening Tools*, the recommended **Static Application Security Testing** (**SAST**) solution is the code security inspection tool **Checkov**. Checkov can be used in various environments like Jenkins, Docker, GitLab CI, Bitbucket, and with Terraform. Checkov also supports systems like Ansible, AWS SAM, CloudFormation, Serverless configuration and **Software Composition Analysis** (**SCA**). In Helm Charts, Checkov supports offline scanning as well, meaning there is no need to install the chart to execute the scanning. With our example on the HashiCorp chart in the previous section, after adding the chart repo, it is possible to inspect it with the following commands:

```
1. $ helm repo add hashicorp https://helm.releases.hashicorp.com

2. $ helm inspect values hashicorp/vault > export-vault-values.yaml
```

This will generate the HashiCorp Vault correspondent YAML file simply from the Vault chart available in the repo. This file is, however, not verifiable by Checkov; it needs to be templated into a Kubernetes framework, similar to the files used by Kubernetes itself for deployment, with the following command:

```
1. $ helm template hashicorp/vault -f export-vault-values.yaml > k8s-
   vault-template.yaml
```

This file is now ready for the Checkov scan:

```
1. $ checkov -f k8s-vault-template.
   yaml --framework kubernetes --quiet --skip-check CKV_K8S_21
   > checkov-vault-scan
```

The **`--quiet`** parameter will silence the output on the checks that are passed, focusing only on the failed one. Among the charts in the HashiCorp helm chart repo, Vault is at version 0.23.0, which was released in November 2022, and it is the latest available at the time of writing this chapter. The checkov tool scanned the **`k8s-vault-template.yaml`** with 279 checks, passed 234, and failed 45.

Note: CKV_K8S_21 alert on the default namespace is skipped because Helm manages the namespace on its own, and it would create false positives.

It would not be possible to list all the failed checks, but the most relevant ones are listed in *Table 11.5, Checkov Vault Failed Checks*:

Check	Resource	Description
CKV_K8S_12	POD	Memory requests is not set
CKV_K8S_20	POD	The **allowPrivilegeEscalation** option should be disabled

https://t.me/learningnets

Check	Resource	Description
CKV_K8S_13	POD	Memory limits is not set
CKV_K8S_40	POD	To avoid conflict with the host systems containers should run as a high UID
CKV_K8S_10	POD	CPU requests is not set
CKV_K8S_29	POD	Security context should be applied to your pods and containers
CKV_K8S_38	POD	The Service Account Tokens should be mounted only where necessary
CKV_K8S_23	POD	Root containers admission should be minimized
CKV_K8S_43	POD	Container Images should use digest
CKV_K8S_11	POD	CPU limits is not set
CKV_K8S_11		CPU limits is not set
CKV_K8S_155	Cluster	Validating and mutating admission configurations should be minimized
CKV_K8S_31		The seccomp profile should be set to docker/default or runtime/default
CKV_K8S_31	StatefulSet	The seccomp profile should be set to docker/default or runtime/default

***Table 11.5**: Checkov Vault Failed Checks*

There is also a second option via the use of the Aqua Security tool **Trivy**, which also provides good results. The tool has a specific command option **config** to scan folders that contain configuration file structures:

```
$ helm inspect values grafana/grafana > export-grafana-values.yaml
$ helm template grafana/grafana \
    -f export-grafana-values.yaml > k8s-grafana-template.yaml
$ trivy config k8s-grafana-template.yaml > trivy-grafana-scan
2023-05-05T19:31:35.564+0100 INFO Misconfiguration scanning is enabled
2023-05-05T19:31:36.761+0100 INFO Detected config files: 1
```

The scanning results of the Grafana helm chart are summarized in the following screen output:

```
k8s-grafana-template.yaml (kubernetes)
==========================================
Tests: 165 (SUCCESSES: 141, FAILURES: 24, EXCEPTIONS: 0)
Failures: 24 (UNKNOWN: 0, LOW: 20, MEDIUM: 4, HIGH: 0, CRITICAL: 0)
```

https://t.me/learningnets

The format of the vulnerability result is similar to the following example code:

```
MEDIUM: Container 'grafana' of Deployment 'release-
name-grafana' should set 'securityContext.
allowPrivilegeEscalation' to false
══════════════════════════════════════════════════════════════════════
A program inside the container can elevate its own privileg-
es and run as root, which might give the program
control over the container and node.

See https://avd.aquasec.com/misconfig/ksv001
──────────────────────────────────────────────────────────────────────
 k8s-grafana-template.yaml:197-249
──────────────────────────────────────────────────────────────────────
  197 ┌         - name: grafana
  198 │           image: "grafana/grafana:9.3.6"
  199 │           imagePullPolicy: IfNotPresent
  200 │           volumeMounts:
  201 │             - name: config
  202 │               mountPath: "/etc/grafana/grafana.ini"
  203 │               subPath: grafana.ini
  204 │             - name: storage
  205 └               mountPath: "/var/lib/grafana"
...
──────────────────────────────────────────────────────────────────────
```

Note that the output of the preceding command has been modified to accommodate publishing requirements.

Dependencies

To deploy an application into Kubernetes without Helm, you need to write multiple manifest files to bring the application up and running. Many complex systems have several dependencies and often require more than one container in order to work properly. A WordPress chart has 20 templates, Grafana has 35 templates, and the longer the list, the higher the chance that a misconfiguration has occurred. Many

https://t.me/learningnets

things can happen that could potentially harm the cluster: no quotas set, run as root set to true, a privilege escalation issue, or a missed capability constraint. Helm has three types of dependencies, as listed in *Table 11.6, Helm Chart Dependency Types*:

Type	Description
Chart	A chart can recall another chart to fulfil installation purposes and streamline the deployment process in subpackages (nested charts).
Library	Available from Helm version 3, it is possible to declare libraries directly into the chart. In Helm, libraries are not libraries in the common Linux sense; they are libraries packed as charts.

***Table 11.6**: Helm Chart Dependency Type*

The chart **dependencies** declaration format in Helm has no substantial difference between version 2 and version 3. What is changed is the file where those dependencies are declared: in version 2 the file is **requirements.yaml** while in version 3 the dependencies are now declared in the **`Chart.yaml` file,** as the following Prometheus chart file illustrates:

```
1. dependencies:
2. - name: alertmanager
3.   version: 0.24.*
4.   repository: https://prometheus-community.github.io/helm-charts
5. - name: kube-state-metrics
6.   version: 4.24.*
7.   repository: https://prometheus-community.github.io/helm-charts
8. - name: rometheus-node-exporter
9.   version: 4.8.*
10.  repository: https://prometheus-community.github.io/helm-charts
11.  version: 4.8.*
```

In addition to the chart interdependence and package declaration, the Helm v3 system brings the capability to declare libraries into the chart configuration file:

```
1. dependencies:
2.   - name: any-library
3.     version: 0.0.1
4.     repository: quay.io
```

https://t.me/learningnets

In *Chapter 4, Securing Container Images and Registries*, we briefly discussed dependencies scanning, the role that **Software Composition Analysis** (**SCA**) has in securing container images, and how the **National Institute of Standards and Technology** (**NIST**) has recently coded the **Software Bill of Materials** (**SBOM**) to tackle the rising issue of supply chain attacks. While **Grype** and **Syft**, the two command-line tools mentioned in the same chapter, created by **Anchore,** are excellent solutions to scan container images and should be considered to enhance the security posture of the environment, in Kubernetes and specifically with Helm Chart, those are less efficient. There is a nice tool called **KubeClarity**, a well-built SBOM detection and management system that integrates the two mentioned tools plus **CycloneDX-gomod** and **Aqua Security Trivy**.

To clarify, Software Composition Analysis is similar to **Static Application Security Testing** (**SAST**) in the broader spectrum of application security, but SCA is focused on the libraries or dependencies that the applications need to run. So, if a Node.js application needs to access a MySQL database back-end system, it would use the Node.js MySQL module. The MySQL module is considered a dependency of the main Node.js application, which will be scanned for vulnerabilities by the SCA system. In contrast, SBOM is the standard industry format report generated by the SCA tool in a XML format. The **Open Web Application Security Project** (**OWASP**) has created **CycloneDX**, a cyber risk reduction framework that aims to standardize the security process of utilizing third-party software into the software development life cycle, to address the security threats generated by the supply chain. CycloneDX list a few types of bills of materials, as shown in *Table 11.7, CycloneDX Bill of Materials*:

Type	Description
Software Bill of Materials	SBOM is the standard format to track and create an inventory of the software components dependencies.
Hardware Bill of Materials	HBOM is the standard format track and create an inventory of the hardware components for embedded devices, such as Internet of Things (IoT) devices.
Software-as-a-Service Bill of Materials	SaaSBOM shifts the focus on cloud-native applications, where the dependency could be a misconfigured cloud service.
Operations Bill of Materials	OBOM is the inventory of configurations and runtimes.
Vulnerability Disclosure Reports	VDR is a standard report aiming to communicate and highlight the vulnerabilities affecting the components.
Vulnerability Exploitability eXchange	VEX reconcile the vulnerabilities of the components in the context of the application in which they are deployed.

***Table 11.7**: CycloneDX Bill of Materials*

https://t.me/learningnets

The OWASP CycloneDX lists several tools that can help with SBOM, but not so many can interact with Helm. There are a couple of ways to extract a SBOM report from a Helm chart, and we can use, as an example, the Grafana helm repo already added to our system and Checkov. First, the helm chart must be extracted and decompressed; the **fetch** option allows this in one single command:

```
$ helm fetch grafana/grafana --destination /save/path/grafana --untar
```

Checkov can scan the grafana helm chart from the parent directory using the output format **cyclonedx**, as per the following example:

```
$ checkov -d grafana -o cyclonedx > sbom-grafana.xml
```

The inspection of the **`grafana.xml`** highlights the following example output that is consistent with the result produced by Checkov when scanning the Grafana Helm Chart:

```
...
<component type="application" bom-ref="pkg:helm/cli_repo/grafana/grafana/templates/clusterrole.yaml/ClusterRole.default.release-name-grafana-clusterrole@sha1:8faaaaa4208e987a1fe4cf37fc63c5e7b99393ee">
  <name>ClusterRole.default.release-name-grafana-clusterrole</name>
  <version>sha1:8faaaaa4208e987a1fe4cf37fc63c5e7b99393ee</version>
  <hashes>
    <hash alg="SHA-1">8faaaaa4208e987a1fe4cf37fc63c5e7b99393ee</hash>
  </hashes>
  <purl>pkg:helm/cli_repo/grafana/grafana/templates/clusterrole.yaml/ClusterRole.default.release-name-grafana-clusterrole@sha1:8faaaaa4208e987a1fe4cf37fc63c5e7b99393ee</purl>
</component>
...
```

The SBOM standard is the last frontier of the supply chain security, and with the OWASP community backing the process, it will evolve fast. One of the suggested tools to consume SBOM report is **Dependency-Track**, a platform that provides Supply Chain Component Analysis insights, with a dashboard to visualize projects and components in an organized and efficient fashion. The vulnerabilities are then structured and managed with risk-based scoring system and policy management.

https://t.me/learningnets

Conclusion

In this chapter, we discussed the importance of securing the Helm Chart system. We analyzed the Helm system architecture, and the changelog between version 2 and version 3. We also discussed how Tiller is still a topic that deserves some security clarifications due to its popularity.

