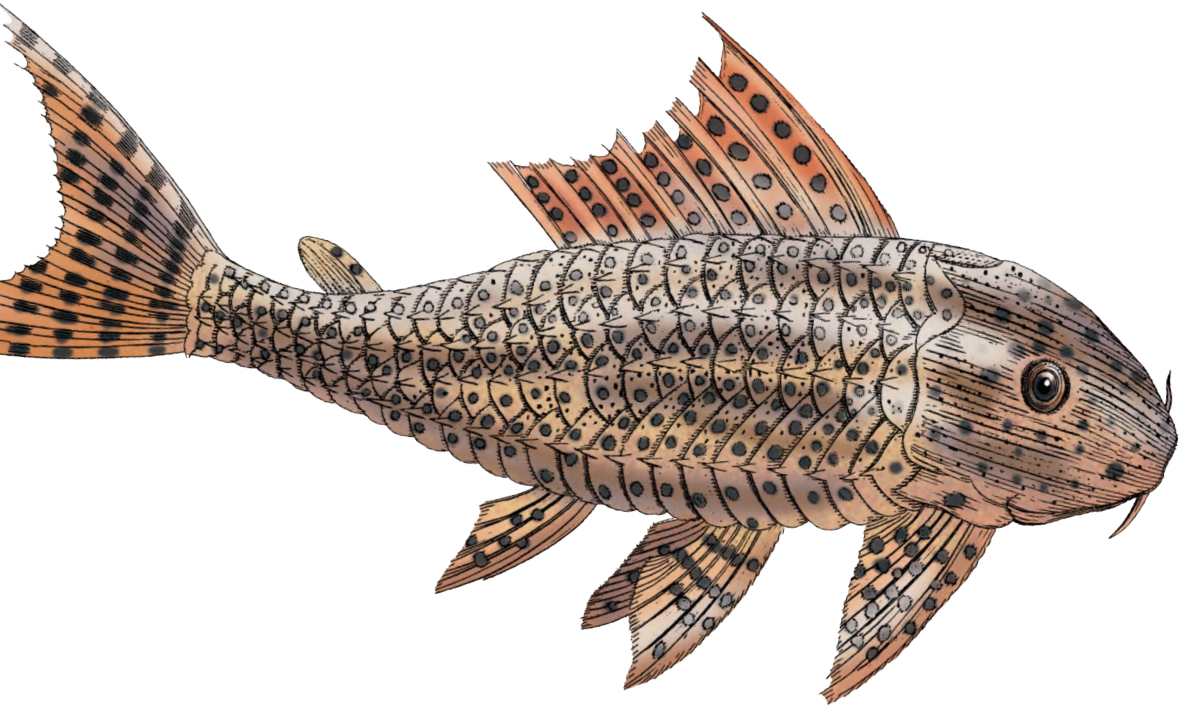
O'REILLY®

2nd Edition

Container Security

Fundamental Technology Concepts That Protect Cloud Native Applications



Liz Rice

"The definitive guide to Linux kernel, container, and VM isolation in Liz's inimitable style. If you enjoy her 'from scratch' talks and demos, this book will take you deeper into the mystical world of container security with illuminating tips, tools, and techniques to keep your applications and data safe."

Andrew Martin, CEO, ControlPlane

"Liz's concrete and hands-on approach to container security makes this book an invaluable resource for technology professionals. With expanded coverage on Linux system and container fundamentals, this new edition is a must-have for secure container operation."

Phil Estes, principal engineer, containers and open source strategy, AWS

Container Security

As containerized and cloud native applications become foundational to modern software infrastructure, the need for a deep, conceptual understanding of their security implications has never been more urgent. *Container Security*, second edition, offers a rigorous yet practical examination of the technologies that underpin container platforms—equipping developers, operations professionals, and security practitioners with the mental models needed to evaluate risk and enhance resilience.

Written by Liz Rice, a recognized authority in cloud native security, this updated edition builds on the foundational principles from the first edition while incorporating today's evolving threat landscape, modern tooling, and advancements in platforms like Kubernetes and Linux. Readers will gain a firm grasp of the architectural components behind containers and the Linux primitives that support them, fostering a systems-level understanding of both threats and mitigation strategies.

- **Examine the technical underpinnings of containers through a security-focused lens**
- **Evaluate evolving risks and defenses across container runtimes and orchestration platforms**
- **Analyze the implications of modern tooling including eBPF and AI-driven approaches**
- **Apply core principles to assess and secure real-world deployments in dynamic environments**

Liz Rice is chief open source officer at Isovalent, the creator of Cilium, and now part of Cisco. She is a former CNCF governing board member, CNCF technical oversight committee chair, and KubeCon cochair. Rice is also the author of O'Reilly's *Learning eBPF* and the first edition of *Container Security*.

CONTAINERS

US $55.99 CAN $69.99
ISBN: 979-8-341-62770-3

O'REILLY®

SECOND EDITION

Container Security

Fundamental Technology Concepts That Protect Cloud Native Applications

Liz Rice

O'REILLY®

Container Security
by Liz Rice

Printed in the United States of America.

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Megan Laddusaw
Development Editor: Rita Fernando
Production Editor: Elizabeth Faerm
Copyeditor: Adam Lawrence
Proofreader: Kim Wimpsett
Indexer: WordCo Indexing Services, Inc.
Cover Designer: Susan Brown
Cover Illustrator: Karen Montgomery
Interior Designer: David Futato
Interior Illustrator: Kate Dullea

April 2020: First Edition
October 2025: Second Edition

Revision History for the Second Edition
2025-10-07: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=0642572174828* for release details.

979-8-341-62770-3

[LSI]

Table of Contents

Preface

Many organizations are running applications in cloud native environments, using containers and orchestration to facilitate scalability and resilience. If you're a member of the Operations, Security, DevOps, or even DevSecOps teams setting up these environments for your company, how do you know whether your deployments are secure? If you're a security professional with experience in traditional server-based or virtual machine–based systems, how can you adapt your existing knowledge for container-based deployments? And as a developer in the cloud native world, what do you need to think about to improve the security of your containerized applications? This book delves into some of the key underlying technologies that containers and cloud native computing rely on, to leave you better equipped to assess the security risks and potential solutions applicable to your environment and to help you avoid falling into bad practices that will leave your technology deployments exposed.

In this book you will learn about many of the building block technologies and mechanisms that are commonly used in container-based systems and how they are constructed in the Linux operating system. Together we will dive deep into the underpinnings of how containers work and how they communicate so that you are well versed not just in the "what" of container security but also, and more importantly, in the "why." My goal in writing this book is to help you better understand what's happening when you deploy containers. I want to encourage you to build mental models that allow you to make your own assessment of potential security risks that could affect your deployments.

This book primarily considers the kind of "application containers" that many businesses are using these days to run their business applications in systems such as Kubernetes and Docker. This is in contrast to "system containers" such as LXC and LXD from the Linux Containers Project. In an application container, you are encouraged to run immutable containers with as little code as is necessary to run the application, whereas in a system container environment, the idea is to run an entire Linux distribution and treat it more like a virtual machine. It's considered perfectly normal to use SSH (Secure Shell) to access a system container, but security experts will look

at you askance if you want to SSH into an application container in production (for reasons covered later in this book). However, the basic mechanisms used to create application and system containers alike are control groups, namespaces, and changing the root directory, so this book will give you a solid foundation from which you may wish to explore the differences in approach taken by the different container projects.

Be warned that there are a few "container" implementations, like the new Apple Containerization project, that isolate workloads using different technologies. You'll understand the differences by the end of this book!

Who This Book Is For

Whether you consider yourself a developer, a security professional, an operator, or a manager, this book will suit you best if you like to get into the nitty-gritty of how things work and if you enjoy time spent in a Linux terminal.

If you are looking for an instruction manual that gives a step-by-step guide to securing containers, this may not be the book for you. I don't believe there is a one-size-fits-all approach that would work for every application in every environment and every organization. Instead, I want to help you understand what is happening when you run applications in containers and how different security mechanisms work so that you can judge the risks for yourself.

As you'll find out later in this book, containers are made with a combination of features from the Linux kernel. Securing containers involves using a lot of the same mechanisms as you would use on a Linux host. (I use the term "host" to cover both virtual machines and bare-metal servers.) I lay out how these mechanisms work and then show how they apply in containers. If you are an experienced system administrator, you'll be able to skip over some sections to get to the container-specific information.

You'll have noticed that I mentioned Linux a few times already—and yes, there are other operating systems! I'll mention other OSs once or twice, but this book mostly focuses on Linux containers running on Linux hosts.

I assume that you have some basic familiarity with containers and you have probably at least toyed with Docker or Kubernetes. You will understand terms like "pulling a container image from a registry" or "running a container" even if you don't know exactly what is happening under the covers when you take these actions. I don't expect you to know the details of how containers work—at least, not until you have read the book.

What This Book Covers

We'll start in Chapter 1 by considering threat models and attack vectors that affect container deployments, and the aspects that differentiate container security from traditional deployment security. The remainder of the book is concerned with helping you build a thorough understanding of containers and these container-specific threats, and with how you can defend against them.

Before you can really think about how to secure containers, you'll need to know how they work. Chapter 2 sets the scene by describing some core Linux mechanisms such as system calls and capabilities that will come into play when we use containers. Then in Chapters 3 and 4, we'll delve into the Linux constructs that containers are made from. This will give you an understanding of what containers really are and of the extent to which they are isolated from each other. We'll compare this with virtual machine isolation in Chapter 5.

In Chapter 6 you'll learn about the contents of container images and consider how to build them with security in mind. Chapter 7 goes on to discuss supply chain security best practices, for ensuring that container images and their contents aren't tampered with. Chapter 8 addresses the need to identify container images with known software vulnerabilities. This relies on treating containers as immutable, and Chapter 9 considers the security benefits of taking immutability a step further, and applying a GitOps approach to deploying containers and their configuration.

In Chapter 10 we will look at some optional Linux security measures that can be applied to harden containers beyond the basic implementation we saw in Chapter 4, and some variant approaches to isolating containers from each other. We will look into ways that container isolation can be compromised through dangerous but commonplace misconfigurations in Chapter 11.

Then we will turn to the communications between containers. Chapter 12 looks at how containers communicate and explores ways to leverage the connections between them to improve security. Chapter 13 explains the basics of keys and certificates, which containerized components can use to identify each other and set up secure network connections between themselves. This is no different for containers than it is for any other component, but this topic is included since keys and certificates are often a source of confusion in distributed systems. In Chapter 14 we will see how certificates and other credentials can be safely (or not so safely) passed to containers at runtime.

In Chapter 15 we will consider ways in which security tooling can prevent attacks at runtime, taking advantage of the features of containers.

Finally, Chapter 16 reviews the top 10 security risks published by the Open Web Application Security Project and considers container-specific approaches for addressing them. Spoiler alert: some of the top security risks are addressed in exactly the same way whether your application is containerized or not.

What Has Changed in the Second Edition?

In the five years since the first edition of this book was published in 2020, there has been considerable change in the container ecosystem. Later that year, the SolarWinds cyberattack focused attention on the software supply chain, leading to the development of a whole field of supply chain security. The term "GitOps" was first uttered in 2018 and has been widely adopted in the last few years, perhaps because organizations took advantage of "runners" provided by GitHub and other repository platforms in a modern approach to CI/CD. eBPF emerged as the preeminent technology for runtime security tools and for container networking with policy enforcement. Even the fundamentals that underpin containers have evolved: they still use Linux namespaces, but the user namespace is now used quite commonly, rootless containers are a reality, and the ecosystem has moved on to control groups version 2.

The communities around cloud native and containers have helped to spread education about security best practices, so some of the most worrying techniques (like installing packages into a running container) are thankfully much less commonplace than they used to be. They are still included in this book because it's important that we don't, as an industry, regress to using these antipatterns!

And perhaps the biggest consolidation since I wrote the first edition: Kubernetes has become firmly established as the dominant orchestrator.

A Note About Kubernetes

These days the majority of folks using containers are doing so under the Kubernetes orchestrator. An orchestrator automates the process of running different workloads in a cluster of machines, and there are places in this book where I will assume you have a basic grasp of this concept. In general, I have tried to stay focused on concepts that act at the level of the underlying containers—the "data plane" in a Kubernetes deployment.

Because Kubernetes workloads run in containers, this book is relevant to Kubernetes security, but it is not a comprehensive treatment of everything related to securing Kubernetes or cloud native deployments. You'll find a good, up-to-date list of security considerations in the Kubernetes documentation, including the configuration and use of the control plane components that are outside the scope of this book.

However widespread Kubernetes may be, it's not the only option. There is widespread use of managed container platforms like AWS Elastic Container Service, AWS Fargate, Azure Container Instances, and Google Cloud Run, and Docker and Podman are often used, particularly for local development, CI pipelines, and small-scale deployments.