We then discussed how to verify the integrity of the helm chart packages, retrieving the PGP public key issued by the publishers from keyservers and ingesting it in our GPG keyring to acknowledge helm chart signature as guarantee of authenticity. Further on, we demonstrated how it is possible to create a circle of trust within the infrastructure as code mechanism to verify helm chart integrity.

In the last part of this chapter, we discussed why scanning Helm Charts for vulnerability is important and which tools are more appropriate for this task. We also provided an overview of how Helm processes dependencies and why we should pay attention to them when securing the Kubernetes cluster environment.

Lastly, we discussed in depth the Software Bill of Materials (SBOM) and explored how the security community is effortlessly trying to standardize the Software Composition Analysis security scans through the CycloneDX project, to enhance security for the supply chain issue.

In the next and the last chapter of this book, we will learn about the Service Mesh and understand why it is important to Kubernetes and what the related security best practices are.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

https://t.me/learningnets

Chapter 12
Service Mesh Security

Introduction

In a microservices architecture scenario, each individual component of the application benefits from independence from the others. The **User Interface** (**UI**) can be written in Angular for example, while the back end can rely on Go or Java,

https://t.me/learningnets

the cache system can use Redis, and the database can store data in PostgreSQL or MySQL or MongoDB. Refer to the following figure:

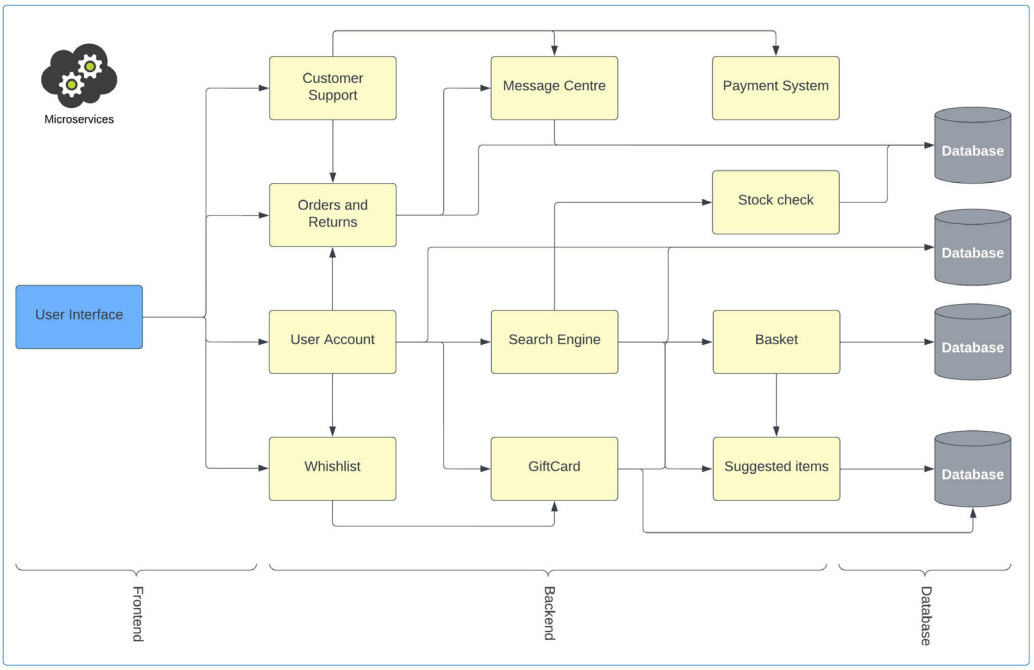

***Figure 12.1**: Microservices concepts*

Often, the microservices architecture goes beyond the simple tier logic subdivision, for instance, each business function is a standalone application, coded with the desired programming language. Each application can be deployed, updated, or dismissed without affecting the overall functionality of the system, as illustrated in *Figure 12.1, Microservices concepts*. While this architecture solution provides great flexibility and streamlines the software development life cycle, allowing the creation of independent functional small teams focused on a single service, it also implies that these services need to communicate to each other to gather the information that they need to perform their tasks. For example, a simple purchase process may involve the search service, the stock check service, the basket service, the payment service and all the related database transactions. The independence obtained from the application development standpoint does not reflect the challenge for a services-to-service network communication system. While the logic governing networking

https://t.me/learningnets

can be included in each component of the application or smoothened out by modules and plugins, it often represents an obstacle that can affect the user experience and generate serious security concerns.

Structure

In this chapter, we will discuss the following topics:

- Overview
- Architecture
- Container Network Interface
- Envoy security
- Secret Discovery Service
- Mutual TLS
- Istio Security
- Zero-Trust Networking

Objectives

This chapter aims to provide security insights for organizations that consider implementing a service mesh on top of their Kubernetes infrastructure, and it explains the implications of such a choice. We will discuss the technology behind service meshes and the intrinsic security values they add to the security posture of the whole cluster. We will also look at the drawbacks that such benefits could generate.

Overview

A service mesh works as an infrastructure network layer in a Kubernetes cluster. This new component installs a dedicated control plane, the **Service Mesh Control Plane**, which works as a platform adaptor interfaced through the **Container Network Interface** (**CNI**) on the nodes, as illustrated in *Figure 12.2, Service Mesh Overview*. A service mesh unloads the Kubernetes cluster from traffic network management, but in contrast, it generates much more computational load because it augments

https://t.me/learningnets

the resources needed to deploy the sidecars onto the nodes. Refer to the following figure:

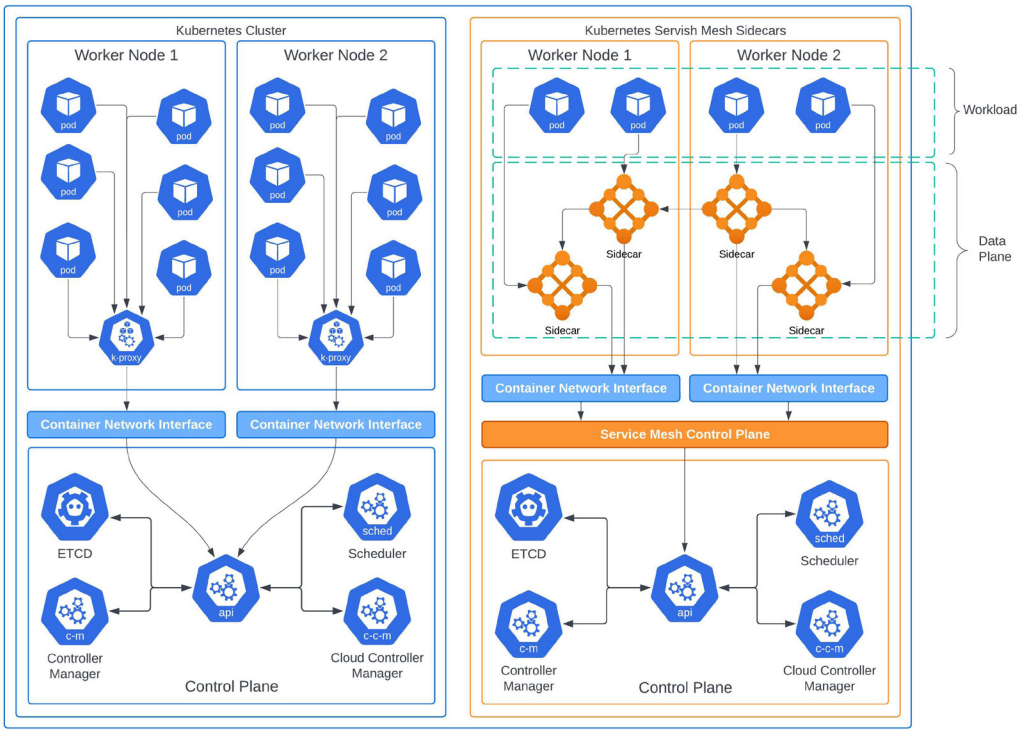

***Figure 12.2**: Service Mesh Overview*

Note: The kube-proxy object has been purposefully removed from the right side of *Figure 12.2, Service Mesh Overview*, because it does not serve the communication with the service mesh, at least not functionally. However, the object is still present to satisfy the necessary communication between the Kubernetes control plane and the worker nodes intended as hosts.

The set of proxies or **sidecars** collectively creates the **Service Mesh Data Plane**, which is not a single element but a high-level nomenclature to identify resources that are not included in the Service Mesh Control Plane. Readers with networking knowledge will be familiar with the concept of a network proxy system, and the name HAProxy is likely known to most of them. HAProxy has been around for more than 20 years now, and although it is one of the most stable, reliable and consistent proxy servers with features like high availability, load balancer, and reverse proxy, it is not the chosen system to be the sidecar because it was not designed for microservices.

The complexity of a system like Kubernetes has technical requirements that go beyond the classic client-server approach or the single file configuration systems. It needs a

https://t.me/learningnets

modular, scalable, API-based system like **Envoy Proxy**. In the Kubernetes cluster, each node runs a kube-proxy system that processes the network communication between the Kubernetes control plane and the host, implements network rules, routes communication to the PODs and therefore to containers, and forwards traffic to virtual IP addresses of the Kubernetes **Service** objects, as explained in *Chapter 7, Kubernetes Hardening*, in *Figure 7.3, kube-proxy*.

Kubernetes does not directly expose any POD, so any request reaching the load balancer in a north-south traffic network scenario will be able to establish a successful connection when kube-proxy routes it via the node's kernel directly to the container's endpoint, also known as **iptables mode**, it also uses the operating system installed on the host as a packet filtering mechanism, such as **Netfilter**.

There is no straightforward methodology to secure load balancers created by the Kubernetes Service. In some cases, we can whitelist IP ranges blocks that are allowed to connect to the cluster via the `loadBalancerSourceRanges`. When this is not configured, the network traffic is supposed to be transparent. That is one of the reasons why, at the cloud level, cloud providers usually implement the load balancer externally to the Kubernetes cluster. Furthermore, when a component of the application attempts to access another component within the cluster in an east-west traffic logic, it starts a lookup request to identify the destination IP address of the connection. The lookup mechanism is also known as **service discovery**.

Architecture

A service mesh is essentially composed of two elements: the control plane and the data plane. The control plane provides configuration and management of the network architecture, while the data plane handles or operates the network traffic routing. The service mesh brings to the Kubernetes cluster many benefits, the most relevant ones being faster application deployment, reduced overhead on service-to-service communication, easier diagnostic capabilities due to localized network errors, and security features like authentication, authorization and encryption supported out-of-the-box.

Cloud-native applications mirroring the microservices model running in containerized environments and sustained by a container orchestrator like Kubernetes benefit immensely by the adoption of the service mesh model, which creates a networking layer to harmonize and unify the network communication model. In the service mesh architecture, as per *Figure 12.3, Service Mesh Microservices*, the service-to-service communication model is replaced by a new infrastructure network layer. Technically, a service mesh is a layer of interconnected proxy systems. Each service of the application has its own service mesh system running in parallel, each microservice talks to its own service mesh, and the service mesh routes the

https://t.me/learningnets

requests or responses to the service mesh of another component, filtering the communication.