Examples

There are lots of examples in this book, and I encourage you to try them out for yourself.

In the examples, I assume you are comfortable with basic Linux command-line tools like `ps` and `grep` and with the basic day-to-day activities of running container applications through the use of tools like `kubectl` or `docker`. This book will use the former set of tools to explain a lot more about what's happening when you use the latter!

To follow along with the examples in this book, you will need access to a Linux machine or virtual machine. I created the examples using an Ubuntu 24.04 virtual machine. You should be able to achieve similar results on different Linux distributions and using virtual machines running on your local machine or on your favorite cloud provider.

How to Run Containers

For many people, their main (perhaps only) experience of running containers directly is with Docker. Docker democratized the use of containers by providing a set of tools that developers generally found easy to use. From a terminal, you manipulate containers and container images using the `docker` command.

This `docker` tool is really a thin layer making API calls to Docker's main component: a daemon that does all the hard work. Within the daemon is a component called `containerd` that is invoked whenever you want to run a container. The `containerd` project was donated by Docker to the Cloud Native Computing Foundation (CNCF) in 2017.

The `containerd` component makes sure the container image you want to run is in place, and it then calls a `runc` component to do the business of actually instantiating a container. If you want to, you can run a container yourself by invoking `containerd` or even `runc` directly.

Kubernetes uses an interface called the Container Runtime Interface (CRI) beneath which users can opt for a container runtime of their choice. The most commonly used options today are the aforementioned `containerd` and `CRI-O` (which originated from Red Hat before being donated to the CNCF).

The `docker` CLI is just one option for managing containers and images. There are several others you can use to run the kind of application containers covered in this book. Red Hat's `podman` tool, originally conceived to remove reliance on a daemon component, is one such option.

The examples in this book use a variety of different container tools to illustrate that there are multiple container implementations that share many common features.

At the time I'm writing this second edition, Apple has just launched its containerization project. It uses the same container image formats as Docker (which you'll learn about in Chapter 6), and the command-line interface has a lot of similarities, but as you'll see when you get to Chapter 10, it uses a different mechanism to isolate containers from each other. For that reason, several examples from this book won't behave the same if you run them using Apple containerization.

Feedback

The website *containersecurity.tech* accompanies this book. You are invited to raise issues there with feedback and any corrections that you'd like to see in future editions.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for commands and output listings, as well as within paragraphs to refer to technical elements such as library, file, image, or function names, environment variables, statements, and keywords.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a general note.

Using Code Examples

Supplemental code examples are available at *https://containersecurity.tech*. There is a sandbox environment for running these examples, along with quiz questions to test your learning, on the O'Reilly online platform.

If you have a technical question or a problem using the code examples, please send email to *support@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Container Security*, 2nd edition, by Liz Rice (O'Reilly). Copyright 2026 Vertical Shift Ltd., 979-8-341-62770-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
141 Stony Circle, Suite 195
Santa Rosa, CA 95401
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
https://oreilly.com/about/contact.html

We have a web page for this book, where we list errata and any additional information. You can access this page at *https://oreil.ly/container-security-2e*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

I'm grateful to many people who have helped and supported me through the process of writing this book:

- My editors at O'Reilly, Rita Fernando (second edition) and Virginia Wilson (first edition), for keeping everything on track and making sure the book is up to scratch
- The technical reviewers who provided thoughtful comments and actionable feedback on the first and second additions: Akhil Behl, Alex Pollitt, Andrew Martin, Erik St. Martin, Jed Salazar, Phil Estes, Rani Osnat, and Robert P. J. Day
- My teammates at Isovalent, my former colleagues at Aqua Security, and countless people from the cloud native community, who taught me so much about container security over the years
- Phil Pearl—my husband, my best critic and coach, and my best friend

CHAPTER 1
Container Security Threats

In the past decade or so, the use of containers has exploded. The concepts around containers existed for several years before Docker, but most observers agree that it was Docker's easy-to-use command-line tools that started to popularize containers among the developer community from its launch in 2013.

Containers bring many advantages: as described in Docker's original tagline, they allow you to "build once, run anywhere." They do this by bundling together an application and all its dependencies and isolating the application from the rest of the machine it's running on. The containerized application has everything it needs, and it is easy to package up as a container image that will run the same on my laptop and yours, in the cloud, or on a server in a data center.

A knock-on effect of this isolation is that you can run multiple different containers side by side without them interfering with each other. Before containers, you could easily end up with a dependency nightmare where two applications required different versions of the same packages. The easiest solution to this problem was simply to run the applications on separate machines. With containers, the dependencies are isolated from each other so it becomes straightforward to run multiple apps on the same server. People quickly realized that they could take advantage of containerization to run multiple applications on the same host (whether it's a virtual machine or a bare-metal server) without having to worry about dependencies.

The next logical step was to spread containerized applications across a cluster of servers. Orchestrators such as Kubernetes automate this process so that you no longer have to manually install an app on a particular machine; you tell the orchestrator what containers you want to run, and it finds a suitable location for each one.

From a security perspective, many things are the same in a containerized environment as they are in a traditional deployment. There are attackers out in the world who want to steal data, modify the way a system behaves, or use other people's compute resources to mine their own cryptocurrencies. This doesn't change when you move to containers. However, containers do change a lot about the way that applications run, and there are a different set of risks as a result (see Figure 1-1).

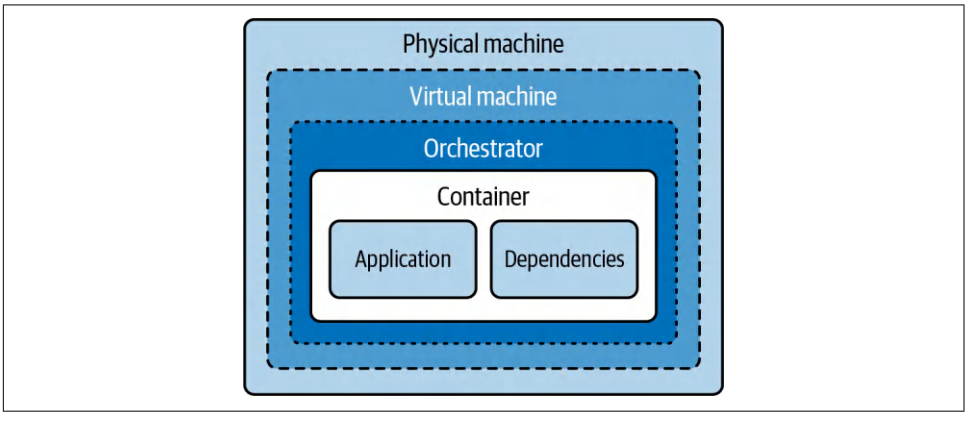


Figure 1-1. A containerized application

As shown in Figure 1-1, an application and its dependencies run inside a container, which might be under the control of an orchestrator like Kubernetes or AWS Elastic Container Service. The container might run directly on a physical host computer or within a virtual machine host, which in turn runs on a physical machine. Chapter 3 and Chapter 4 of this book will help you understand how a container isolates the application from the underlying host and from other containers, and Chapter 5 explains the very different isolation mechanisms used for virtual machines. Understanding the differences is crucial for thinking effectively about the possible risks and threats to your containerized deployment and the mitigations you can use to keep your containers safe.

Risks, Threats, and Mitigations

A *risk* is a potential problem and the effects of that problem if it were to occur.

A *threat* is a path to that risk occurring.

A *mitigation* is a countermeasure against a threat—something you can do to prevent the threat or at least reduce the likelihood of its success.

For example, there is a risk that someone could steal your car keys from your house and thus drive off in your car. The threats would be the different ways they might steal the keys: breaking a window to reach in and pick them up; putting a fishing rod

through your letter box; knocking on your door and distracting you while an accomplice slips in quickly to grab the keys. A mitigation for all these threats might be to keep your car keys out of sight.

Risks vary greatly from one organization to another. For a bank holding money on behalf of customers, the biggest risk is almost certainly of money being stolen. An ecommerce organization will worry about the risks of fraudulent transactions. An individual running a personal blog site might fear someone breaking in to impersonate them and post inappropriate comments. Because privacy regulations differ between nations, the risk of leaking customers' personal data varies with geography—in some countries, the risk is "only" reputational, while in Europe, the General Data Protection Regulation (GDPR) allows for fines of up to 4% of a company's total revenue.

Because the risks vary greatly, the relative importance of different threats will also vary, as will the appropriate set of mitigations. A risk management framework is a process for thinking about risks in a systematic way, enumerating the possible threats, prioritizing their importance, and defining an approach to mitigation.

Threat modeling is a process of identifying and enumerating the potential threats to a system. By systematically looking at the system's components and the possible modes of attack, a threat model can help you identify where your system is most vulnerable to attack.

There is no single comprehensive threat model, as it depends on your risks, your particular environment, your organization, and the applications you're running, but it is possible to list some potential threats that are common to most, if not all, container deployments.

Container Threat Model

One way to start thinking about the threat model is to consider the actors involved. These might include:

- *External attackers* attempting to access a deployment from outside
- *Internal attackers* who have managed to access some part of the deployment
- *Malicious internal actors* such as developers and administrators who have some level of privilege to access the deployment
- *Inadvertent internal actors* who may accidentally cause problems
- *Application processes* that, while not sentient beings intending to compromise your system, might have programmatic access to the system

Each of these threat actors has a certain set of permissions that you need to consider:

- What access do they have through credentials? For example, do they have access to user accounts on the host machines your deployment is running on?
- What permissions do they have on the system? In Kubernetes, this could refer to the role-based access control settings for each user, as well as anonymous users. Permissions could be additionally controlled by policy engines or runtime security tools. In public cloud environments, permissions also depend on identity and access management (IAM) policies.
- What network access do they have? For example, which parts of the system are included within a virtual private cloud (VPC)? Are network security policies in place to limit access?

There are several possible ways to attack a containerized deployment, and one way to map them is to think of the potential attack vectors at each stage of a container's life cycle. These are summarized in Figure 1-2.

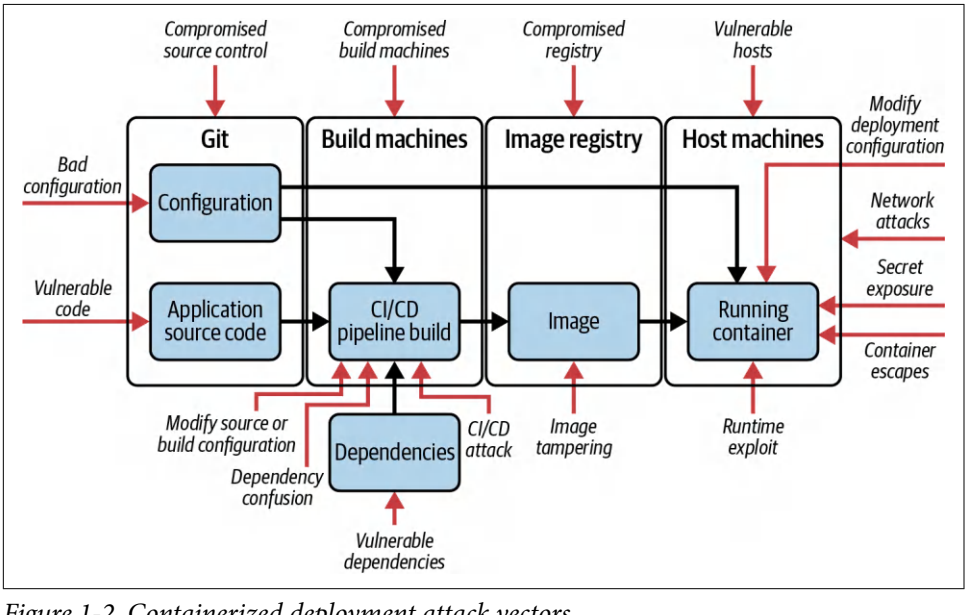


Figure 1-2. Containerized deployment attack vectors

Vulnerable application code and dependencies

The life cycle of a container starts with the application source code that a developer writes. This code, and the third-party dependencies that it relies on, can include flaws known as vulnerabilities, and there are thousands of published vulnerabilities that an attacker can exploit if they are present in an application. The best way to avoid running containers with known vulnerabilities is to scan

images, as you will see in Chapter 8. This isn't a one-off activity, because new vulnerabilities are discovered in existing code all the time. The scanning process also needs to identify when containers are running with out-of-date packages that need to be updated for security patches. Some scanners can also identify malware that has been built into an image.

Badly configured container images
: Once the code has been written, it gets built into a container image. When you are configuring how a container image is going to be built, there are plenty of opportunities to introduce weaknesses that can later be used to attack the running container. These include configuring the container to run as the root user, giving it more privilege on the host than it really needs. You'll read more about this in Chapter 6.

Supply chain and build machine attacks
: Source code and build configuration are inputs to a build process that creates a container image. The container image gets stored in a registry and later retrieved or "pulled" from the registry at the point where it's going to be run. If an attacker can modify the source, influence the way a container image is built, replace the container image in the registry, or cause the wrong image to be pulled, they could insert malicious code that will subsequently get run in the production environment.

  How do you know that the right source code and dependencies are being used to build the image? How do you know that the image you pull is exactly the same as what you pushed earlier? Could it have been tampered with? An actor who can affect source code or configuration files, replace an image, or modify an image between build and deployment has the ability to run arbitrary code on your deployment. You'll read about these attacks on the "supply chain" in Chapter 7.

Badly configured containers
: As we'll discuss in Chapter 11, it's possible to run containers with settings that give them unnecessary, and perhaps unplanned, privileges. If you download YAML configuration files from the internet, please don't run them without carefully checking that they do not include insecure settings!

Vulnerable hosts
: Containers run on host machines, and you need to ensure that those hosts are not running vulnerable code (for example, old versions of orchestration components with known vulnerabilities). It's a good idea to minimize the amount of software installed on each host to reduce the attack surface, and hosts also need to be configured correctly according to security best practices. This is discussed in Chapter 4.

It's also a good idea to limit the users who have access to the host machines. One way to achieve this is through GitOps, discussed in Chapter 9, so that all code and configuration is stored under code control and updated using an automated process rather than allowing manual configuration.

Insecure networking
: Containers generally need to communicate with other containers or with the outside world. Chapter 12 discusses how networking works in containers, and how microsegmentation and network policy can restrict network access. Chapter 13 goes on to consider secure, encrypted connections between components.

Exposed secrets
: Application code often needs credentials, tokens, or passwords in order to communicate with other components in a system. In a containerized deployment, you need to be able to pass these secret values into the containerized code. As you'll see in Chapter 14, there are different approaches to this, with varying levels of security.

Runtime exploits
: The widely used container runtimes including `containerd` and `CRI-O` are by now pretty battle-hardened, but it's still within the realm of possibility that there are bugs yet to be found that would let malicious code running inside a container escape out onto the host. Similarly, kernel vulnerabilities are found from time to time that can allow for container escape or privilege escalation. You'll read about the isolation that is supposed to keep application code constrained within a container in Chapter 4. For some applications, the consequences of an escape could be damaging enough that it's worth considering stronger isolation mechanisms, such as those covered in Chapter 10. There are also security tools, considered in Chapter 15, that can detect and even prevent container escapes and application runtime exploits.

Some of the attack vectors shown in Figure 1-2 are outside the scope of this book:

- Source code is generally held in repositories, which could conceivably be attacked to poison the application. Similarly, container images are held in registries. You will need to ensure that user access to the source repositories and image registries is controlled appropriately.
- The host machines on which your containers run are networked together, often using a VPC for security, and typically connected to the internet. Exactly as in a traditional deployment, you need to protect the host machines (or virtual machines) from access by threat actors. Secure network configuration, firewalling, and IAM all still apply in a cloud native deployment as they do in a traditional deployment. These protections also apply to the build machines on which the code is compiled and the container images are created.

- Containers typically run under an orchestrator. If the orchestrator is configured insecurely or if administrative access is not controlled effectively, this gives attackers additional vectors to interfere with the deployment.

In addition to this book, the following resources can help you assess the potential threats to your deployments:

- Microsoft published a good summary of the threats, tactics, and procedures that attackers might use against Kubernetes in the Threat Matrix for Kubernetes.
- Researchers from the Singapore University of Technology and Design and ETH Zurich published a paper titled "Threat Modeling and Security Analysis of Containers: A Survey".

Security Boundaries

A security boundary (sometimes called a *trust boundary*) appears between parts of the system, such that you would need some different set of permissions to move between those parts. Sometimes these boundaries are set up administratively. For example, in a Linux system, the system administrator can modify the security boundary defining what files a user can access by changing the groups that the user is a member of. If you are rusty on Linux file permissions, a refresher is coming up in Chapter 2.

You'll sometimes hear people saying that "Containers are not a security boundary." That's not really true—a container is a security boundary, just not a strong one! Application code is supposed to run within that container, and it should not be able to access code or data outside of the container except where it has explicitly been given permission to do so (for example, through a volume mounted into the container). As you'll learn in this book, because the boundary provided by a container is relatively weak, you'll need additional boundaries to have confidence in the security of your applications.

The more security boundaries there are between an attacker and their target (your customer data, for example), the harder it is for them to reach that target.

The attack vectors described in "Container Threat Model" on page 3 can be chained together to breach several security boundaries. For example:

- An attacker may find that because of a vulnerability in an application dependency, they are able to execute code remotely within a container.
- Suppose that the breached container doesn't have direct access to any data of value. The attacker needs to find a way to move out of the container, either to another container or to the host. A container escape vulnerability would be one route out of the container; insecure configuration of that container could provide another. If the attacker finds either of these routes available, they can now access the host.
- The next step would be to look for ways to gain root privileges on the host. This step might be trivial if your application code is running as root inside the container, as you'll see in Chapter 4.
- With root privileges on the host machine, the attacker can get to anything that the host, or any of the containers running on that host, can reach.

Adding and strengthening the security boundaries in your deployment will make life more difficult for the attacker.

An important aspect of the threat model is to consider the possibility of attacks from within the environment in which your applications are running. In cloud deployments, you may be sharing some resources with other users and their applications. Sharing machine resources is called *multitenancy*, and it has a significant bearing on the threat model.

Multitenancy

In a multitenant environment, different users, or *tenants*, run their workloads on shared hardware. (You may also come across the term *multitenancy* in a software application context, where it refers to multiple users sharing the same instance of software, but for the purposes of this discussion, only the hardware is shared.) Depending on who owns those different workloads and how much the different tenants trust each other, you might need stronger boundaries between them to prevent them from interfering with each other.

Multitenancy is a concept that has been around since the mainframe days in the 1960s, when customers rented CPU time, memory, and storage on a shared machine. This is not so very different from today's public clouds, like Amazon AWS, Microsoft Azure, and Google Cloud Platform, where customers rent CPU time, memory, and storage, along with other features and managed services. Since Amazon AWS launched EC2 in 2006, we have been able to rent virtual machine instances running on racks of servers in data centers around the world. There may be many virtual machines (VMs) running on a physical machine, and as a cloud customer operating a set of VMs, you have no idea who is operating the VMs that neighbor yours.

Shared Machines

There are situations in which a single Linux machine (or virtual machine) may be shared by many users. This is common in university settings, for instance, and this is a good example of true multitenancy, where users don't trust each other and, quite frankly, the system administrators don't trust the users. In this environment, Linux access controls are used to strictly limit user access. Each user has their own login ID, and the access controls of Linux are used to limit access to ensure, for example, that users can modify the files only in their own directories. Can you imagine the chaos if university students could read or—even worse—modify their classmates' files?

As you'll see in Chapter 4, all the containers running on the same host share the same kernel. If the machine is running the Docker daemon, any user who can issue `docker` commands effectively has root access, so a system administrator won't want to grant that to untrusted users.

In enterprise situations, and more specifically in cloud native environments, you are less likely to see this kind of shared machine. Instead, users (or teams of users who trust each other) will typically use their own resources allocated to them in the form of virtual machines.

Virtualization

Generally speaking, virtual machines are considered to be pretty strongly isolated from each other, by which we mean that it's unlikely that your neighbors can observe or interfere with the activities in your VMs. You can read more about how this isolation is achieved in Chapter 5. In fact, according to the accepted definition, virtualization doesn't count as multitenancy at all: multitenancy is when different groups of people share a single instance of the same software, and in virtualization, the users don't have access to the hypervisor that manages their virtual machines, so they don't share any software.

That's not to say that the isolation between virtual machines is perfect, and historically users have complained about "noisy neighbor" issues, where the fact that you are sharing a physical machine with other users can result in unexpected variances in performance. Netflix was an early adopter of the public cloud, and as discussed in a 2010 blog post, it was acknowledged that Netflix built systems that might deliberately abandon a subtask if it proved to be operating too slowly. Others have claimed that the noisy neighbor problem isn't a real issue.

There have also been cases of software vulnerabilities that could compromise the boundary between virtual machines. Cloud providers do additional hardening to mitigate against these hypervisor-level vulnerabilities.

For some applications and some organizations (especially government, financial, or healthcare), the consequences of a security breach are sufficiently serious to warrant

full physical separation. You can operate a private cloud, running in your own data center or managed by a service provider on your behalf, to ensure total isolation of your workloads. Private clouds sometimes come with additional security features such as additional background checks on the personnel who have access to the data center.

Google provides nice documentation of its data center security practices.

Many cloud providers have VM options where you are guaranteed to be the only customer on a physical machine. It's also possible to rent bare-metal machines operated by cloud providers. In both these scenarios, you will completely avoid the noisy neighbor issue, and you also have the advantage of the stronger security isolation between physical machines.

Whether you are renting physical or virtual machines in the cloud or using your own servers, if you're running containers, you may need to consider the security boundaries between multiple groups of users.

Container Multitenancy

As you'll see in Chapter 4, the isolation between containers is not as strong as that between VMs. While it does depend on your risk profile, it's unlikely that you want to use containers on the same machine as a party that you don't trust.

Even if all the containers running on your machines are run by you or by people you absolutely trust, you might still want to mitigate against the fallibility of humans by making sure your containers can't interfere with each other.

In Kubernetes, you can use *namespaces* to subdivide a cluster of machines for use by different individuals, teams, or applications.

The word *namespace* is an overloaded term. In Kubernetes, a namespace is a high-level abstraction that subdivides cluster resources that can have different Kubernetes access controls applied to them. In Linux, a namespace is a low-level mechanism for isolating the machine resources that a process is aware of. You'll learn about this kind of namespace in detail in Chapter 4.

Use role-based access control (RBAC) to limit the people and components that can access these different Kubernetes namespaces. The details of how to do this are outside the scope of this book, but I would like to mention that Kubernetes RBAC

controls only the actions you can perform through the Kubernetes API. Application containers in Kubernetes pods that happen to be running on the same host are protected from each other only by container isolation, as described in this book, even if they are in different namespaces. If an attacker can escape a container to the host, the Kubernetes namespace boundary makes not one jot of difference to their ability to affect other containers.

Container Instances

Cloud services such as Amazon AWS, Microsoft Azure, or Google Cloud Platform offer many *managed services*, through which the user can rent software, storage, and other components without having to install or manage them. A classic example is Amazon's Relational Database Service (RDS); with RDS, you can easily provision databases that use well-known software like PostgreSQL, and getting your data backed up is as simple as ticking a box (and paying a bill, of course).

Managed services have extended to the world of containers too. Azure Container Instances and AWS Fargate are services that allow you to run containers without having to worry about the underlying machine (or virtual machine) on which they run.

This can save you from a significant management burden and allows you to easily scale the deployment at will. However, at least in theory, your container instances could be colocated on the same virtual machine as those of other customers. Check with your cloud provider if in doubt.

You are now aware of a good number of potential threats to your deployment. Before we dive into the rest of the book, I'd like to introduce some basic security principles that should guide your thinking when assessing what security tools and processes you need to introduce into your deployment.

Security Principles

These are general guidelines that are commonly considered to be a wise approach regardless of the details of what you're trying to secure.

Least Privilege

The principle of least privilege states that you should limit access to the bare minimum that a person or component needs in order to do their job. For example, if you have a microservice that performs product search in an ecommerce application, the principle of least privilege suggests that the microservice should only have credentials that give it read-only access to the product database. It has no need to access, say, user or payment information, and it has no need to write product information.

Defense in Depth

As you'll see in this book, there are many different ways you can improve the security of your deployment and the applications running within it. The principle of defense in depth tells us that you should apply layers of protection. If an attacker is able to breach one defense, another layer should prevent them from harming your deployment or exfiltrating your data.

Reducing the Attack Surface

As a general rule, the more complex a system is, the more likely it is that there is a way to attack it. Eliminating complexity can make the system harder to attack. This includes:

- Reducing access points by keeping interfaces small and simple where possible
- Limiting the users and components who can access a service
- Minimizing the amount of code

Limiting the Blast Radius

The concept of segmenting security controls into smaller subcomponents, or *cells*, means that should the worst happen, the impact is limited. Containers are well suited to this principle, because by dividing an architecture into many instances of a microservice, the container itself can act as a security boundary.

Segregation of Duties

Related to both least privilege and limiting blast radius is the idea of segregating duties so that, as much as possible, different components or people are given authority over only the smallest subset of the overall system that they need. This approach limits the damage a single privileged user might inflict by ensuring that certain operations require more than one user's authority.