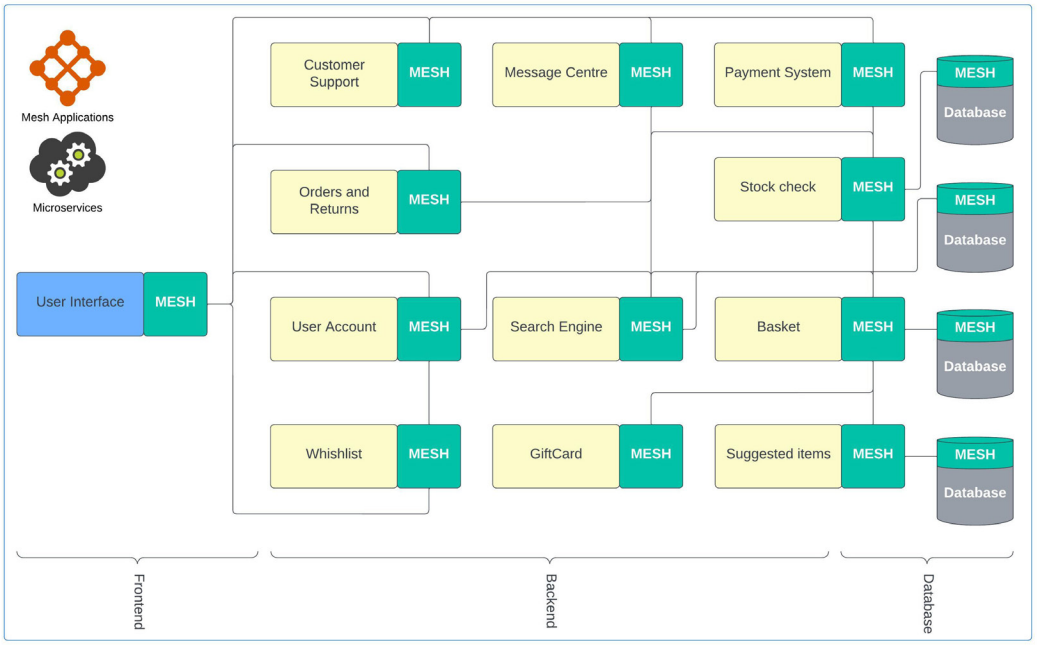

***Figure 12.3**: Service Mesh Microservices*

Service mesh components are sometimes referred to as **sidecars**, as they run alongside each service, not within them. In a service mesh scenario, expanding the application with new features and new services does not add complexity in terms of networking to the entirety of the application because the only communication needed would be that of the microservice with the service mesh. A breakdown of the main service mesh components is described in *Table 12.1, Service Mesh Components*:

Component	Description
Control Plane	A POD deployed into the that governs the network communication applied through the Container Network Interface.
Data Plane	The data plane is not a specific component on its own; it is the collection of all the sidecars considered together.

https://t.me/learningnets

Component	Description
Sidecars	A proxy server deployed alongside each POD that bridges the communication between the following: • The microservice attached to the sidecar and the other sidecars • The microservice and the Worker Node • The microservice and the control plane POD • The microservice and the cluster • The microservice and external requests routed into the POD
SMI	Service Mesh Interface is a Cloud Native Compute Foundation project to standardize the interface connection on Kubernetes. The standardization would affect the following elements: • Network Traffic Management • Network Traffic Policy • Network Traffic Telemetry • A set of common APIs requirements will allow the SMI to be provider agnostic, meaning that it will not be linked to any specific cloud provider or technology.

***Table 12.1**: Service Mesh Components*

There are several service mesh projects that can be deployed in a Kubernetes cluster: **Istio** (the most popular), **Linkerd**; **Kuma**; **Nginx Service Mesh**; **Open Service Mesh** created by Microsoft and donated to Cloud Native Computing Foundation; **Consul Connect** by HashiCorp; **Maesh**; and **Network Service Mesh**, a hybrid and multi-cloud IP service mesh. Microsoft was not the only cloud provider to have created an ad hoc service mesh system for its own cloud service; there is another popular system created by Amazon, called **AWS App Mesh**.

Container Network Interface

Container Network Interface (**CNI**) is a Cloud Native Computing Foundation project, a Kubernetes network interface designed to overcome some of the limitations imposed by the kube-proxy system and to expand the features that the cluster can offer by plugging in a third-party network component mechanism. The main purpose of CNI is to provide control over network communication to monitor network activities and simultaneously reduce the burden of generating manual network configurations by the implementation of network policies; refer to *Chapter 9, Kubernetes Governance*, the *Network Policies* section.

https://t.me/learningnets

In essence, a CNI is a pluggable module that works on top of the Linux container network interface. As a reminder, containerized platforms draw physical resources from the underlying host, so any mechanism working on top of those is an additional abstraction layer of the operating system. A Container Network Interface can be integrated using two network models: an encapsulated model like **Virtual eXtensible LAN (VxLAN)**, and an unencapsulated model like **Border Gateway Protocol** (**BGP**). A visual representation of the difference between the two models is provided in *Figure 12.4, CNI Models*:

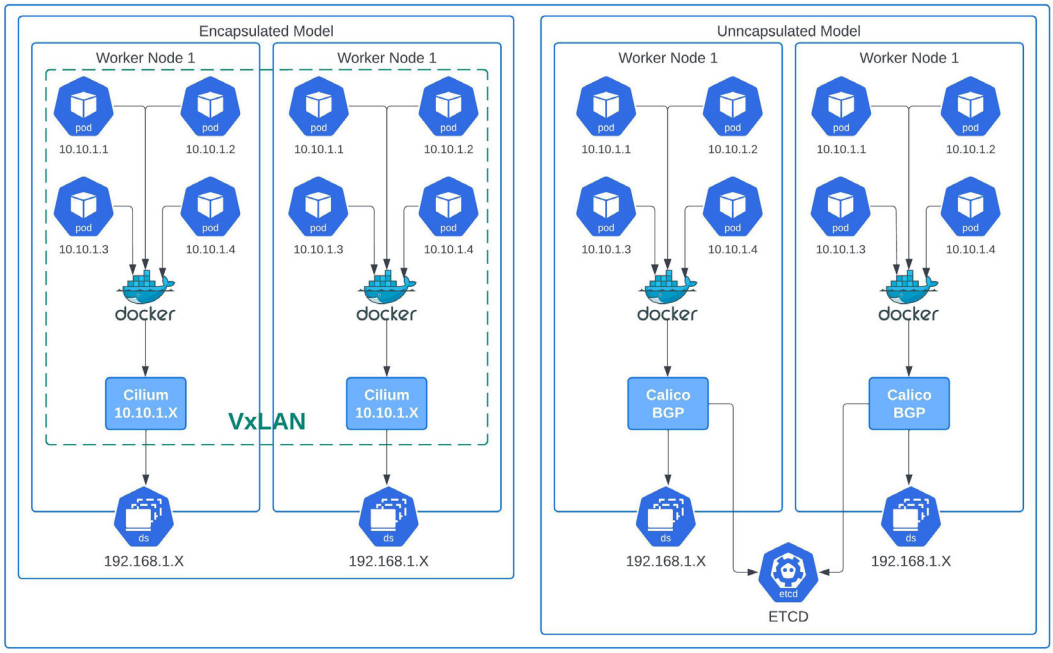

***Figure 12.4**: CNI Models*

The differences between the two network mechanisms are highlighted in *Table 12.2, CNI Network Models*, but in essence, a VxLAN is a network protocol that tunnels network traffics at the Data-Link Layer 2 over a Network Layer 3, while BGP is a protocol designed to exchange information on the routing paths of autonomous systems at the Network Layer 3 only.

https://t.me/learningnets

Type	Description
Encapsulated	• The encapsulation mechanism happens at the OSI Layer 2, Data Link over an existing OSI Layer 3, Network. • The Data Link layer is isolated, and the encapsulation creates an IP header. This translates data from the control plane to the nodes Media Access Control (**MAC**) addresses, distributing encapsulated data between the nodes in the cluster. • A popular encapsulation model is Internet Protocol Security (IPsec). • A network bridging connection is created between the PODs and the nodes through the container runtime engine.
Unencapsulated	• The unencapsulated mechanism happens at the OSI Layer 3, the Network layer, directly. • Missing the Data Layer, the worker nodes need to handle the routing within the hosts. • The routing is executed by the BGP protocol, which distributes information to the PODs, while the container runtime engine defines the communication between the PODs.

***Table 12.2**: CNI Network Models*

The most common Container Network Interface providers are **Flannel**, **Canal**, **Cilium**, **Calico** and **WeaveNet** by WeaveWorks. In *Chapter 6, Monitoring Container and Security*, we discussed another WeaveWorks tool in the *Topology Visualization* section: WeaveScope. It is worth highlighting that the Flannel container network interface does not support network policies and the Canal container network interface does not support encryption. Some of the most popular cloud providers have created their own solutions, such as the AWS **Elastic Kubernetes Service** (**EKS**) **VPC-CNI**, Microsoft with the **Azure CNI**, and Google with the **GKE CNI** in order to implement network policies and service meshes derived by their own cloud services, but they also provide the capability to integrate external CNI drivers with different flavors. For instance, AWS and Google both support the Calico CNI, while Azure is developing a new Azure CNI based on Cilium, which is in preview at the time of writing this chapter.

https://t.me/learningnets

In the Kubernetes network environment, it is common to talk about the **overlay network**. The overlay network is intended as the capability of the cluster to communicate to a POD by IP addresses, and all the cluster services rely on the overlay network to work properly. In order for the IP-based communication to happen, the kube-proxy system invokes the **CoreDNS** system, which aims to resolve the **Domain Name System** (**DNS**) name to an IP address. Once the name has been resolved to an IP address, CoreDNS creates the correspondent iptables rules on the host operating system to route the traffic network accordingly. The CoreDNS is indeed one of the processes of the cluster service-to-service communication mechanism.

A spoofing attack is a way by which a system, application, or a person masquerades as a trusted entity attempts to gain an illegitimate advantage. **DNS Spoofing**, also known as DNS Cache Poisoning, is when corrupted data is legitimated by DNS, redirecting the network traffic to an IP address that is not the true resolver of a domain name. Kubernetes is not immune to this kind of an attack, and the network communication can be compromised if the target of the attack is the CoreDNS deployment, in a scenario where the CNI is not installed. The CoreDNS system that receives DNS queries attempts to resolve requests from within the cluster, using the local configuration; if a request cannot be satisfied, the name server will query the authoritative name servers configured, asking if they know which IP corresponds to that domain.

A POD Escaping is eventually possible (refer to *Chapter 8, Kubernetes Orchestration Security*, the *POD Escaping Privilege Escalation* section for reference) when the Linux Capability `CAP_NET_RAW` is enabled into the POD by means of an attack technique called **ARP Spoofing**. ARP stands for Address Resolution Protocol, and it is accessible along with the **Internet Control Message Protocol** (**ICMP**) through the `NET_RAW` capability. The ARP Spoofing attack exploits the correlation between the IP address and the physical address of the network machine by impersonating a fake MAC address. With the implementation of a Container Network Interface, the cluster is not only taking advantage of the network policy capability but also of a virtual layer of IPs dynamically assigned to PODs, reducing the overhead on the network cluster management, plus securing most of the network communication mechanism out-of-the-box. For example, the Calico CNI can prevent both ARP Spoofing and IP Spoofing, and the Calico Cloud service also provides a DNS Dashboard aiming to visualize DNS traffic flow of data.

Envoy security

Envoy proxy is the sidecar in the service mesh; it was written to satisfy and support microservices or distributed infrastructure systems, and while still supporting the single file configuration mode, it also supports the gRPC or the API-based configuration approaches. It is the most popular, modern proxy server adopted by service mesh systems like Istio, Kuma, and Consul Connect, and many cloud

https://t.me/learningnets

providers. Envoy works at the OSI Model Layer 7, the application layer; it has been designed to bring network transparency, in which applications have no knowledge of the network structure, by virtually working with any programming language. *Table 12.3, Envoy Features*, provides a high-level overview of the main characteristics available in the system, and the observability feature is one of the most significant ones because it provides powerful tools to diagnose the cluster network behavior.

Type	Description
HTTP L7 Routing	The routing system can route requests based on various factors, such as runtime values, content type, path, and authority.
HTTP L7 Filter	The filter can be implemented into the HTTP connection system to perform routing and forwarding, rate limiting, and buffering.
HTTP/2	Any combination of HTTP/1 and HTTP/2 can be bridged.
OSI L3 and L4	Listener mechanism that performs network layer filters such as Read, Write and Read/Write.
gRPC	Remote Procedure Call is a high-performance cross-platform framework used by Google before becoming open-source to connect microservices at scale.
Edge Proxy	It's a type of proxy that sits logically at the edge of the network, providing visibility into networking traffic management.
Discovery Service	It provides dynamic updates of listening sockets, HTTP routing, hosts and backends, and cryptographic communications.
Advanced Load Balancer	Currently, supported features are circuit breaking, automatic retries, global rate limiting, outlier detection and request shadowing.
Health Check	It performs active health check of cluster services.
Observability	It provides server side statistics, access logs, application logs.

***Table 12.3**: Envoy Features*

The Envoy Proxy system provides several ways to secure network traffic, not only the north-south traffic, but also the east-west traffic. To better understand Envoy's approach to security, it is worth recalling the Envoy threat model, which is based on the two main logic aspects:

- Control Plane, which is generally trusted, being the "network orchestrator" for the proxy system.
- Data Plane, the core components are hardened when considering untrusted upstream and downstream connections.

The Envoy threat model implements the CIA Triad model, model expressed around the three fundamental principles of Confidentiality, Integrity and Availability, as

https://t.me/learningnets

discussed in *Chapter 5, Application Container Security*, in the *Penetration Testing* section. Confidentiality refers to the capability of an organization to implement rules or policies to limit and control access to information, ensuring that only a limited group of people with the right set of permissions can access sensitive information. Integrity refers to the assurance that the information has not been tampered, preserving the truthfulness of the data, by implementing a data consistency mechanism over the data management life cycle. Availability ensures that access to information is guaranteed when needed, maintaining hardware, network and software in healthy conditions. The **Transport Layer Security** (**TLS**) is widely supported by the Envoy Proxy system, along with the **TLS termination** and the **TLS origination** features, and a few other HTTP-based protocols like **JWT**, **RBAC** and external OAuth authentication system.

Table 12.4, Envoy TLS feature, summarizes the main features supported by Envoy TLS implementation:

Type	Description
ALPN	TLS extension Application-Layer Protocol Negotiation (ALPN) allows the OSI Layer 7 to choose which protocol should negotiate the secure connection.
BoringSSL	Key Management extension, such as TPM and TLS acceleration, based on the Google version of OpenSSL.
Certificate revocation	Envoy verifies if a certificate is valid against a revocation list. If matches, the certificate is marked as revoked.
Certificate verification	A chain verification process allows Envoy to verify the validity of a certificate using hash pinning and subject name verification.
Client certificate	Client can provide a client certificate.
Ciphers	Each listener can specify which TLS cipher to use.
FIPS 140-2	Federal Information Processing Standard (FIPS) Publication 140-2 sets the cryptographic module security requirements for organizations that need business with US-based federal entities. The BoringSSL crypto library is not FIPS compliant, but compliance can be built to be validated for the FIPS framework using the BoringCrypto core library.
OCSP Stapling	Online Certificate Status Protocol (OCSP) Stapling is a standard to verify X.509 digital certificates revocation status.
Session resumption	Based on RFC 5077, server connection can be resumed via TLS session ticket.
SNI	Server Name Indication (SNI) helps client to declare the hostnames of the machine they are trying to establish a secure connection with.

***Table 12.4**: Envoy TLS features*

https://t.me/learningnets

Despite the funny name, BoringSSL is forked by Google from OpenSSL and is designed by the popular tech giant to meet the specific needs of their security requirements, providing the foundation for the TLS provider used by Envoy. Edge Proxy VPN support can be implemented, thanks to the TLS client authentication filter. When using a VPN service that integrated REST API capabilities, the filter can be invoked to verify that the client certificate hash matches any of the certificates on the principal list to determine whether the connection should be established. Envoy classifies the network traffic into two main categories:

- Containers connecting to Envoy are considered downstream.
- When the connection is from an Envoy Proxy to a container, the connection is considered upstream.

Figure 12.5, Envoy Downstream and Upstream Connection, illustrates the two traffic methodologies adopted by Envoy:

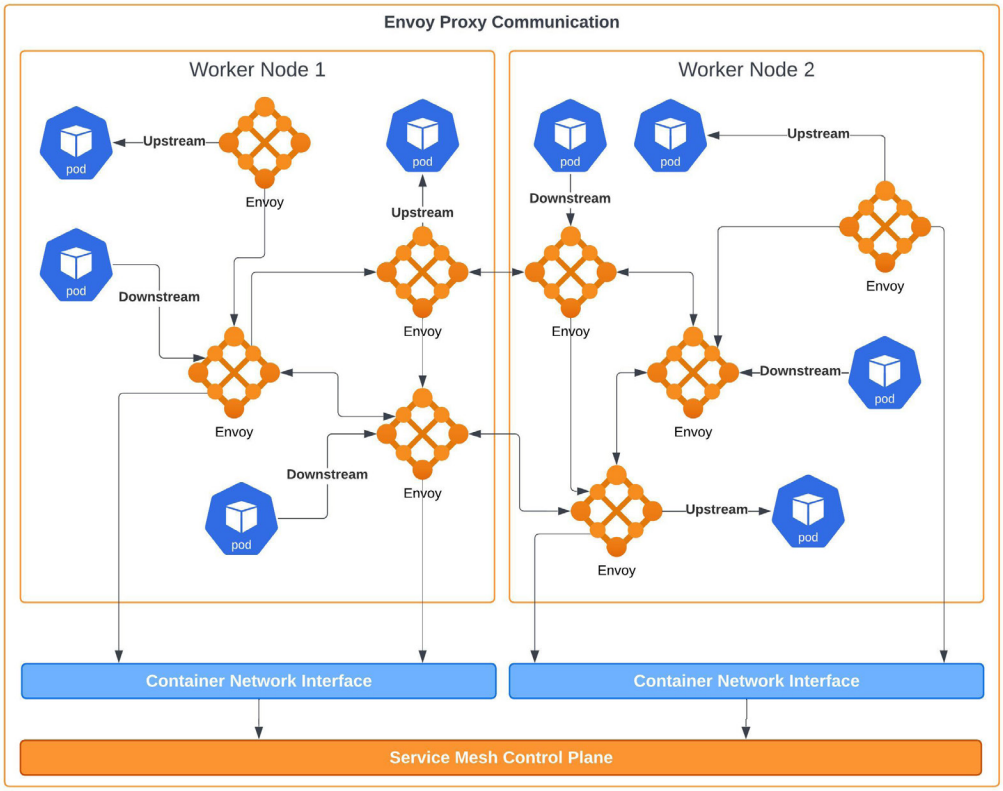

***Figure 12.5**: Envoy Downstream and Upstream*

https://t.me/learningnets

When a client needs to establish a secure connection downstream via TLS, it is possible to configure TLS via the **DownstreamTLSContext** parameter, declaring a server certificate and server key. The following coding snippet clarifies the TLS connection for the listener transport socket:

```
1. static_resources:
2. …
3.   listeners:-  - name: listener_XYZ
4. ...
5.     filter_chains:
6.  -  - filters:
7.    -  - name: envoy.filters.network.http_connection_manager
8. ...
9.       transport_socket:
10.        name: envoy.transport_sockets.tls
11.        typed_config:
12.       “  "@t”pe": type.googleapis.com/envoy.extensions.transport_
   sockets.tls.v3.DownstreamTlsContext
13.          common_tls_context:
14.            tls_certificates:
15.         -  - certificate_chain:
16.               filename: /path-to/servercert.pem
17.             private_key:
18.               filename: /path-to/serverkey.pem
```

Similarly, we can declare the upstream connection with **UpstreamTLSContext**:

```
1. ...
2.     transport_socket:
3.       name: envoy.transport_sockets.tls
4.       typed_config:
5.         "@type": type.googleapis.com/envoy.extensions.transport_
   sockets.tls.v3.UpstreamTlsContext
```

Certificate validation is not implemented by default but can be enforced on the upstream TLS connection through the **validation_context** parameter; therefore, the previous code becomes this:

https://t.me/learningnets

```
...
    transport_socket:
      name: envoy.transport_sockets.tls
      typed_config:
        "@type": type.googleapis.com/envoy.extensions.transport_
sockets.tls.v3.UpstreamTlsContext
        common_tls_context:
          validation_context:
            trusted_ca:
              filename: /path-to/cacert.pem
```

Certificate validation for X.509 can be implemented by adding the **Subject Alternative Names** parameter to the preceding code:

```
...
            trusted_ca:
              filename: /path-to/cacert.pem
            match_typed_subject_alt_names:
            - san_type: DNS
```

The communication between the application and the Envoy proxy supports Online Certificate Status Protocol stapling through the **DownstreamTLSContext** parameter during the handshake. The stapling mechanism enables the verification of revoked certificates, and it is executed with the **`ocsp_staple`** field, where a pre-computed response is provided. Envoy will require a valid response to be provided, which declares the certificate as valid or non-revoked. There is also a custom handshake extension that can override the default SSL behavior by declaring the **`CommonTLSContext`** parameter. Alongside the TLS authentication system discussed so far, Envoy Proxy offers the JSON Web Token, or **JWT** authentication system, which is based on HTTP filter configuration. When an incoming request is received, the JWT Authentication filter verifies the JWT signature's validity based on the HTTP filter; if the validity check fails, the request can either be rejected or passed to an upstream filter that can accept or reject it. Like any other modern authentication system platform, Envoy Proxy supports **Role-Based Access Control**. The RBAC authentication process filters incoming requests against the policy list configured in the filter config; this can be configured as a simple HTTP filter or as a network filter, or eventually, as both. The rules can be defined as listed in *Table 12.5, RBAC Rule Filters*:

https://t.me/learningnets

Type	Description
Policy	A policy is a set of principals and permissions the requests are checked against; inside the permissions logic, the request action is declared, which allows a specific request to be approved or rejected.
Matcher	A matcher supersedes the policy by allowing the request to traverse the matching API system until a match is found. If none is found, the request is denied.
Shadow	The Shadow Policy and Shadow Matcher in relation to the two previous filters can be configured to log request events. It is a sort of dry-run for testing purposes as it does not affect the filters.