Applying Security Principles with Containers

As you'll see in later sections of this book, the granularity of containers can help us in the application of all these security principles:

Least privilege
: You can give different containers different sets of privileges, each minimized to the smallest set of permissions it needs to fulfill its function.

Defense in depth
: Containers give another boundary where security protections can be enforced.

Reducing the attack surface
: Splitting a monolith into simple microservices can create clean interfaces between them that may, if carefully designed, reduce complexity and hence limit the attack surface. There is a counterargument that adding a complex orchestration layer to coordinate containers introduces another attack surface.

Limiting the blast radius
: If a containerized application is compromised, security controls can help constrain the attack within the container and prevent it from affecting the rest of the system.

Segregation of duties
: Permissions and credentials can be passed only into the containers that need them, so that the compromise of one set of secrets does not necessarily mean that all secrets are lost.

These benefits sound good, but they are somewhat theoretical. In practice, they can easily be outweighed by poor system configuration, bad container image hygiene, or insecure practices. By the end of this book, you should be well armed to avoid the security pitfalls that can appear in a containerized deployment and take advantage of the benefits.

Summary

You've now got a high-level view of the kinds of attacks that can affect a container-based deployment and an introduction to the security principles that you can apply to defend against those attacks. In the rest of the book, you'll delve into the mechanisms that underpin containers so that you can understand how security tools and best-practice processes combine to implement those security principles.

CHAPTER 2

Linux System Calls, Permissions, and Capabilities

In most cases, containers run within a computer running a Linux operating system, and it's going to be helpful to understand some of the fundamental features of Linux so that you can see how they affect security and, in particular, how they apply to containers. I'll cover system calls, file-based permissions, and capabilities and conclude with a discussion of privilege escalation. If you're familiar with these concepts, feel free to skip to Chapter 3.

This is all important because *containers run Linux processes that are visible from the host*. A containerized process uses system calls and needs permissions and privileges in just the same way that a regular process does. But containers give us some new ways to control how these permissions are assigned at runtime or during the container image build process, which will have a significant impact on security.

System Calls

Applications run in what's called *user space*, which has a lower level of privilege than the operating system kernel. If an application wants to do something like access a file, communicate using a network, or even find the time of day, it has to ask the kernel to do it on the application's behalf. The programmatic interface that the user space code uses to make these requests of the kernel is known as the *system call* or *syscall* interface.

There are some 400+ different system calls, with the number varying according to the version of Linux kernel. Here are a few examples:

read
: Read data from a file.

write
: Write data to a file.

open
: Open a file for subsequent reading or writing.

execve
: Run an executable program.

chown
: Change the owner of a file.

clone
: Create a new process.

Application developers rarely if ever need to worry about system calls directly, as they are usually wrapped in higher-level programming abstractions. The lowest-level abstraction you're likely to come across as an app developer is the `glibc` or `musl` library, or the Golang `syscall` package. In practice, these are usually wrapped by higher layers of abstractions as well.

If you would like to learn more about system calls, check out my talk "A Beginner's Guide to Syscalls", available on O'Reilly's learning platform.

Application code uses system calls in exactly the same way whether it's running in a container or not, but as you will see later in this book, there are security implications to the fact that all the containers on a single host share—that is, they are making system calls to—the same kernel.

Not all applications need all system calls, so—following the principle of least privilege—there are Linux security features that allow users to limit the set of system calls that different programs can access. You'll see how these can be applied to containers in Chapter 10.

I'll return to the subject of user space and kernel-level privileges in Chapter 5. For now, let's turn to the question of how Linux controls permissions on files.

File Permissions

On any Linux system, whether you are running containers or not, file permissions are the cornerstone of security. There is a saying that in Linux, everything is a file. Application code, data, configuration information, logs, and so on—it's all held in files. Even physical devices like screens and printers are represented as files. Permissions on files determine which users are allowed to access those files and what actions they can perform on the files. These permissions are sometimes referred to as *discretionary access control*, or DAC.

Let's examine this a little more closely.

If you have spent much time in a Linux terminal, you will likely have run the `ls -l` command to retrieve information about files and their attributes (see Figure 2-1).

Figure 2-1. Linux file permissions example

In the example in Figure 2-1, you can see a file called `myapp` that is owned by a user called `liz` and is associated with the group `staff`. The permission attributes tell you what actions users can perform on this file, depending on their identity. There are nine characters in this output that represent the permissions attributes, and you should think of these in groups of three:

- The first group of three characters describes permissions for the user who owns the file (`liz` in this example).
- The second group gives permissions for members of the file's group (here, `staff`).
- The final set shows what any other user (who isn't `liz` or a member of `staff`) is permitted to do.

There are three actions that users might be able to perform on this file: read, write, or execute, depending on whether the *r*, *w*, and *x* bits are set. The three characters in each group represent bits that are either on or off, showing which of these three actions are permitted. A dash means that the bit isn't set.

In this example, only the owner of the file can write to it, because the *w* bit is set only in the first group, representing the owner permissions. The owner can execute the file, as can any member of the group `staff`. Any user is allowed to read the file, because the *r* bit is set in all three groups.

If you'd like more detail on Linux permissions and information on how to modify them using the `chmod` command, there is a good article in *Linux Journal*.

There's a good chance that you were already familiar with these *r*, *w*, and *x* bits, but that's not the end of the story. Permissions can be affected by the use of `setuid`, `setgid`, and *sticky bits*. The first two are important from a security perspective because they can allow a process to obtain additional permissions, which an attacker might use for malevolent purposes.

setuid and setgid

Normally, when you execute a file, the process that gets started inherits your user ID. If the file has the `setuid` bit set, the process will have the user ID of the file's owner. Let's see this in action using a copy of the `sleep` executable:

```
$ ls -l $(which sleep)
-rwxr-xr-x 1 root root 35336 Apr  5  2024 /usr/bin/sleep
$ cp /usr/bin/sleep ./mysleep
$ ls -l mysleep
-rwxr-xr-x 1 liz liz 35336 May  7 10:50 mysleep
```

The `ls` outputs show that the original version of the command is owned by root, but the copy is owned by my user called `liz`, who ran the `cp` command. Run this copy by executing `./mysleep 10`, and in a second terminal you can take a look at the running process: the `10` means you'll have 10 seconds to do this before the process terminates (I have removed some lines from this output for clarity):

```
$ ps -fC mysleep
UID          PID    PPID  C STIME TTY          TIME CMD
liz        37920   37876  0 06:07 pts/0    00:00:00 ./mysleep 10
```

This is running under my user name `liz`. Now let's modify the executable so that it is owned by root and has the `setuid` bit turned on:

```
$ sudo chown root ./mysleep
$ sudo chmod +s ./mysleep
$ ls -l mysleep
-rwsr-sr-x 1 root liz 35336 May  7 10:50 mysleep
```

As a regular, unprivileged user, run `./mysleep 10` again and look at the running processes again from the second terminal:

```
$ ps -fC mysleep
UID          PID    PPID  C STIME TTY          TIME CMD
root       37940   37876  0 06:09 pts/0    00:00:00 ./mysleep 10
```

The `mysleep` process has taken the user identity of root, from the owner of the `mysleep` executable file. This process now has all the privileges associated with that root user.

In this way, the `setuid` bit can be used to give a program privileges that it needs but that are not usually extended to regular users.

ping and setuid

Perhaps the canonical example of the `setuid` bit being used to add privileges was the executable `ping`, which needed permission to open raw network sockets in order to send and receive the Internet Control Message Protocol (ICMP) messages that it uses to check network connectivity.

An administrator might be happy for their users to run `ping`, but that doesn't mean they are comfortable letting users open raw network sockets for any other purpose they might think of. Instead, the `ping` executable was typically installed with the `setuid` bit set and owned by the root user so that `ping` can use privileges normally associated with root.

This is no longer needed since the addition of ICMP sockets in kernel version 5.6. Processes are allowed to open this type of socket, usable purely for ICMP protocol messages, so that they can run `ping` without needing any special privileges.

At the time of writing, most distributions don't yet use this ICMP sockets mechanism for `ping`. Instead, they give the `ping` executable permission to access raw sockets using a *capability* called `CAP_NET_RAW`. We'll look into this in more detail in "Linux Capabilities" on page 21.

Even before the `mysleep` example, you've already seen `setuid` in action. It's used by `sudo`, which is an executable owned by root:

```
$ ls -l $(which sudo)
-rwsr-xr-x 1 root root 277936 Apr  8  2024 /usr/bin/sudo
```

The `setuid` bit on `sudo` means that the executable runs as the root user, which matches what we saw in the output from `ps` earlier.

Once it's running, `sudo` goes on to check that the real user that invoked it actually has permissions to run `sudo`, by checking the `sudoers` file or some other configured security policy mechanism. If you want to explore this in more detail, `man sudo` has a good explanation.

As you saw with the `mysleep` example, a `setuid` executable can allow a user to escalate privileges. However, for a few executables, the results might not always be exactly what you expect! Let's explore what happens if we use the `setuid` bit on a copy of `bash`:

```
$ cp $(which bash) ./mybash
$ sudo chown root ./mybash
$ sudo chmod +s ./mybash
$ ls -l mybash
-rwsr-sr-x  1 root liz  1446024 May  7 15:33 mybash
```

Since this executable is owned by root and has `setuid`, it seems reasonable to imagine that when you run it, the process will be running as root. And yet, look what happens when you try it:

```
$ ./mybash
mybash-5.2$ whoami
liz
```

As you can see, the process is *not* running as root, even though the `setuid` bit is on and the file is owned by root. What's happening here? The answer is that in modern versions of `bash` (and several other interpreters like `python`, `node`, and `ruby`) the executable might start off running as root, but it explicitly resets its user ID to be that of the original user to avoid potential privilege escalations.

To explore this for yourself in more detail, you can use `strace` to see the system calls that the `bash` (or `mybash`) executable makes. Find the process ID of your shell (you can do this by running `echo $$`), and then in a second terminal run the following command:

```
$ sudo strace -f -p <shell process ID> \
-e trace=execve,getuid,setresuid,setuid
```

This will trace out system calls from within your first shell, including any executables running within it. Look for the `setresuid()` or `setuid()` system calls being used to reset the user ID.

Only a very few executables are written to reset the user ID in this way. As you saw with the copy of `sleep` from earlier in this chapter, the normal `setuid` behavior is simply to adopt the file owner's ID.

Now that you have experimented with the `setuid` bit, you are in a good position to consider its security implications.

Security Implications of setuid

The `setuid` bit allows someone to act as if they were a different user, which could give them access to different files, executables, and privileges that they are not supposed to have. Because `setuid` provides a dangerous pathway to privilege escalation, some container image scanners (covered in Chapter 8) will report on the presence of files with the `setuid` bit set.

As you've seen, modern versions of `bash` and most shells and interpreters reset their user ID to avoid being used for trivial privilege escalations. You can also prevent `setuid` from being used within a container using the `--security-opt no-new-privileges` option on a `docker run` command—I'll come back to this in Chapter 4. However, that won't stop an attacker from writing a `setuid` executable owned by root onto mounted directory on the host. You'll find an example of this in the `chapter2/setuid` directory of the GitHub repo that accompanies this book. Host volume mounts can lead to all sorts of attacks, and we'll discuss this more in Chapter 11.

The `setuid` bit dates from a time when privileges were much simpler—either your process had root privileges or it didn't. The `setuid` bit provided a mechanism for granting extra privileges to non-root users. Version 2.2 of the Linux kernel introduced more granular control over these extra privileges through *capabilities*.

Linux Capabilities

There are more than 40 different capabilities in today's Linux kernel. Capabilities can be assigned to a thread to determine whether that thread can perform certain actions. For example, a thread needs the `CAP_NET_BIND_SERVICE` capability to bind to a low-numbered (below 1024) port. `CAP_SYS_BOOT` exists so that arbitrary executables don't have permission to reboot the system. `CAP_SYS_MODULE` is needed to load or unload kernel modules, and `CAP_BPF` is needed to load eBPF programs.

Consult `man capabilities` on a Linux machine for detailed information on each individual capability.

I mentioned earlier that the `ping` tool uses the `CAP_NET_RAW` capability so that it can open a raw network socket.

Capabilities can be assigned to both files and processes. You can see the capabilities for a file using `getcap`, like this:

```
$ getcap $(which ping)
/usr/bin/ping cap_net_raw=ep
```

You can see the capabilities assigned to a process by using the `getpcaps` command. On an Ubuntu system, `journald` is an example that has several capabilities:

```
$ getpcaps $(pgrep journal)
307: cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_setgid,
cap_setuid,cap_sys_ptrace,cap_sys_admin,cap_audit_control,cap_mac_override,
cap_syslog,cap_audit_read=ep
```

Many processes typically won't have any capabilities. My current shell is an example:

```
$ getpcaps $$
22355: =
```

In the past, `getpcaps` assumed that if a process was running as root, it had all capabilities, so it would report the whole list. These days, `getpcaps` and other tools have been updated not to make this assumption, so processes running as root will typically appear with no capabilities.