***Table 12.5**: RBAC Rule Filters*

As part of the logic behind the RBAC filter, a condition can applied to the policy for the request to be satisfied. This is considered an extra clause of the authentication mechanism in order for the request to be approved. Similar to the RBAC filter, the **External Authorization** system executes external requests to the external identity provider, such as OAuth, and it can be used in conjunction with the HTTP filter and the network filter, or with both.

Secret discovery service

Envoy has been designed to satisfy any use case, from the simplest one with the adoption of a full file static configuration management implementation to the most complex deployment environments where the resource load cannot be inventoried but they need to be discovered. The **Discovery Service**, also referred to as **xDS**, is a set of API-based systems specifically designed for the discovery of a certain type of resources. *Table 12.6, xDS*, details the various type of discovery services supported by Envoy:

Type	Description
EDS	Endpoint Discovery Service is allocated to discover upstream cluster's endpoints.
CDS	Cluster Discovery Service is a layer mechanism to discover upstream clusters through routing. It is recommended to use CDS in combination with EDS.
RDS	Route Discovery Service is an API service aiming to discover route configuration for an HTTP connection.
VHDS	Virtual Host Discovery Service can request virtual host to be discovered independently from the route configuration.
SRDS	Scope Route Discovery Service can decouple route tables.
LDS	Listener Discovery Service looks for listeners.

https://t.me/learningnets

Type	Description
SDS	Secret Discovery Service aims to let Envoy discover cryptographic secrets.
RTDS	RunTime Discovery Service helps the API system retrieve runtime layers.
ECDS	Extension Config Discovery Service allows Envoy to retrieve extension configurations.

***Table 12.6**: xDS*

Among the various type of discovery services, the **Secret Discovery Service** stands out from the security standpoint, unloading the burden to configure static secrets. In a standard Kubernetes deployment, the certificates are stored as secrets, and upon expiration, a secret rotation mechanism must be triggered, which involves re-deployment. In an SDS system, the server will manage the certificates, and upon expiration, will redistribute the new certificates; the Envoy proxy containers do not need to be redeployed because they will simply acknowledge the certificate update. The SDS also acts as a defense mechanism because the listener waits for the certificate to be fetched before establishing any new connection. The SDS server is, essentially, the implementation of the gRPC Secret Discovery Service. The certificates can be declared as secrets in a YAML file format with the **`SdsSecretConfig`** parameter in two specific fields of **`CommonTlsContext`**:

- The **`tls_certificate_sds_secret_configs`**, where the SDS server gets the TLS certificate from
- The **`validation_context_sds_secret_config`**, where the SDS server verifies the TLS certificate validity

The following code provides an illustrated example of SDS Server implementation:

```
1. listeners:
2. ...
3.   filter_chains:
4.   - transport_socket:
5.       name: envoy.transport_sockets.tls
6.       typed_config:
7.         "@type": type.googleapis.com/envoy.extensions.transport_
   sockets.tls.v3.DownstreamTlsContext
8.         common_tls_context:
9.           tls_certificate_sds_secret_configs:
10.          - name: server_certificate
```

https://t.me/learningnets

```
            sds_config:
              resource_api_version: V3
              api_config_source:
                api_type: GRPC
                transport_api_version: V3
                grpc_services:
                  envoy_grpc:
                    cluster_name: sds_server_mtls
          validation_context_sds_secret_config:
          - name: context_validation
            sds_config:
...
   <same as line 12 to 17>
...
                    cluster_name: sds_server_uds
...
```

The SDS Server allows a key rotation mechanism that can be used to rotate the certificates on a need basis. This can be achieved either on the SDS static configuration file or through the gRPC SDS system, with the latter being the recommended approach. The Envoy Proxy currently supports two secret types that can be automatically rotated: the **`CertificateValidationContext`** and the **`TlsCertificate`**. Rotation is applied by adding the **`watched_directory`** parameter, which will monitor the folder in which the certificates are loaded. Therefore, we can amend the code example in the previous section by adding the following:

```
...
            tls_certificates:
            - certificate_chain:
                filename: /path-to/servercert.pem
              private_key:
                filename: /path-to/serverkey.pem
              watched_directory:
                path: /path-to/
```

https://t.me/learningnets

The SDS server also provides a few statistics that can be used for troubleshooting purposes, such as the total count of updated SSL context, total count of empty SSL certification connections that have been reset, both for downstream and upstream listeners, plus the total count of failed key rotations update.

Mutual TLS

Transport Layer Security is the standard de facto today to secure network communications over the internet. The Google Transparency Report website shows a steep curve outlining the adoption of HTTPS encryption traffic on the web over the last 10 years or so, with 94 percent of encrypted traffic registered across the Google platform in February 2023. The report is available at **https://transparencyreport.google.com/safe-browsing/overview**. Still, there is a good percentage of the web traffic that is unencrypted, a percentage that grows when considering communication systems outside Google itself. When we switch the focus from inspecting internet traffic to a local network or **Local Area Network** (**LAN**) traffic, the scenario changes, because the necessity for "trust" is often attributed to external networks only: many consider their home, office, and local network intrinsically secure just because it is at their fingertip. Refer to the following figure:

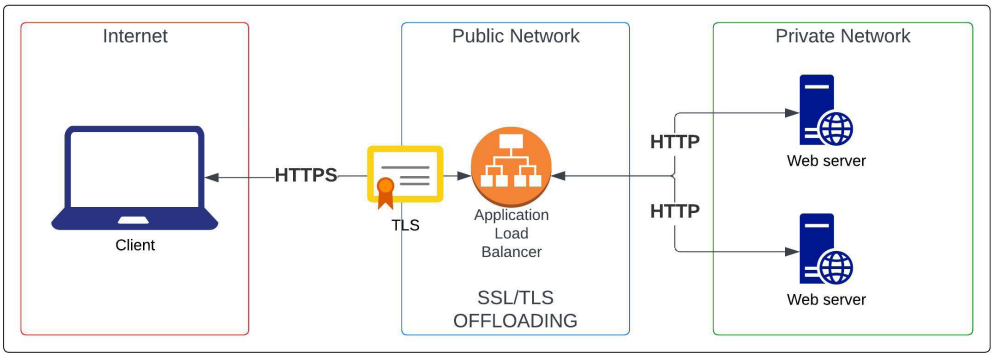

***Figure 12.6**: TLS Termination*

Cloud mechanisms are not indifferent to the same attitude. In many cloud architecture logics, the load balancer that is exposed on the public internet manages the TLS certificate, the network traffic travels on HTTPS until it hits the load balancer, and then it gets decrypted and redirected over HTTP in plain text to the private network where the web server is running, as illustrated in *Figure 12.6, TLS Termination*. The logic behind the TLS termination mechanism is justified by a few reasons:

- Previous versions of SSL/TLS encryption add latency to network connections, slowing down the website response time.
- The SSL/TLS needs additional compute resources creating overhead on the encryption and decryption processes.

https://t.me/learningnets

- There is a tendency to trust internal networks, drawing the line between trusted and not trusted networks exactly where the traffic hits the load balancer.

With the rise of the microservices model and the spread of the containers and the Kubernetes platforms, with a subsequently increasingly complex development and deployment of the application ecosystem, each individual communication between microservices should be secured by implementing authentication, authorization and encryption in each single network transaction to fulfil the requirements of a zero-trust security model. Although this solution elevates the intrinsic security posture of any application, it would add complexity to the programming code needed to encrypt and decrypt the traffic between microservices, and it would not only slow down the software development life cycle but also enhance the need for expertise on any single component of the software stack.

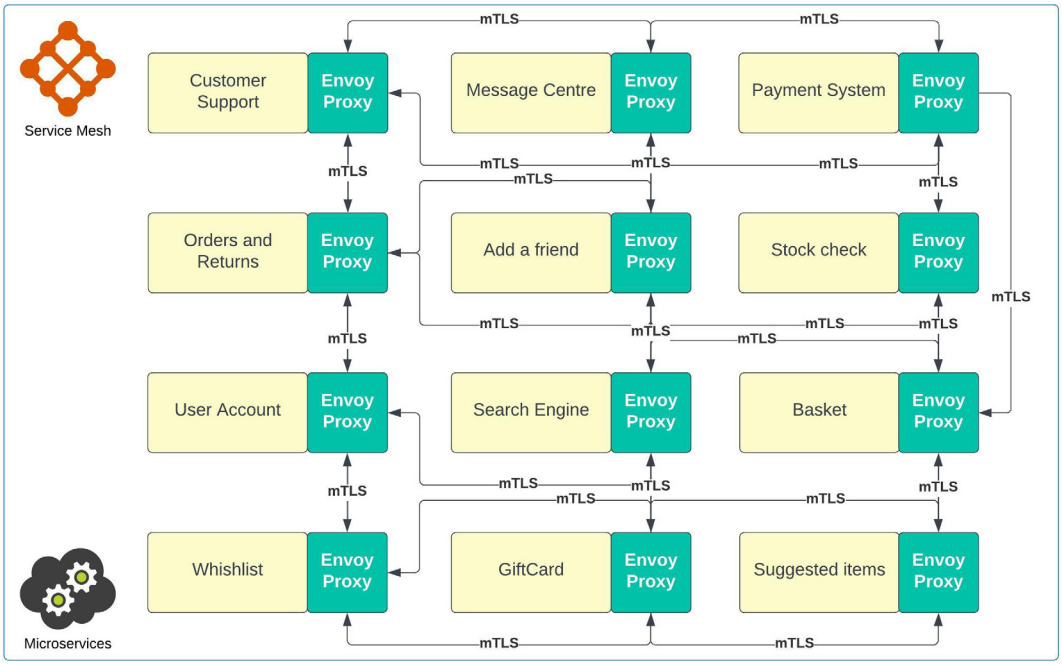

***Figure 12.7**: mTLS*

A service mesh is the ideal software solution to implement security in a microservices model scenario, through the adoption of the **Mutual TLS** (**mTLS**) mechanism, as shown in *Figure 12.7, mTLS*.