You've seen that the executable file for `ping` has `CAP_NET_RAW` associated with it, so let's see the capabilities assigned to the process when we run it. Leave `ping` running in one terminal:

```
$ ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.162 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.188 ms
...
```

In a second terminal, get the process ID of this `ping` and check its capabilities:

```
$ getpcaps $(pgrep ping)
50394: =
```

How is `ping` successfully opening a (raw) socket if it doesn't have the `CAP_NET_RAW` capability required? And why doesn't it have that capability, when you saw earlier in this section that the executable file has it?

The answer can be found by tracing system calls with `strace`, much like earlier when examining how `bash` behaves. Terminate the `ping` command and find the process ID of the shell using `echo $$`. In a second terminal, run:

```
$ sudo strace -f -p <shell process ID> \
-e trace=capget,capset,socket
```

You'll see that `ping` is now capabilities-aware and deliberately discards `CAP_NET_RAW` once the socket is open and it has no further use for the capability (a few irrelevant calls and details have been omitted for clarity):

```
capget({version=_LINUX_CAPABILITY_VERSION_3, pid=0},     ❶
{effective=0, permitted=1<<CAP_NET_RAW, inheritable=0}) = 0
capset({version=_LINUX_CAPABILITY_VERSION_3, pid=0},     ❷
{effective=1<<CAP_NET_RAW, permitted=1<<CAP_NET_RAW, inheritable=0}) = 0
socket(AF_INET, SOCK_RAW, IPPROTO_ICMP) = 3              ❸
socket(AF_INET6, SOCK_RAW, IPPROTO_ICMPV6) = 4
capget({version=_LINUX_CAPABILITY_VERSION_3, pid=0}, {effective=1<<CAP_NET_RAW,
permitted=1<<CAP_NET_RAW, inheritable=0}) = 0
capset({version=_LINUX_CAPABILITY_VERSION_3, pid=0},     ❹
{effective=0, permitted=1<<CAP_NET_RAW, inheritable=0})
```

❶ The process checks that `CAP_NET_RAW` is in the set of permissions that it is permitted to use. The return code 0 tells it that it is.

❷ The process uses the `capset` system call to make that capability effective.

❸ It opens sockets for IPv4 and IPv6.

❹ Now that the sockets are open, it uses `capset` to remove the capability from its effective set.

By the time `getpcaps` inspected the process's capabilities, `CAP_NET_RAW` was no longer in effect for the process.

For a more in-depth discussion of the ways that file and process permissions interact, see Adrian Mouat's post on Linux capabilities in practice.

Following the principle of least privilege, it's a good idea to grant only the capabilities that are needed for a process to do its job. When you run a container, you get the option to control the capabilities that are permitted, as you'll see in Chapter 11. Now that you are familiar with the basic concepts of permissions and privileges in Linux, I'd like to turn to the idea of escalating privileges.

Privilege Escalation

The term *privilege escalation* means extending beyond the privileges you were supposed to have so that you can take actions that you shouldn't be permitted to take. To escalate their privileges, an attacker takes advantage of a system vulnerability or poor configuration to grant themselves extra permissions.

Oftentimes, the attacker starts as a nonprivileged user and wants to gain root privileges on the machine. A common method of escalating privileges is to look for software that's already running as root and then take advantage of known vulnerabilities in the software. For example, web server software might include a vulnerability that allows an attacker to remotely execute code, such as the Struts vulnerabilities.[1] If the web server is running as root, anything that is remotely executed by an attacker will run with root privileges. For this reason, it is a good idea to run software as a nonprivileged user whenever possible.

As you'll learn later in this book, by default, *containers run as root*. This means that compared with a traditional Linux machine, applications running in containers are far more likely to be running as root. An attacker who can take control of a process inside a container still has to somehow escape the container, but once they achieve that, they will be root on the host, and there is no need for any further privilege escalation. Chapter 11 discusses this in more detail.

Even if a container is running as a non-root user, there is potential for privilege escalation simply based on the Linux permissions mechanisms you have seen earlier in this chapter:

- Container images including executable files with the `setuid` bit
- Additional capabilities granted to a container running as a non-root user

You'll learn about approaches for mitigating these issues later in the book.

Summary

In this chapter, you have learned (or been reminded about) some fundamental Linux mechanisms that will be essential to understanding later chapters of this book. They also come into play in security in numerous ways; the container security controls that you will encounter are all built on top of these fundamentals.

Now that you understand some basic Linux security controls, it's time to start looking at the mechanisms that make up containers so that you can understand for yourself how root on the host and in the container are one and the same thing.

1 In the first edition of this book, I was referring to a 2018 critical remote code execution vulnerability that had received a lot of press coverage. It turns out there have been other serious Struts vulnerabilities since then, which is a good case study in how vulnerabilities continue to be found in widely used software packages and why it's important to keep updating your dependencies!

CHAPTER 3
Control Groups

In this chapter, you will learn about one of the fundamental building blocks that are used to make containers: *control groups*, more commonly known as *cgroups*.

Cgroups limit the resources, such as memory, CPU, and network input/output, that a group of processes can use. In containers, they are used to distribute resources across different workloads in a controlled fashion. From a security perspective, well-tuned cgroups can ensure that one process can't affect the behavior of other processes by hogging all the resources—for example, using all the CPU or memory to starve other applications. You can also limit the total number of processes allowed within a control group—a handy technique to protect against *fork bombs*, which I'll cover at the end of the chapter.

As you will see in detail in Chapter 4, containers run as regular Linux processes, so cgroups can be used to limit the resources available to each container.

Most Linux distributions today use cgroups version 2, which has some improvements over the original implementation that was widely deployed when containers first became popular. Cgroups v2 is now what's used by Kubernetes and all the popular container runtimes, and it's what is discussed here. However, you might find some references to v1 in older literature.

The main difference is that version 2 uses a single, unified hierarchy for managing all the supported resource types rather than having separate hierarchies for the different types of resource being managed.

If you're a system administrator looking after Linux servers directly, you might have reasons to create and manage control groups directly, but when we use containers, the container runtime takes care of this for us. As you'll see later in this chapter, all we have to do is specify any resources we want to allocate to different workloads. But let's dive in so that you can build an understanding of how cgroups are used to constrain resources for containers.

Control Group Controllers

Control groups are represented in the Linux filesystem under a mount point residing at `/sys/fs/cgroup`. Managing cgroups involves reading and writing to the files and directories under this mount point. Let's take a look at the contents of that directory:

```
root@vm:/sys/fs/cgroup# ls
cgroup.controllers      io.pressure
cgroup.max.depth        io.prio.class
cgroup.max.descendants  io.stat
cgroup.pressure         memory.numa_stat
cgroup.procs            memory.pressure
cgroup.stat             memory.reclaim
cgroup.subtree_control  memory.stat
cgroup.threads          memory.zswap.writeback
cpu.pressure            misc.capacity
cpu.stat                misc.current
cpu.stat.local          misc.peak
cpuset.cpus.effective   proc-sys-fs-binfmt_misc.mount
cpuset.cpus.isolated    sys-fs-fuse-connections.mount
cpuset.mems.effective   sys-kernel-config.mount
dev-hugepages.mount     sys-kernel-debug.mount
dev-mqueue.mount        sys-kernel-tracing.mount
init.scope              system.slice
io.cost.model           user.slice
io.cost.qos
```

As I'll show you shortly, a new control group can be created by making a new directory, which in turn can have child control groups created within it, building up a hierarchy of control groups.

The `cgroup.controllers` file shows what cgroup *controllers* are available on this machine:

```
root@vm:/sys/fs/cgroup# cat cgroup.controllers
cpuset cpu io memory hugetlb pids rdma misc
```

Each controller manages a type of resource that processes might want to consume. For example, the `cpu` controller manages the CPU usage of the processes in a cgroup, and the `memory` controller manages the memory they can access.

To take effect, controllers have to be enabled by writing the controller name into the `cgroup.subtree_control` file, and a controller can be enabled for a cgroup only if it is enabled in its parent. Every running Linux process is a member of exactly one control group, and you'll find the process IDs of all a cgroup's members listed in its `cgroup.procs` file.

Creating and Configuring Cgroups

Creating a subdirectory inside the `/sys/fs/cgroup` directory creates a cgroup, and the kernel automatically populates the directory with the various files that represent parameters and statistics about that group and its resources:

```
root@vm:/sys/fs/cgroup# mkdir liz
root@vm:/sys/fs/cgroup# ls liz
cgroup.controllers
cgroup.events
cgroup.freeze
...
pids.max
pids.peak
rdma.current
rdma.max
```

The details of what each of these different files means are beyond the scope of this book, but some of the files hold parameters that you can manipulate to define limits for the control group, and others communicate statistics about the current use of resources in the control group. You could probably make an educated guess that, for example, `memory.current` is the file that describes how much memory is currently being used by the control group. The maximum that the cgroup is allowed to use is defined by `memory.max`:

```
root@vm:/sys/fs/cgroup/liz# cat memory.max
max
```

The output `max` tells us that memory for this cgroup isn't limited—this is the default if a limit is not specified. If a process is allowed to consume unlimited memory, it can starve other processes on the same host. This might happen inadvertently through a memory leak in an application, or it could be the result of a resource exhaustion attack that takes advantage of a memory leak to deliberately use as much memory as possible.

By setting limits on the memory and other resources that one process can access, you can reduce the effects of this kind of attack and ensure that other processes can carry on as normal.

To set a limit for a cgroup, you simply have to write the value into the file that corresponds to the parameter you want to limit. Let's set the maximum available memory in bytes for the cgroup I just created:

```
root@vm:/sys/fs/cgroup/memory/liz# echo 100000 > memory.max
```

Now you'll find that the `memory.max` parameter is approximately what you configured as the limit—presumably, rounded down to the nearest page size:

```
root@vm:/sys/fs/cgroup/memory/liz# cat memory.max
98304
```

This illustrates how the limits are set for a group, but the final piece of the cgroups puzzle is to see how processes get assigned into cgroups.

Assigning a Process to a Cgroup

As mentioned earlier, the set of processes in a cgroup is listed in its `cgroup.procs` file. A new cgroup will start off with no processes, so that file will be empty.

When you start a process, it joins the cgroup of its parent, but you can move it into a new cgroup by simply writing its process ID into the `cgroup.procs` file of the group you want it to join. In the following example, 29903 is the process ID of a shell:

```
root@vm:/sys/fs/cgroup/memory/liz# echo 29903 > cgroup.procs
root@vm:/sys/fs/cgroup/memory/liz# cat cgroup.procs
29903
root@vm:/sys/fs/cgroup/memory/liz# cat /proc/29903/cgroup
0::/liz
```

The shell is now a member of the `liz` cgroup, with its memory limited to a little under 100kB. This isn't a lot to play with, so even trying to run `ls` from inside the shell breaches the cgroup limit:

```
$ ls
Killed
```

The process gets killed when it attempts to exceed the memory limit.

Cgroups for Containers

You've seen how cgroups are manipulated by modifying the files in the cgroup filesystem for a particular type of resource. It's straightforward to see this in action in Docker.

To follow along with these examples, you will need Docker running directly on a Linux (virtual) machine. If you're running Docker for Mac/Windows, it's running within a virtual machine, which means (as you'll see in Chapter 5) that these examples won't work for you, because the Docker daemon and containers are running using a separate kernel within that virtual machine.

Since most Linux distributions use `systemd` and because the `systemd` driver is the recommended default driver for cgroups in Kubernetes, my examples assume the `systemd` driver. Some readers might encounter the alternative `cgroupfs` driver, but for the purposes of this book, the only relevant difference is that the cgroups are created in a different place in the cgroup file system hierarchy.

Docker automatically creates a cgroup for each container, with a hierarchy that looks like this:

```
/sys/fs/cgroup/system.slice/
└── docker-<container_id>.scope/
    ├── cpu.max
    ├── memory.max
    ├── pids.max
    ├── cgroup.procs
    └── ...
```

The `system.slice` part of the hierarchy indicates that the cgroups are being managed using the `systemd` driver.

This example runs a container in the background with a limit of 100MiB (which is 104,857,600 bytes) of memory. As you'll see, Docker uses the cgroup mechanism to enforce this limit. The container will sleep for long enough for you to see its cgroup:

```
root@vm:~# docker run --rm --memory 100M -d alpine sleep 10000
68fb008c5fd3f9067e1aa245b4522a9f3675720d8953371ecfcf2e9faf91b8a0
root@vm:/sys/fs/cgroup# ls system.slice/docker-68fb...2e9faf91b8a0.scope
cgroup.controllers
cgroup.events
cgroup.freeze
...
```

Check the memory limit for and current usage by this container:

```
root@vm:/sys/fs/cgroup# cat system.slice/docker-68fb...scope/memory.max
104857600
root@vm:/sys/fs/cgroup# cat system.slice/docker-68fb...scope/memory.current
462848
```

You can also confirm that the sleeping process is a member of the cgroup:

```
root@vm:/sys/fs/cgroup# cat system.slice/docker-68fb...scope/cgroup.procs  19824
root@vm:/sys/fs/cgroup# pgrep sleep
19824
```

In Kubernetes you can set memory and CPU limits for pods and individual containers through the Pod specification. Once the pod is running, if you inspected the host's cgroup configuration, you would see those limits translated into cgroup settings.

Preventing a Fork Bomb

A fork bomb rapidly creates processes that in turn create more processes, leading to an exponential growth in the use of resources that ultimately cripples the machine. I'll show you how to reproduce this, but for caution's sake, please don't attempt running the fork bomb on a system that you can't risk bringing to its knees!

If you don't want to risk this yourself, you can watch a video of a talk I gave a few years back that includes a demonstration.

Earlier in this chapter, I created a cgroup called `liz` and set a memory limit. Let's remove the memory limit and instead define the maximum number of processes allowed in the cgroup:

```
root@vm:/sys/fs/cgroup/liz# echo max > memory.max
root@vm:/sys/fs/cgroup/liz# echo 20 > pids.max
```

Add the current shell to the cgroup:

```
root@vm:/sys/fs/cgroup/liz# echo $$ > cgroup.procs
```

Inspect the number of processes:

```
root@vm:/sys/fs/cgroup/liz# cat pids.current
2
```

Why are there two processes in this cgroup? The first is the shell, added explicitly by writing its process ID to the `cgroup.procs` file. Since a newly created process inherits the cgroup of its parent, the second process observed is the `cat` program running within the shell.

Now you can run a fork bomb—this syntax will work if your shell is `bash`:

```
root@vm:/sys/fs/cgroup/liz# :(){ :|:& };:
```

You should very soon see your terminal filling up with `bash: fork: retry: Resource temporarily unavailable` messages, as processes fail to be started due to the limit imposed by the cgroup. While this might be annoying in your terminal window, other processes on the machine will still be able to operate fine. If the fork bomb were allowed to keep creating new processes, you would see other operations grinding to a halt.

You can use the kill feature of cgroups to terminate the fork bomb. Unfortunately, the shell you started the fork bomb in will also be a casualty of this operation, which kills all the processes in this group:

```
root@vm:/sys/fs/cgroup/liz# echo 1 > cgroup.kill
```

Curious about how the fork bomb `:(){ :|:& };:` works? While this has nothing really to do with container security, it's a fun bit of syntax:

- `:() {...}` defines a function called `:` (yes, a colon is a valid name for a function in Bash).
- The content of the function is `:|:&`. Each `:` is a call to the function, so the function calls itself and pipes the output into another invocation of itself, with the `&` causing this second invocation to run in the background. Piping the output of a function causes it to be run in a new process, as does running a process in the background. As a result, each invocation of the `:` function spawns two processes, each executing the `:` function.
- The `;` terminates the function definition.
- The final `:` calls the function that has just been defined, kicking off an exponential cascade of process creation—until the cgroup process limit is hit.

Rather than manipulating cgroups directly, you can configure the process limit with the `--pids-limit` parameter on a `docker run` command. At the time of writing, in Kubernetes you can configure the maximum processes per pod through a Kubelet setting (which means the limit applies to all pods on a node rather than having individual limits set for each pod).

Summary

Cgroups limit the resources available to different Linux processes. You don't have to be using containers to take advantage of cgroups, but Docker, Kubernetes, and other container runtimes provide a convenient interface for using them: it's easy to set resource limits at the point where you run a container, and those limits are policed by cgroups.

Constraining resources provides protection against a class of attacks that attempt to disrupt your deployment by consuming excessive resources, thereby starving legitimate applications. It's recommended that you set memory and CPU limits when you run your container applications.

Now that you know how resources are constrained in containers, you are ready to learn about the other pieces of the puzzle that make up containers: namespaces and changing the root directory. Move on to Chapter 4 to find out how these work.

CHAPTER 4
Container Isolation

This is the chapter in which you'll find out how containers really work! This will be essential to understanding the extent to which containers are isolated from each other and from the host. You will be able to assess for yourself the strength of the security boundary that surrounds a container.

As you'll know if you have ever run `docker exec <image> bash`, a container looks a lot like a virtual machine from the inside. If you have shell access to a container and run `ps`, you can see only the processes that are running inside it. The container has its own network stack, and it seems to have its own filesystem with a root directory that bears no relation to root on the host. You can run containers with limited resources, such as a restricted amount of memory or a fraction of the available CPUs. This all happens using the Linux features that we're going to delve into in this chapter.

However much they might superficially resemble each other, it's important to realize that containers *aren't* virtual machines, and in Chapter 5 we'll take a look at the differences between these two types of isolation. In my experience, really understanding and being able to contrast the two is absolutely key to grasping the extent to which traditional security measures can be effective in containers and to identifying where container-specific tooling is necessary.

You'll see how containers are built out of Linux constructs such as namespaces and chroot, along with cgroups, which were covered in Chapter 3. With an understanding of these constructs, you'll have a feeling for how well protected your applications are when they run inside containers.

Although the general concepts of these constructs are fairly straightforward, the way they work together with other features of the Linux kernel can be complex. Container escape vulnerabilities (for example, CVE-2019-5736, a serious vulnerability discovered in both `runc` and `LXC`) have been based on subtleties in the way that namespaces, capabilities, and filesystems interact.

Linux Namespaces

If cgroups control the resources that a process can use, *namespaces* control what it can see. By putting a process in a namespace, you can restrict the resources that are visible to that process.

The origins of namespaces date back to the Plan 9 operating system. At the time, most operating systems had a single "name space" of files. Unix systems allowed the mounting of filesystems, but they would all be mounted into the same system-wide view of all filenames. In Plan 9, each process was part of a process group that had its own "name space" abstraction, the hierarchy of files (and file-like objects) that this group of processes could see. Each process group could mount its own set of filesystems without seeing each other.

The first namespace was introduced to the Linux kernel in version 2.4.19 back in 2002. This was the mount namespace, and it followed similar functionality to that in Plan 9. Nowadays there are several different kinds of namespaces supported by Linux:

- Unix Timesharing System (UTS)—this sounds complicated, but to all intents and purposes this namespace is really just about the hostname and domain names for the system that a process is aware of
- Process IDs
- Mount points
- Network
- User and group IDs
- Inter-process communications (IPC)
- Control groups (cgroups)
- Time

A process is always in exactly one namespace of each type. When you start a Linux system, it has a single namespace of each type, but as you'll see, you can create additional namespaces and assign processes into them. You can easily see the namespaces on your machine using the `lsns` command:

```
$ lsns
        NS TYPE   NPROCS    PID USER COMMAND
4026531834 time        3 848409 liz  -bash
4026531835 cgroup      3 848409 liz  -bash
4026531836 pid         3 848409 liz  -bash
4026531837 user        3 848409 liz  -bash
4026531838 uts         3 848409 liz  -bash
4026531839 ipc         3 848409 liz  -bash
4026531840 net         3 848409 liz  -bash
4026531841 mnt         3 848409 liz  -bash
```

This looks nice and neat, and there is one namespace for each of the types I mentioned previously. Sadly, this is an incomplete picture! The man page for `lsns` tells us that it "reads information directly from the */proc* filesystem and for non-root users it may return incomplete information." The additional namespaces you might see as root are not terribly interesting until you start creating some of your own, but I mentioned it to point out that when we are using `lsns`, we should run as root (or use `sudo`) to get the complete picture.

Let's explore how you can use namespaces to create something that behaves like what we call a *container*.

The examples in this chapter use Linux shell commands to create a container. If you would like to try creating a container using the Go programming language, you will find instructions at *https://github.com/lizrice/containers-from-scratch*.

Isolating the Hostname

Let's start with the namespace for the Unix Timesharing System (UTS). As mentioned previously, this covers the hostname and domain names. By putting a process in its own UTS namespace, you can change the hostname for this process independently of the hostname of the machine or virtual machine on which it's running.

If you open a terminal on Linux, you can see the hostname:

```
$ hostname
myhost
```

Most (perhaps all?) container systems give each container a random ID. By default, this ID is used as the hostname. You can see this by running a container and getting shell access. For example, in Docker, you could do the following:

```
$ docker run --rm -it --name hello ubuntu bash
root@cdf75e7a6c50:/$ hostname
cdf75e7a6c50
```

Incidentally, you can see in this example that even if you give the container a name in Docker (here I specified `--name hello`), that name isn't used for the hostname of the container.

The container can have its own hostname because Docker created it with its own UTS namespace. You can explore the same thing by using the `unshare` command to create a process that has a UTS namespace of its own.

As it's described on the man page (seen by running `man unshare`), `unshare` lets you "run a program with some namespaces unshared from the parent." Let's dig a little deeper into that description. When you "run a program," the kernel creates a new process and executes the program in it. This is done from the context of a running process—the *parent*—and the new process will be referred to as the *child*. The parent process is cloned (or *forked*) to create the child. The command is called "unshare" to indicate that, rather than sharing namespaces of its parent, the child is going to be given its own.

Let's give it a try. You need to have root privileges to do this, which is the reason for the `sudo` at the start of the line:

```
liz@myhost:~$ sudo unshare --uts sh
$ hostname
myhost
$ hostname experiment
$ hostname
experiment
$ exit
liz@myhost:~$ hostname
myhost
```

This runs a `sh` shell in a new process that has a new UTS namespace. Any programs you run inside the shell will inherit its namespaces. When you run the `hostname` command, it executes in the new UTS namespace that has been isolated from that of the host machine.

If you were to open another terminal window to the same host before the `exit`, you could confirm that the hostname hasn't changed for the whole (virtual) machine. You can change the hostname on the host without affecting the hostname that the namespaced process is aware of, and vice versa.

This is a key component of the way containers work. Namespaces give them a set of resources (in this case the hostname) that are independent of the host machine and of other containers. But we are still talking about a process that is being run by the same Linux kernel. This has security implications that I'll discuss later in the chapter. For now, let's look at another example of a namespace by seeing how you can give a container its own view of running processes.

Isolating Process IDs

If you run the `ps` command inside a Docker container, you can see only the processes running inside that container and none of the processes running on the host:

```
$ docker run --rm -it --name hello ubuntu bash
root@cdf75e7a6c50:/$ ps -eaf
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 18:41 pts/0    00:00:00 bash
root        10     1  0 18:42 pts/0    00:00:00 ps -eaf
```

This is achieved with the process ID namespace, which restricts the set of process IDs that are visible. Try running `unshare` again, but this time specifying that you want a new PID namespace with the `--pid` flag:

```
liz@myhost:~$ sudo unshare --pid sh
$ whoami
root
$ whoami
sh: 2: Cannot fork
$ whoami
sh: 3: Cannot fork
$ ls
sh: 4: Cannot fork
```

This doesn't seem very successful—it's not possible to run any commands after the first `whoami`! But there are some interesting artifacts in this output.

The first process under `sh` seems to have worked OK, but every command after that fails due to an inability to fork (the act of creating a child clone of a parent process). The error is output in the form `<command>: <process ID>: <message>`, and you can see that the process IDs are incrementing each time. Given the sequence, it would be reasonable to assume that the first `whoami` ran as process ID 1. That is a clue that the PID namespace is working in some fashion, in that the process ID numbering has restarted. But it's pretty much useless if you can't run more than one process!

There are clues to what the problem is in the description of the `--fork` flag in the man page for `unshare`: "Fork the specified program as a child process of unshare rather than running it directly. This is useful when creating a new PID namespace."

You can explore this by running `ps` to view the process hierarchy from a second terminal window:

```
liz@myhost:~$ ps fa
    PID TTY      STAT   TIME COMMAND
...
 924537 pts/0    Ss     0:00 -bash
 924718 pts/0    S+     0:00  \_ sudo unshare --pid sh
 924719 pts/1    Ss     0:00      \_ sudo unshare --pid sh
 924720 pts/1    S+     0:00          \_ sh
```

The sudo process forks itself to provide a monitor process and then goes on to execute unshare. The sh process is not a child of unshare; it's a child of the (second) sudo process.

If your version of sudo is 1.9.14 or above, you'll see two sudo processes, as in this example, but for older versions, there will be only one. The man page for sudo tells us that in newer versions, it forks one process in a new pseudo-terminal (pts/1 in my example output) to act as a monitor process, before forking a second time to run the command. You can check what version is installed on your machine by running sudo -V.