https://t.me/learningnets

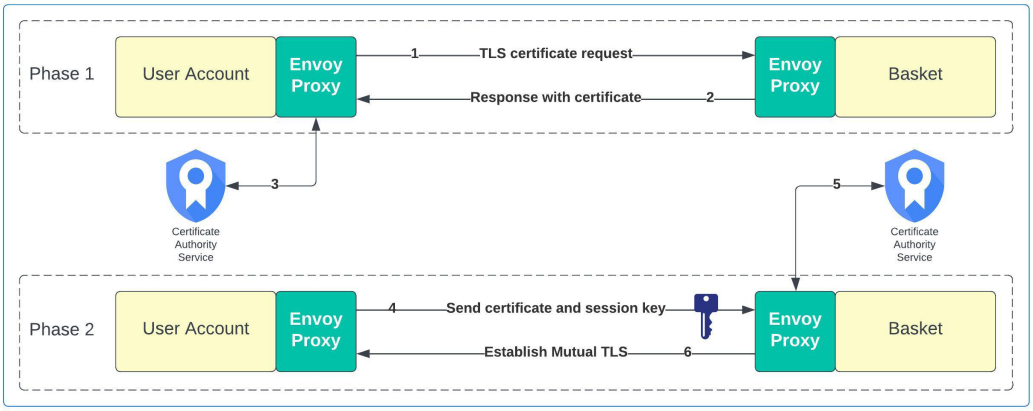

***Figure 12.8**: Establishing Mutual TLS*

The TLS uses a two-way encrypted communication between sidecars, applying a many-to-many TLS communication logic. *Figure 12.8, Establishing Mutual TLS*, shows in detail how the Mutual TLS is created by describing visually what the following steps aim to explain:

1. The User Account microservice's Envoy proxy requests the TLS certificate from the Basket microservice's Envoy proxy.
2. The Basket microservice's Envoy proxy replies with the TLS certificate and requests the User Account microservice's Envoy proxy TLS certificate.
3. The User Account microservice's Envoy proxy verifies the Basket microservice's Envoy proxy certificate by querying the Certificate Authority.
4. Upon successful verification, the User Account microservice's Envoy proxy sends its TLS certificate, plus a session key based on the Basket's Envoy proxy certificate public key, to the Basket's Envoy proxy.
5. The Basket microservice's Envoy proxy verifies the User Account microservice's Envoy proxy TLS certificate by querying the Certificate Authority.
6. Upon successful verification, the Basket's Envoy proxy accepts the session key and establishes Mutual TLS with the User Account microservice's Envoy proxy.

From the code perspective, the minimum requirement to set up a mTLS connection is to leverage the **`require_client_certificate`** parameter and declare a mutual certificate authority, as per the following example:

https://t.me/learningnets

```
      transport_socket:
...
          require_client_certificate: true
          common_tls_context:
            validation_context:
              trusted_ca:
                filename: /path-to/cacert.pem
              match_typed_subject_alt_names:
              - san_type: DNS
                matcher:
                  exact: user-account.mydomain.com
```

The **match_typed_subject_alt_names** parameter is an added restriction on the client authentication side. Upstream connections can be enforced on the client by implementing a certificate chain mechanism, as shown in the following example:

```
        common_tls_context:
          tls_certificates:
          - certificate_chain:
              filename: /path-to/clientcert.pem
```

The **Certificate Authority** (**CA**) is implemented by the service mesh itself, specifically by the service mesh control plane, and must be able to handle the certificate management, the communication with the sidecars, the authentication and authorization policies, and the TLS configuration.

Istio security

The architecture and features described so far are reflected into the majority of the service meshes listed in this chapter, but Istio is likely the most mature and stable among all. Istio has been built around the concepts described in the previous sections, and in particular, it implements security around two distinct logical approaches: control plane security and data plane security. Istio's vision on control plane security is focused on the following:

- A trusted **Certificate Authority** (**CA**) acting as Key Management System, specifically for keys and certificates, also known as **Citadel**
- An authentication policy engine
- An authorization policy engine

https://t.me/learningnets

Data plane security in Istio is built around the following:

- **Policy Enforcement Points** (**PEPs**), to define the sidecar perimeter and secure the communication
- Envoy extension for auditing and monitoring purposes

One of the fundamental concepts Istio relies on is the service identity to verify the identity of the incoming request. In security, identity is a fundamental concept that, in Istio, is the base model for a service-to-service communication to be executed. In the client-server authentication model, Istio provides security on both sides: on the client side, the identity of the server is verified against the **secure naming** information system (the identity is encoded inside the certificate), while on the server side, the server satisfies the "who can access what" logic by implementing **authorization policies** to control the level of permissions. To pursue CA management tasks, three elements need to be considered:

- The **istiod** service running within the control plane.
- The Certificate Authority (Citadel) system running inside the control plane.
- The Istio Agent running alongside the Envoy Proxy inside the data plane.

The provisioning flow of secure artifacts illustrated in *Figure 12.9, Istio Certificate Management,* follows these high-level steps:

- The **istiod** service enables a gRPC service to process certificate signing requests.
- The **Istio Agent** generates both the private key and the certificate signing request, forwarding these to the istiod system via the gRPC protocol.
- The **istiod** system queries the CA to validate the certificate signing requests, and upon successful validation, it creates the X.509 certificate.
- When the POD is created, the Envoy Proxy system queries the Istio Agent for the certificate though the **Secret Discovery Service** (**SDS**).
- The **Istio Agent** provides the certificate retrieved from the istiod system to the Envoy Proxy.

https://t.me/learningnets

- The **Istio Agent** monitors the certificate validity and expiration to execute certificate rotation when necessary.

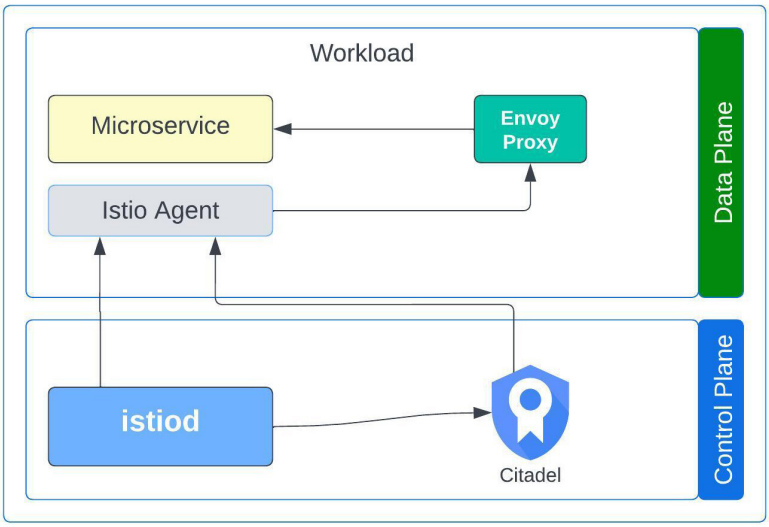

***Figure 12.9**: Istio Certificate Management*

Two authentication systems are provided by Istio, as per *Table 12.7, Istio Authentication*:

Type	Description
Peer	Service-to-service communication via mTLS with the benefits of strong inter-clusters identity, secure network connection between microservices, and Key Management System capability. The Peer authentication works in three modes: • mTLS is disabled; not recommended • Permissive; both mTLS and unencrypted traffic will be allowed • Strict; only mTLS is allowed
Request	User authentication through JWT or any external authentication providers that supports OpenID Connect; the required JWT parameters are the issuer, the public JSON Web Key, and the location of the token.

***Table 12.7**: Istio Authentication*

The **peer authentication** and the **request authentication** as components of the Istio authentication architecture can control the authentication requirements through authentication policies. The concept here is similar to what we discussed in *Chapter 9, Kubernetes Governance*, with reference to systems like Open Policy Agent or Kyverno. When a policy is updated, a new version of the policy is created, and it is enforced into the cluster by the **Policy Enforcement Point** (**PEP**).

https://t.me/learningnets

The authorization mechanism acts at three layers of the Kubernetes cluster: the service mesh architecture, the namespace layer and the POD layer. Authorization policies allow a priority implementation using three selectors: **custom**, **deny** and **allow**. The request will be allowed only if there is no custom action coded or if it has not explicitly denied. The following code provides an authorization policy example, allowing the **staging** namespace and the service account **`cluster.local/ns/default/sa/idle`** to access http-ingress version v1 in **my-namespace** when the JWT token is valid:

```
1. apiVersion: security.istio.io/v1
2. kind: AuthorizationPolicy
3. metadata:
4.   name: http-ingress
5.   namespace: my-namespace
6. spec:
7.   selector:
8.     matchLabels:
9.       app: http-ingress
10.      version: v1
11. action: ALLOW
12. rules:
13. - from:
14.    - source:
15.        principals: ["cluster.local/ns/default/sa/idle"]
16.    - source:
17.        namespaces: ["staging"]
18.    to:
19.    - operation:
20.        methods: ["GET"]
21.    when:
22.    - key: request.auth.claims[iss]
23.      values: ["https://accounts.google.com"]
```

https://t.me/learningnets

The custom condition in the preceding code is implemented by the **when** parameter at line 21. The key elements of a policy are three, as per *Table 12.8, Authorization Policy Elements*:

Type	Description
selector	This is the target field of the policy.
action	The request can be either allow or deny.
rules	The action is triggered by the rule in consideration of the **from**, **to** and **when** fields.

***Table 12.8**: Authorization Policy Elements*

The recommended approach is to use mTLS in the Peer Authentication mode to enhance the security posture of the service mesh. Istio offers a rich set of telemetry for all the components of the service mesh, among which we can observe the following:

- Access logs
- Distributed traces
- Metrics

Access logs are generated to keep track of machine behavioral analytics, which overlaps with the Envoy Proxy access log system. **Distributed traces** are extremely beneficial in troubleshooting scenarios as they monitor individual requests flowing though the service mesh, enabling a very detailed visualization of the information related to the various hops. **Metrics** are provided at the control plane layer, the service layer and the proxy layer. By default, those are exported to Prometheus and can be easily visualized in Grafana.

Zero-Trust networking

The logic behind a zero-trust approach is that user access must always be verified, resource access defined via policies, and devices never trusted, as per the **National Institute of Standard and Technology** (**NIST**) SP 800-207 guidelines. Zero-trust networking is the concept of zero-trust security applied to network communications. A service mesh looks at the software development life cycle from the application standpoint, it is a new network model designed to interconnect microservices. The security features described in this chapter address many security aspects of the Kubernetes attacks surface, as per *Table 12.9, Service Mesh Attack Mitigation*:

https://t.me/learningnets

Type	Description
Impersonation	A combination of mTLS, Secure Naming and Authentication mechanism is difficult to break and also prevents spoofing.
Data Exfiltration	Rate limiting puts control over network traffic.
Sniffing	mTLS secures the communication in both directions.
Unauthorized access	Service-to-service communication is secured via RBAC.

***Table 12.9**: Service Mesh Attack Mitigation*

Unfortunately, there is always the possibility that infrastructures, platforms and networks can be infiltrated by malicious code, exploited by bad actors or exposed due to coding vulnerabilities or misconfigured systems. The Zero-trust networking model uses the motto "never trust, always verify"; and it is exactly what the service mesh is achieving, mainly with the Mutual TLS feature, but also with the enforcement of network policies and the implementation of a Certificate Authority that governs the certificate life cycle. With cloud technologies, the traditional network perimeter, usually defined in the past by the extension of the on-premises network, has shifted to a grey area. The introduction of complex environments like the Kubernetes cluster and the containerized platforms has enhanced the difficulty to define the exact boundaries of the network perimeter. Assuming that every single service, POD or container is legitimately running on the cluster and can be trusted is a bad assumption. In a microservices model, new services can spin up and start serving other services, expanding the attack surface.