Now try unshare with the --fork parameter:

```
liz@myhost:~$ sudo unshare --pid --fork sh
$ whoami
root
$ whoami
root
```

This is progress, in that you can now run more than one command before running into the "Cannot fork" error. If you look at the process hierarchy again from a second terminal, you'll see an important difference:

```
liz@myhost:~$ ps fa
    PID TTY      STAT   TIME COMMAND
...
 924537 pts/0    Ss     0:00 -bash
 925113 pts/0    S+     0:00  \_ sudo unshare --pid --fork sh
 925114 pts/1    Ss     0:00      \_ sudo unshare --pid --fork sh
 925115 pts/1    S      0:00          \_ unshare --pid --fork sh
 925116 pts/1    S+     0:00              \_ sh
...
```

With the --fork parameter, the sh shell is running as a child of the unshare process, and you can successfully run as many different child commands as you choose within this shell. Given that the shell is within its own process ID namespace, the results of running ps inside it might be surprising:

```
liz@myhost:~$ sudo unshare --pid --fork sh
$ ps -eaf
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 Jul22 ?        00:00:09 /sbin/init
root           2       0  0 Jul22 ?        00:00:00 [kthreadd]
root           3       2  0 Jul22 ?        00:00:00 [pool_workqueue_release]
...many more lines of output about processes...
root       13172   12943  0 13:03 pts/2    00:00:00 unshare --pid --fork sh
root       13173   13172  0 13:03 pts/2    00:00:00 sh
root       13174   13173  0 13:03 pts/2    00:00:00 ps -eaf
```

As you can see, `ps` can show all the processes on the whole host, despite running inside a new process ID namespace. If you want the `ps` behavior that you would see in a Docker container, it's not sufficient just to use a new process ID namespace, and the reason for this is included in the man page for `ps`: "This `ps` works by reading the virtual files in `/proc`."

Let's take a look at the `/proc` directory to see what virtual files this is referring to. Your system will look similar but not exactly the same, as it will be running a different set of processes:

```
liz@myhost:~$ ls /proc
1      29      492628  64      acpi            loadavg
10     29375   492642  65      bootconfig      locks
10927  29451   492656  7       buddyinfo       mdstat
1181   3       492664  72      bus             meminfo
...many more lines...
```

Every numbered directory in `/proc` corresponds to a process ID, and there is a lot of interesting information about a process inside its directory. For example, `/proc/<pid>/exe` is a symbolic link to the executable that's being run inside this particular process, as you can see in the following example:

```
liz@myhost:~$ ps
  PID TTY          TIME CMD
28441 pts/0    00:00:00 bash
28558 pts/0    00:00:00 ps

liz@myhost:~$ ls /proc/28441
arch_status         fdinfo             ns              smaps_rollup
attr                gid_map            numa_maps       stack
autogroup           io                 oom_adj         stat
auxv                ksm_merging_pages  oom_score       statm
cgroup              ksm_stat           oom_score_adj   status
clear_refs          latency            pagemap         syscall
cmdline             limits             patch_state     task
comm                loginuid           personality     timens_offsets
coredump_filter     map_files          projid_map      timers
cpu_resctrl_groups  maps               root            timerslack_ns
cpuset              mem                sched           uid_map
cwd                 mountinfo          schedstat       wchan
environ             mounts             sessionid
exe                 mountstats         setgroups
fd                  net                smaps

liz@myhost:~$ ls -l /proc/28441/exe
lrwxrwxrwx 1 liz liz 0 Oct 10 13:32 /proc/28441/exe -> /usr/bin/bash
```

Irrespective of the process ID namespace it's running in, `ps` is going to look in `/proc` for information about running processes. To have `ps` return only the information about the processes inside the new namespace, there needs to be a separate copy of

the `/proc` directory, where the kernel can write information about the namespaced processes. Given that `/proc` is a directory directly under root, this means changing the root directory.

Changing the Root Directory

From within a container, you don't see the host's entire filesystem; instead, you see a subset, because the root directory gets changed as the container is created.

You can change the root directory in Linux with the `chroot` command. This effectively moves the root directory for the current process to point to some other location within the filesystem. Once you have done a `chroot` command, you lose access to anything that was higher in the file hierarchy than your current root directory, since there is no way to go any higher than root within the filesystem, as illustrated in Figure 4-1.

The description in `chroot`'s man page reads as follows: "Run COMMAND with root directory set to NEWROOT. […] If no command is given, run *${SHELL} -i* (default: */bin/sh -i*)."

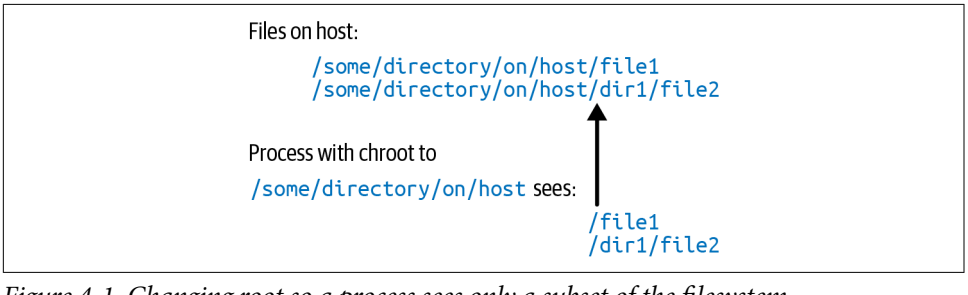


Figure 4-1. Changing root so a process sees only a subset of the filesystem

From this you can see that `chroot` doesn't just change the directory but also runs a command, falling back to running a shell if you don't specify a different command.

Create a new directory and try to `chroot` into it:

```
liz@myhost:~$ mkdir new_root
liz@myhost:~$ sudo chroot new_root
chroot: failed to run command '/bin/bash': No such file or directory
liz@myhost:~$ sudo chroot new_root ls
chroot: failed to run command 'ls': No such file or directory
```

This doesn't work! The problem is that once you are inside the new root directory, there is no `bin` directory inside this root, so it's impossible to run the `/bin/bash` shell. Similarly, if you try to run the `ls` command, it's not there. You'll need the files for any commands you want to run to be available within the new root. This is exactly what happens in a "real" container: the container is instantiated from a container image,

which encapsulates the filesystem that the container sees. If an executable isn't present within that filesystem, the container won't be able to find and run it.

Why not try running Alpine Linux within your container? Alpine is a fairly minimal Linux distribution designed for containers. You'll need to start by downloading the filesystem:[1]

```
liz@myhost:~$ mkdir alpine
liz@myhost:~$ cd alpine
liz@myhost:~/alpine$ curl -o alpine.tar.gz https://dl-cdn.alpinelinux.org/alpine/
latest-stable/releases/$(uname -m)/alpine-minirootfs-3.22.1-$(uname -m).tar.gz
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 3425k  100 3425k    0     0  22.1M      0 --:--:-- --:--:-- --:--:-- 22.3M
liz@myhost:~/alpine$ tar xvf alpine.tar.gz
```

At this point, you have a copy of the Alpine filesystem inside the `alpine` directory you created. Remove the compressed version and move back to the parent directory:

```
liz@myhost:~/alpine$ rm alpine.tar.gz
liz@myhost:~/alpine$ cd ..
```

You can explore the contents of the filesystem with `ls alpine` to see that it looks like the root of a Linux filesystem with directories such as `bin`, `lib`, `var`, `tmp`, and so on.

Now that you have the Alpine distribution unpacked, you can use `chroot` to move into the `alpine` directory, provided you supply a command that exists within that directory's hierarchy.

It's slightly more subtle than that, because the executable has to be in the new process's path. This process inherits the parent's environment, including the `PATH` environment variable. The `bin` directory within `alpine` has become `/bin` for the new process, and assuming that your regular path includes `/bin`, you can pick up the `ls` executable from that directory without specifying its path explicitly:

```
liz@myhost:~$ sudo chroot alpine ls
bin    etc    lib    mnt    proc   run    srv    tmp    var
dev    home   media  opt    root   sbin   sys    usr
liz@myhost:~$
```

Notice that it is only the child process (in this example, the process that ran `ls`) that gets the new root directory. When that process finishes, control returns to the parent process. If you run a shell as the child process, it won't complete immediately, so that makes it easier to see the effects of changing the root directory:

1 The `$(uname -m)` here identifies the architecture. For example, this will give `aarch64` if you're running on an ARM machine or `x86_64` on Intel. You'll also need to look up the latest release of Alpine. At the time of writing, it was v3.22.1, but that has likely moved on by the time you read this.

```
liz@myhost:~$ sudo chroot alpine sh
/ $ ls
bin    etc    lib    mnt    proc   run    srv    tmp    var
dev    home   media  opt    root   sbin   sys    usr
/ $ whoami
root
/ $ exit
liz@myhost:~$
```

If you try to run the `bash` shell, it won't work. This is because the Alpine distribution doesn't include it, so it's not present inside the new root directory. If you tried the same thing with the filesystem of a distribution like Ubuntu, which does include `bash`, it would work.

To summarize, `chroot` literally "changes the root" for a process. After changing the root, the process (and its children) will be able to access only the files and directories that are lower in the hierarchy than the new root directory.

In addition to `chroot`, there is a more sophisticated version called `pivot_root`. For the purposes of this chapter, whether `chroot` or `pivot_root` is used is an implementation detail; the key point is that a container needs to have its own root directory. I have used `chroot` in these examples because it is simpler and more familiar to many people.

There are security advantages to using `pivot_root` over `chroot`, so in practice you should find the former if you look at the source code of a container runtime implementation. The main difference is that `pivot_root` takes advantage of the mount namespace; the old root is no longer mounted and is therefore no longer accessible within that mount namespace. The `chroot` system call doesn't take this approach, leaving the old root accessible via mount points.

You have now seen how a container can be given its own root filesystem. I'll discuss this further in Chapter 6, but right now let's see how having its own root filesystem allows the kernel to show a container just a restricted view of namespaced resources.

Combine Namespacing and Changing the Root

So far you have seen namespacing and changing the root as two separate things, but you can combine the two by running `chroot` in a new namespace:

```
liz@myhost:~$ sudo unshare --pid --fork chroot alpine sh
/ $ ls
bin    etc    lib    mnt    proc   run    srv    tmp    var
dev    home   media  opt    root   sbin   sys    usr
```

If you recall from earlier in this chapter (see "Isolating Process IDs" on page 37), giving the container its own root directory allows it to create a `/proc` directory for the container that's independent of `/proc` on the host. For this to be populated with process information, you will need to mount it as a pseudo-filesystem of type `proc`. With the combination of a process ID namespace and an independent `/proc` directory, `ps` will now show just the processes that are inside the process ID namespace:

```
/ $ mount -t proc proc proc
/ $ ps
PID   USER     TIME  COMMAND
  1   root     0:00  sh
  5   root     0:00  ps
```

Success! It has been more complex than isolating the container's hostname, but through the combination of creating a process ID namespace, changing the root directory, and mounting a pseudo-filesystem to handle process information, you can limit a container so that it has a view only of its own processes.

If this seems complex, you might like to know that the `unshare` command has a `--mount-proc` option to simplify it:

```
liz@myhost:~$ sudo unshare --pid --fork --mount-proc bash
root@myhost:/home/liz# ps
    PID TTY          TIME CMD
      1 pts/4    00:00:00 bash
      8 pts/4    00:00:00 ps
```

There are more namespaces left to explore. Let's consider the mount namespace next.

Mount Namespace

Typically you don't want a container to have all the same filesystem mounts as its host. Giving the container its own mount namespace achieves this separation. Here's an example that creates a simple bind mount for a process with its own mount namespace:

```
liz@myhost:~$ sudo unshare --mount sh
$ mkdir source
$ touch source/HELLO
$ ls source
HELLO
$ mkdir target
$ ls target
$ mount --bind source target
$ ls target
HELLO
```

Once the bind mount is in place, the contents of the `source` directory are also available in `target`. If you look at all the mounts from within this process, there will probably be a lot of them, but the following command finds the target you created if you followed the preceding example:

```
$ findmnt target
TARGET       SOURCE                      FSTYPE OPTIONS
/home/liz/target
             /dev/sda1[/home/liz/source] ext4   rw,relatime,discard,
                                                errors=remount-ro, commit=30
```

From the host's perspective, this isn't visible, which you can prove by running the same command from another terminal window and confirming that it doesn't return anything.

Try running `findmnt` from within the mount namespace again but this time without any parameters, and you will get a long list. You might be thinking that it seems wrong for a container to be able to see all the mounts on the host. This is a similar situation to what you saw with the process ID namespace: the kernel uses the `/proc/<PID>/mounts` directory to communicate information about mount points for each process. If you create a process with its own mount namespace but using the host's `/proc` directory, you'll find that its `/proc/<PID>/mounts` file includes all the preexisting host mounts. (You can simply `cat` this file to get a list of mounts.)