Identity validation and identity verification are not intended as standalone one-time processes; in connection with Continuous Integration and Continuous Deployment security best practices, they all are key elements of a Zero-trust security model applied to Kubernetes. A new era has recently started with the rise of the cryptocurrency market where attacker's main objective is to steal compute power resources for the purpose of illegal cryptocurrency mining, also known as **cryptojacking**. Attackers are willing to cover their tracks and stay undetected for as long as possible to consume compute resources to mine cryptocurrencies, in contrast to the much famous ransomware, where the attacked system is encrypted until the ransom is paid. That is exactly what happened to **Tesla** in late 2017, where some Kubernetes clusters of the EV car-maker's AWS infrastructure were running undetected mining processes.

Conclusion

In this chapter, we discussed what a service mesh is, the deployment model is based upon and its architecture. We also described the main concepts behind the container network interface and its contribution to the security of the service mesh and, in

https://t.me/learningnets

general, of the Kubernetes cluster. We then moved on to analyzing what Envoy Proxy is, why it is a fundamental component of the service mesh model and how it contributes to the security of the microservices model, with a particular introspective on two of the Envoy Proxy features: the Secret Discovery Service and the Mutual TLS mechanism.

Among all the service mesh available, we picked Istio, one of the most mature and stable service meshes, and we looked at its peculiarity in consistently applying the security requirements of the service mesh model and the most secure Envoy Proxy capabilities defense and protection techniques. Lastly, we discussed the Zero-Trust Networking model and why both the service mesh and the Istio system naturally complement the security of the Kubernetes cluster in this philosophy.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

https://t.me/learningnets

Index

A

Abstract Container Processes 54
Acceptable Risk Level 134
acceptable risk level concept 132
Access Vector Cache (AVC) 44
Active Directory (AD) 120
admission controller 247-251
After Build scan approach 134
alerts 189
Alpine 109
Alpine Linux 8
Amazon ECS Anomaly Detector 184
Amazon Elastic Container Registry (AWS ECR) 117
Amazon Elastic Kubernetes service 312-321
Amazon Web Services (AWS) 307
Anchore 114, 115, 161, 339
anomaly detection 182-187
Anomaly Detector API 183
Antivirus (AV) engines 11
Apache Mesos 20, 125
Apache web server httpd 41
AppArmor 45
  applying 46-48
Application Container Security 139-150
Application Load Balancer (ALB) 316
Aqua Security Trivy 360
ARP Spoofing 372
ArtifactHUB 343
Artificial Intelligence (AI) 61
Artificial Intelligence Center of Excellence (AICOE) 185
ASTO 152, 153
auditability 125
audit.log file 72
audit logs 72
auditors 264

https://t.me/learningnets

AWS ECS (Elastic Container Service) 184
AWS Elastic Load Balancer 321
AWS EventBridge 184
AWS KMS 206
AWS Lambda 184
AWS Simple Storage Service (S3) 321
AWS SNS 184
Azure CNI 371
Azure Cognitive Service 183
Azure Container Service (ACS) 322
Azure Key Vault 206
Azure Kubernetes Service (AKS) 183, 322-327
Azure Security Benchmark (ASB) 322

B

base image layer 7
Baseline Policy 219
Baseline Scan 146
bastion host access mechanism 331
BIOS 28
BIOS DoS vulnerability DSA-2021-103
  reference link 27
Bitdefender 180
bitlocker-spi-toolkit 32
Black Hat 31
Blue Team 158
BootHole 30
Border Gateway Protocol (BGP) 370
Buildkit 102, 103

C

Canonical Kubernetes 197
capabilities
  applying 42
CasC 86
Center for Internet Security (CIS) 36, 111
Center for Internet Security (CIS) Kubernetes Benchmark 197, 228
Central Processing Unit CPU 62
Certificate Authority (CA) 384
Certificate Signing Request (CSR) 77
CFS parameters 62, 63
Chain of Trust 29
change root 5
Checkov 228
CheckOV
  URL 22
chroot 5, 37
CI/CD Security integration 152
CIS Benchmarks Docker 111
CIS Benchmarks Hardened Images 111
CIS Docker Benchmark 59, 140
CIS Kubernetes Benchmark 112, 203
Citadel 384
Clair 114
Clear Linux 4, 94
client-cert.pem certificate 80
client libraries 173
client-server model 72
Cloud Controller Manager 310
Cloud Integrated Advanced Orchestrator (CIAO) 94
Cloud Native Compute Foundation (CNCF) 339
Cloud Native Computing Foundation (CNCF) 57, 115, 119, 186
Cloud Native Security model 308-312
cloud providers 12
CloudScanner 190
Cloud Security Posture Management (CSPM) 154
CloudSploit 155
Cloud Workload Protection Platform (CWPP) 181

https://t.me/learningnets

CodeZero 197
coexisting tenancy model 253
Common Vulnerabilities and Exposure (CVEs) 94, 130
Common Vulnerability Scoring System (CVSS) 132
Comparing Container Runtimes 55
Completely Fair Scheduler (CFS) 62
Constraint Framework 337
container activity 165
Container Advisor (cAdvisor) 168
containerd 55, 57
Container Engines 54
container image 7, 97, 98
  auditability 125, 126
  control 127-130
  vulnerability management 130-136
container image hardening 98, 99
container images
  evolution 107
  minimal and distroless images 106-112
  scanning 112-117
  verifying 112-117
container image security 97
container layer 7
Container Managers 55
Container Network Interface (CNI) 365, 369
  models 370
  providers 371
container registry 12
  authentication 120-124
  private registry 12
  public registry 12
container risks 15
  application layer 17, 18
  container runtime 15, 16
  network traffic 16, 17
  Rogue containers 18
Container Runtime Interface (CRI) 51
  architecture 216
  list 72
  securing 215-219
Container Scope 294
container security 53-57
  containerd 57
  CRI-O 57-59
  Docker 59
Container Security Verification Standard (CSVS) 15
container-specific OSes 4
container stack monitoring 165
  application monitoring 173-178
  containers monitoring 168-171
  Docker engine monitoring 166-168
  host monitoring 171-173
Continuous Integration (CI) pipeline 112
Continuous Integration Continuous Deployment (CI / CD) pipeline 116
control groups 39-41
  features 40
control plane 196
  hardening 197-212
Control Plane Isolation 253
copy and remove approach 100
Copy-On-Write (COW) 106, 224
CoreDNS system 372
CoreOS rkt 56
Cosign 339
CRI-O 57-59
Cross Site Request Forgery (CSRF) protection 86
Cross Site Scripting (XSS) 147

https://t.me/learningnets

cryptojacking 389
curl command 85
Customer Resource Definition (CRD) 282
CVE-2020-10713 30
CVE-2022-0492 41
CVE-2022-0811 218
CVE-2022-23648 218
CVE framework 132
Cybersecurity and Infrastructure Security Agency (CISA) 193
CycloneDX 360
CycloneDX-gomod 360

D

database container 68
Datadog 148
Data Encryption Key (DEK) system 206
Data Execution Prevention 109
Data Plane 196
Data Plane Isolation 254
Debian Linux distro 111
default service account 316
dependency confusion 160
Dependency-Track 361
deployment script 105
Detectify Deep Scan 146
Device Identifier Composition Engine (DICE) 34, 35
DevSec Hardening Framework 36
DevSecOps 36
Dirty Pipe 225
Discovery Service 378
Discretionary Access Control (DAC) 44
distroless project 107
DNS Spoofing 372
Docker 51, 59
  AppArmor 66, 67
  container isolation 64
  least privilege 59-61
  namespaces 65, 66
  resource limitation 61-64
  SECCOMP 67, 68
Docker Bench for Security 59
docker-ce-rootless-extras package 60
docker-cli 73
Docker Container Processes 53
dockerd 73
docker-default 66
Docker Engine API 73
Docker engine monitoring 166-168
DockerHub 117
docker info command 61
docker inspect 11
dockermap 65
docker run command 61
docker sbom command 161
docker sbom directive 160
Dockershim 193
Docker Swarm 19, 157
Domain Name System (DNS) 372
Dynamic Application Security Testing (DAST) 113, 135, 145, 146

E

East-West traffic 163
ECC P256 31
echo command 40
Elastic Kubernetes Service (EKS) 312, 354
Elasticsearch Logstash and Kibana (ELK) 165
Endpoint Detection and Response (EDR) 152
Envoy proxy 372-378
Envoy TLS feature 374

https://t.me/learningnets

Evil-Maid attack 32
ExecStart 76
exposition format 174
eXtended Detection and Response (XDR) 180
Extensible Markup Language (XML) 151
external method 141
extfile-client.cnf 80

F

Falco 114, 115, 229
FallOfSudo 42
Federal Information Processing Standard (FIPS) 140-2 71
Fedora CoreOS project 56
file configuration
  building 100-104
  multi-stage builds 104-106
FIPS encryption standard mechanism 71
fork/exec model 72
Fourier 186
FROM directive 99
FwHunt application 28

G

gadgets 109
Gatekeeper 269
general-purpose OSes 4
GitHub 113
GitHub Security 113
GitLab 113
GitLab Application Security 113
GitLab Container Registry 119
GitLab Runner agent 114
GKE Autopilot 334
GKE CNI 371
GKE Private Cluster 331
GnuPG 349
GNU Privacy Guard (GNU-PG) 349
Google Cloud Operations Suite 328
Google Cloud Platform (GCP) 328
Google Container Registry 117
Google Distroless 108
Google Kubernetes Engine (GKE) 57, 327-334
GravityZone EDR 180
GravityZone Security for Containers 180
GRUB2 30
Grype 115, 339
gVisor 333

H

Harbor 125
hardening tools 226-229
hard limit 62
hardware security 27-29
  DICE 34, 35
  Security Boot 29, 30
  Trusted Execution Environment (TEE) 33, 34
  Trusted Platform Module (TPM) 31-33
  virtualization-based security 30, 31
Hashicorp Nomad 125
HashiCorp Nomad 20
HashiCorp Vault 157
Helm 343, 344
  architecture 344-346
  chart scanner 356-358
  dependencies 358-361
  IaC trust 351-355
  integrity 349-351
  Tiller 346-348

https://t.me/learningnets

Host OSes
container-specific OSes 4
general-purpose OSes 4
host OS hardening 35, 36
AppArmor 45-48
capabilities 41-43
control groups 39, 40
Linux namespaces 37-39
SECCOMP 48-50
SELinux 43-45
Host OS risks 3
attack surface 4, 5
authentication 6
component vulnerabilities 6
file system integrity 7
system-level virtualization 5
httpd daemon 72
Hvid Security 148
Hyper Text Markup Language (HTML) 151

I

Identity Aware Proxy (IAP) 331
Identity Provider (IDP) 316
image layer 7
image risks 7
embedded malware 10, 11
embedded secrets 9, 10
image misconfiguration 8, 9
image vulnerabilities 8
untrusted images 11, 12
Information Technology (IT) 111, 163
Infrastructure as Code (IaC) 22, 228
scanning 113
InsydeH2O UEFI system 28
Interactive Application Security Testing (IAST) 145, 147
types 148
internal method 141
Internet Control Message Protocol (ICMP) 372
Internet Engineer Task Force (IETF) 349
Intrusion Protection System (IPS) 150
iptables 214
iptables mode 367
IPVS 214
istio security 384

J

Java Server Pages (JSP) 151
JFrog Container Registry 119
JSON WEB KEY 145
JSON WEB Key file 144

K

Keybase.io 351
Key Encryption Key (KEK) 206
KMS 206
Kondukto.io 153
Konnectivity Agent 215
Konnectivity Server 215
kops 197
Kubeadm 197, 230
kube-apiserver 73, 201
Kubeaudit 264
Kube-bench 228
kubebuilder 269
KubeClarity 360
kube-hunter 229
Kubernetes 19, 125, 193
architecture 195, 196
client-server architecture 195
control plane components 196
life cycle, updating 230
namespaces 39

https://t.me/learningnets

Kubernetes Cloud Security
Amazon Elastic Kubernetes service 312-321
Azure Kubernetes Service 322-327
cloud native security model 308-312
Google Kubernetes Engine (GKE) 327-334
Rancher 337, 338
Red Hat OpenShift 334-337
Tanzu 339-341
Kubernetes Features Gates 212
Kubernetes governance 267
admission controller threat model 282-286
limits and limitations 302-304
network policies 286-290
policy engines 268-282
resources management 290-295
security policies 295-302
Kubernetes Network Policies 286
Kubernetes network proxy 213
Kubernetes Orchestration Security
access and verify 261-265
admission controller 247-251
API bypass risks 241-244
audit logging 255-259
authentication 235-241
authorization 235-241
cluster isolation 253-255
POD escaping privilege escalation 260, 261
RBAC, versus ABAC 245-247
secrets, securing 251, 252
Kubernetes Special Interest Group (SIG) 282
Kubernetes STIG Viewer 203
Kubernetes Threats Reference Matrix 194
Kube-Scan 22
Kubespray 197
KVM hypervisor system 57
Kyverno 276, 277

L

LAMP 138
lazy load 144
Let us Encrypt 143, 144
Level 1 Basic Security 140
Level 2 Advanced Security 140
Level 3 High Security 140
Lightweight Directory Access Protocol (LDAP) 120
Linux namespaces 37-39
Local Area Network (LAN) traffic 381
Log4j 160
logs
externalising 187-189
LTS (Long Term Support) 108
LUKS 33
Lynis 36, 37

M

Mandatory Access Control (MAC) 44
Massachusetts Institute of Technology (MIT) License 22
MCSB AWS guidance 323
microservices architecture 138
Microsoft Azure Container Registry 117
Microsoft Cloud Security Benchmark (MCSB) 322
Microsoft Defender system 327
minikube 197, 198
minimal 107
Minimal Ubuntu 108
Mirantis Container Runtime 71
Mitre ATT&CK Container Matrix 158

https://t.me/learningnets

Mitre ATT&CK Kubernetes Matrix 229, 261
Mkit 23
  URL 23
Multi Factor Authentication. (MFA) 119
multi-stage builds 104-106
mutating admission 271
myremapuser 65, 66
my-ubuntu container 40

N

namespace per tenant approach 255
National Institute of Standard and Technology (NIST) 25
National Security Agency (NSA) 43
National Vulnerability Database (NVD) 132
Netfilter 367
network communication
  securing 213-215
network security 68-71
  Mirantis Container Runtime 71
NeuVector 338
Nginx Reverse Proxy 142
NIST Application Container Security Guide 140
node 293
Node Exporter 171
Node-pressure Eviction 303
non-root user 42
northbound traffic 164
North-South traffic 163
Notary 114
  features 117
NSA CISA 228

O

OPA Constraint Framework 269
Open Container Initiative (OCI) 57, 115 161
Open ID Connect (OIDC) 316
Open Policy Agent (OPA) 268
OpenRASP 151
openSUSE MicroOS 4
Open System Interconnection (OSI) model 286
Open Web Application Security Project (OWASP) 15, 360
orchestration 18, 233
orchestrator risks 18-20
  admin access 20
  network segregation 21, 22
  unauthorized access 20
  worker node trust 23
  workload levels 22
Original Device Manufacturers (ODM) 28
Original Equipment Manufacturers (OEM) 28
Out Of Memory Exception (OOME) 62
OverlayFS mechanism 102
overlay network 372
OWASP Container Security Verification Standard 140
OWASP CSVS 139
OWASP Threat Dragon 158
OWASP ZAP (Zed Attack Proxy) 159

P

Palo Alto Networks solution 247
Palo Alto Networks Unit 42 10

https://t.me/learningnets

parallel authorizers method 247
peer authentication 386
penetration testing 157, 158
Personal Health Information (PHI) 140
PID 37
PID1 37
pip (Python Packaging) package 174
Platform9 197
POD 219
  escaping 224, 225
  security 219-223
POD Eviction 303
POD Governance 273
POD level 298
Podman system 72
POD Scope 294
POD Security Admission Controller 295
Pod Security Standards 250
POD Security Standards 295
policy library 271
Portainer.io 125
Position Independent Executables (PIE) 109
Power On phase 28
Pretty Good Privacy (PGP) 349
private registry 117-120
Privileged Policy 219
privileges
  managing 155-157
Proactive Data Loss Prevention (DLP) 11
production database container 68
production registry 135
Prometheus 165
Prophet 186
public registry 117-120

Q

Quality of Service (QoS) 290

R

RAM parameters 62
Rancher 20, 125, 337, 338
RancherOS 4
Rancher Prime 337
Rapid7 InsightAppSec 146
RASP 149, 150
RBAC API system 245
RBAC Police 247
Realtime Scheduler 63
  parameters 63, 64
Red Hat OpenShift 20, 125, 334-337
Red Hat OpenShift Container Platform 334
Red Hat OpenShift Kubernetes Engine 334
RedHat Quay.io 119
Red Team 157
registry 12
Registry Access Management (RAM) 125
registry risks
  authentication and authorization 15
  non-secure connections 13, 14
  stale images 14
Registry Scan approach 134
Rego 269
remediation 154, 155
request authentication 386
Restricted Policy 220
Return Oriented Programming (ROP) 109, 110
Reverse Proxy
  features 141

https://t.me/learningnets

Rogue containers 18
Role-Based Access Control (RBAC) 18, 124, 125
RSA private key 32
Runtime Application Self-Protection (RASP) 145
Runtime Scan strategy 135

S
scaling 233
Scope 191
  characteristics 191
Scope Traffic Control 191
Scope Volume Count 191
SECCOMP 48-50
Secret Discovery Service 379
Secure and Validated Containers 71
secure connection 73-76
  CI/CD, securing 86-94
  client certificate 79-81
  dockerd TLS, enabling 81-85
  server certificate 77-79
Secure Connection 117
Secure File Transfer Protocol (SFTP) 44
Secure Software Development Life Cycle (SSDLC) 10, 152
Secure Validated Encryption 71
Security Contexts 220
Security Event Management (SEM) 187
Security Group (SG) service 354
Security Information and Event Management (SIEM) 181
Security Information Management (SIM) tool 178
security information monitoring 163
security-opt parameter 66
Security Orchestration Automation Response (SOAR) 189
Security Posture Dashboard 328, 334
security profiles 220
Security Technical Implementation Guide (STIG) 36, 71
SELinux (Security Enhanced Linux) 43-45
Sensor Agents 190
service discovery 367
Service Mesh Control Plane 365
Service Mesh Data Plane, 366
Service Mesh Security
  architecture 367-369
  Container Network Interface (CNI) 369-372
  Envoy security 372-378
  istio security 384-388
  mutual TLS 381-384
  overview 365-367
  secret discovery service 378-381
  Zero-Trust networking 388, 389
service-oriented architecture (SOA) 138
SHA-256 31
Shielded GKE Nodes 332
Shielded VM (Virtual Machine) 332
shift left methodology 153, 154
Shift Right methodology 154
shift-to-the-left approach 112
sidecars 366, 368
SIEM tool 188
  capabilities 188
Single Sign On (SSO) 120
Social Security Numbers (SSNs) 154
soft limit 62
Software Bill of Materials (SBOM) 360
Software Composition Analysis (SCA) 10, 356, 360
Software Development Life Cycle (SDLC) 10, 22, 139

https://t.me/learningnets

SolarWinds 115
SonarQube 113
Sonatype Nexus Repository 119
southbound traffic 164
Sparrow 115
SSH tunnelling mechanism 214
stack buffer overflow 109
stack smashing attack 109
Staging Registry strategy 135
staging web container 68
Static Application Security Testing (SAST) 10, 113, 356, 360
Static PODs 243
sudoers 42
Super User DO (sudo) command 42
Syft 115, 161
Sysdig 115
systemd-nspawn container runtime feature 56

T

T2 Chip 32
Tallow 94
Tanzu 339-341
Tanzu Application Platform 339
Tanzu Mission Control 339
tcp socket 73
Terraform 228
Terraform helm chart 354
Tesla 389
Test Docker Pipeline job 93
Text-based format 174
Thanos 186
The Container Technology Stack 25
The Update Framework (TUF) concept 117
third-party components 160
threat intelligence 150-152
ThreatMapper 190
Tiller 345, 346
Tinyproxy service 331
Titan 328
Topology Manager 294
topology visualization 190, 191
Transport Layer Security (TLS) 197, 374
  connection 117
TrendMicro 118
Trivy 116, 345
Trusted Execution Environment (TEE) 33
  requirements 33
Trusted Platform Module (TPM) 28-33

U

UDICA 45
UEFI solution 28
uidmap package 59
UNC2452 115
Unified Extensible Firmware Interface Secure Boot 30
UnionFS 7
Unit 42 researcher 41
Universal Base Image 107
unix socket 73
untrusted hypervisors 32

V

v1alpha2 51
validating admission 271
Vault 157
Vega 148
Veracode Dynamic Analysis 146
virtual control plane per tenant approach 255
virtualenv 5
Virtual eXtensible LAN (VxLAN) 370

https://t.me/learningnets

virtualization-based security (VBS) 28, 30
Virtual Local Area Network (VLAN) 68
Virtual Private Cloud (VPC) 354
VMWare Tanzu Build Service 339
VMWare vSphere implementation 337
vTPM (virtual TPM) 32
vulnerability management 130-136

W

W3AF 148
Watchtower 129
web container 68
Windows Nano Server 8
worker node 196, 197
  monitoring 212, 213
workload density 179
workload observability 178-182
workloads 195
writable layer 99

X

xDS 378
XMRig 10
XRAY 114

Y

Yet Another Markup Language (YAML) 345

Z

ZAP 159
ZAP (Zed Attack Proxy) 146
zero-day attacks 45
zero-trust networking 388, 389

https://t.me/learningnets