To get a fully isolated set of mounts for the containerized process, you will need to combine creating a new mount namespace with a new root filesystem and a new `proc` mount, like this:

```
liz@myhost:~$ sudo unshare --mount chroot alpine sh
/ $ mount -t proc proc proc
/ $ mount
proc on /proc type proc (rw,relatime)
/ $ mkdir source
/ $ touch source/HELLO
/ $ mkdir target
/ $ mount --bind source target
/ $ mount
proc on /proc type proc (rw,relatime)
/dev/root on /target type ext4 (rw,relatime,discard,errors=remount-ro,commit=30)
```

Alpine Linux doesn't come with the `findmnt` command, so this example uses `mount` with no parameters to generate the list of mounts. (If you are cynical about this change, try the earlier example with `mount` instead of `findmnt` to check that you get the same results.)

You may be familiar with the concept of mounting host directories into a container using `docker run -v <host directory>:<container directory> ...`. To achieve this, after the root filesystem has been put in place for the container, the target

container directory is created and then the source host directory gets bind mounted into that target. Because each container has its own mount namespace, host directories mounted like this are not visible from other containers.

If you create a mount that is visible to the host, it won't automatically get cleaned up when your "container" process terminates. You will need to destroy it using `umount`. This also applies to the `/proc` pseudo-filesystems. They won't do any particular harm, but if you like to keep things tidy, you can remove them with `umount proc`. The system won't let you unmount the final `/proc` used by the host.

Network Namespace

The network namespace allows a container to have its own view of network interfaces and routing tables. When you create a process with its own network namespace, you can see it with `lsns`:

```
liz@myhost:~$ sudo lsns -t net
        NS TYPE NPROCS    PID USER    NETNSID NSFS COMMAND
4026531840 net     126      1 root unassigned      /sbin/init

liz@myhost:~$ sudo unshare --net bash
# lsns -t net
        NS TYPE NPROCS    PID USER    NETNSID NSFS COMMAND
4026531840 net     125      1 root unassigned      /sbin/init
4026532277 net       2  28586 root unassigned      bash
```

The output shows information about all the network namespaces on this host and the process IDs associated with them. I'll use the `bash` process ID, 28586, shortly.

You might come across the `ip netns` command, but that is not much use to us here. Using `unshare --net` creates an anonymous network namespace, and anonymous namespaces don't appear in the output from `ip netns list`.

When you put a process into its own network namespace, it starts with just the loopback interface:

```
# ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

With nothing but a loopback interface, your container won't be able to communicate. To give it a path to the outside world, you create a virtual Ethernet interface—or more strictly, a pair of virtual Ethernet interfaces. These act as if they were the two ends of a

metaphorical cable connecting your container namespace to the default network namespace on the host.

In a second terminal window on the host, you can create a virtual Ethernet pair by specifying the anonymous namespaces associated with their process IDs, like this:

```
liz@myhost:~$ sudo ip link add ve1 netns 28586 type veth \
peer name ve2 netns 1
```

- `ip link add` indicates that you want to add a link.
- `ve1` is the name of one "end" of the virtual Ethernet "cable."
- `netns 28586` says that this end is "plugged in" to the network namespace associated with process ID 28586 (which is shown in the output from `lsns -t net` you saw earlier).
- `type veth` shows that this is a virtual Ethernet pair.
- `peer name ve2` gives the name of the other end of the "cable."
- `netns 1` specifies that this second end is "plugged in" to the network namespace associated with process ID 1.

The `ve1` virtual Ethernet interface is now visible from inside the "container" process:

```
# ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: ve1@if3: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group ...
    link/ether 7a:8a:3f:ba:61:2c brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

The link is in "DOWN" state and needs to be brought up before it's of any use. Both ends of the connection need to be brought up.

Bring up the `ve2` end on the host:

```
liz@myhost:~$ sudo ip link set ve2 up
```

And once you bring up the `ve1` end in the container, the link should move to "UP" state:

```
# ip link set ve1 up
# ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: ve1@if3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP ...
    link/ether 7a:8a:3f:ba:61:2c brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::788a:3fff:feba:612c/64 scope link
       valid_lft forever preferred_lft forever
```

You can see that an IPv6 address has automatically been assigned to this interface in the container. Let's perform IPv6 ping from the host to that address within the container:

```
liz@myhost:~$ ping6 fe80::788a:3fff:feba:612c%ve2
PING fe80::788a:3fff:feba:612c%ve2 (fe80::788a:3fff:feba:612c) 56 data bytes
64 bytes from fe80::788a:3fff:feba:612c%ve2: icmp_seq=1 ttl=64 time=0.160 ms
64 bytes from fe80::788a:3fff:feba:612c%ve2: icmp_seq=2 ttl=64 time=0.052 ms
64 bytes from fe80::788a:3fff:feba:612c%ve2: icmp_seq=3 ttl=64 time=0.072 ms
```

Unlike IPv6, addresses are not automatically added to IPv4-capable interfaces, so if you want to send IPv4 traffic over the virtual Ethernet connection, you'll need to define the IPv4 address at either end. In the container:

```
# ip addr add 192.168.1.100/24 dev ve1
```

The kernel now knows that traffic from this network namespace to any `192.168.1.*` address should go out via the `ve1` interface, using the source IP address `192.168.1.100`. You can see this has been added into the routing table in the container:

```
# ip route
192.168.1.0/24 dev ve1 proto kernel scope link src 192.168.1.100
```

As mentioned at the start of this section, the network namespace isolates both the interfaces and the routing table, so this routing information in the container is independent of the IP routing table on the host. At this point the container can send traffic only to `192.168.1.0/24` addresses.

You also need to add an address on the host:

```
liz@myhost:~$ sudo ip addr add 192.168.1.200/24 dev ve2
```

Now you should be able to ping from within the container to the host:

```
# ping 192.168.1.200
PING 192.168.1.200 (192.168.1.200) 56(84) bytes of data.
64 bytes from 192.168.1.200: icmp_seq=1 ttl=64 time=0.355 ms
64 bytes from 192.168.1.200: icmp_seq=2 ttl=64 time=0.035 ms
^C
```

We will dig further into networking and container network security in Chapter 12.

User Namespace

The user namespace allows processes to use different IDs for users and groups inside the namespace, compared to the IDs used outside. Much like process IDs, the users and groups still exist on the host, but they can be mapped to different IDs within the namespace. The main benefit of this is that you can map the root ID of 0 within a container to some other non-root identity on the host. This is a huge advantage from a security perspective, since it allows software to run as root inside a container, but an attacker who escapes from the container to the host will have a non-root,

unprivileged identity. As you'll see in Chapter 11, it's not hard to misconfigure a container to make it easy to escape to the host. With user namespaces, you're not just one false move away from host takeover.

User namespace support is enabled by default from Kubernetes 1.33 onward, although you need a Linux kernel version 6.3 or newer. It's also supported in recent versions of container runtimes like `containerd` and `runc` and can be enabled in Docker using the `--userns-remap` flag on the daemon.

Generally speaking, you need to be root to create new namespaces (which is why the Docker daemon runs as root), but the user namespace is an exception:

```
liz@myhost:~$ unshare --user bash
nobody@myhost:/home/liz$ id
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup)
nobody@myhost:/home/liz$ echo $$
31196
```

I'll use the process ID of the current process, as returned by `echo $$`, in a moment. First, let's notice that inside the new user namespace, the user has the `nobody` ID. You need to put in place a mapping between user IDs inside and outside the namespace, as shown in Figure 4-2.

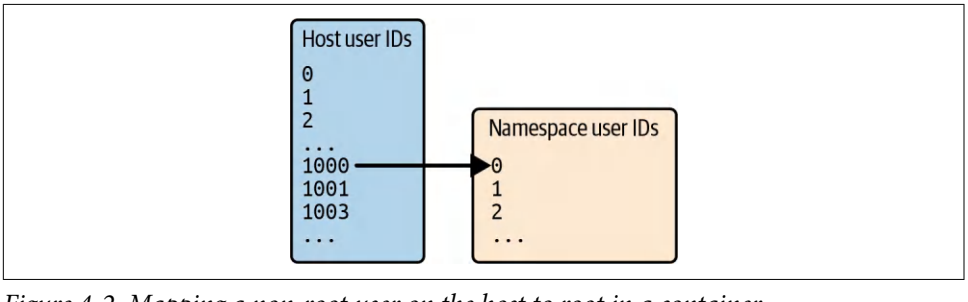


Figure 4-2. Mapping a non-root user on the host to root in a container

This mapping exists in `/proc/<pid>/uid_map`, which you can edit as root (on the host). There are three fields in this file:

- The lowest ID to map from the child process's perspective
- The lowest corresponding ID that this should map to on the host
- The number of IDs to be mapped

As an example, on my machine, the `liz` user has ID 1001. To have `liz` get assigned the root ID of 0 inside the child process, the first two fields are 0 and 1001. The last field can be 1 if you want to map only one ID (which may well be the case if you want

only one user inside the container). Here's the command I used in a second terminal window to set up that mapping:

```
liz@myhost:~$ sudo echo '0 1001 1' > /proc/31196/uid_map
```

Inside its user namespace, the process has taken on the root identity. Don't be put off by the fact that the `bash` prompt still says "nobody"; this doesn't get updated unless you rerun the scripts that get run when you start a new shell (e.g., `~/.bash_profile`):

```
nobody@myhost:/home/liz$ id
uid=0(root) gid=65534(nogroup) groups=65534(nogroup)
```

A similar mapping process using `/proc/<pid>/gid_map` can be used to map the group(s) used inside the child process.

So now the process is running under root's user ID, and in older versions of Linux, this used to be sufficient to get the full set of root's capabilities. In kernel 5.8, this was changed in important ways so that root in the child process no longer automatically gets the privileges of root across the whole host machine—it's merely a "namespace root." Let's explore what that ID looks like from the host's perspective. Start by running a `sleep` command:

```
nobody@myhost:/home/liz$ sleep 100
```

Notice that the prompt has not updated to show `root`.

In a second terminal, let's see what this process looks like:

```
liz@myhost:~$ ps -fC sleep
UID          PID    PPID  C STIME TTY          TIME CMD
liz        84714   84272  0 17:33 pts/0    00:00:00 sleep 100
```

The `sleep` command is being run under the unprivileged `liz` identity from the host's perspective, even though this looks like root inside the user namespace. This unprivileged user doesn't have the `CAP_SYS_ADMIN` capability required to create, say, a new UTS namespace:

```
nobody@myhost:~$ unshare --uts
unshare: unshare failed: Operation not permitted
```

Earlier in this chapter, in "Isolating the Hostname" on page 35, I mentioned you need to be root to run `unshare --uts` successfully. It would fail with "Operation not permitted" for exactly the same reason as in this case—no `CAP_SYS_ADMIN` capability. However, the kernel does permit a one-shot approach to creating other namespaces along with the user namespace. A regular, unprivileged user on the host can run a command like this:

```
liz@myhost:~$ unshare --user --uts sleep 100
```

Find the process ID for this sleep command in another terminal, and inspect its namespaces:

```
liz@myhost:~$ lsns -p 87982
        NS TYPE   NPROCS   PID USER COMMAND
4026531834 time        4 15244 liz  -bash
4026531835 cgroup      4 15244 liz  -bash
4026531836 pid         4 15244 liz  -bash
4026531839 ipc         4 15244 liz  -bash
4026531840 net         4 15244 liz  -bash
4026531841 mnt         4 15244 liz  -bash
4026532267 user        1 87982 liz  ├─sleep 100
4026532306 uts         1 87982 liz  └─sleep 100
```

You can see that the sleep process has inherited most of the namespaces, but it has its own user and UTS namespaces.

So an unprivileged user can create other namespaces after all! That seems great—except done like this, it's not terribly useful. No extra capabilities are given to the user, so trying to change the hostname won't be permitted:

```
liz@myhost:~$ unshare --user --uts hostname hello
hostname: you must be root to change the host name
```

Let's see what happens if you try this as a privileged user:

```
liz@myhost:~$ sudo unshare --user --uts bash
nobody@myhost:/home/liz$ hostname new
hostname: you must be root to change the host name
nobody@myhost:/home/liz$ id
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup)
```

The user inside the new namespace is `nobody`. There needs to be an explicit mapping to set the UID 0 on the host to UID 0 inside the user namespace. In a second terminal, you could achieve this by writing a *uid_map* file for the process similar to how we did earlier, or you can use a convenient `--map-root-user option` on the `unshare` command:

```
liz@myhost:~$ sudo unshare --user --uts --map-root-user bash
root@myhost:/home/liz# id
uid=0(root) gid=65534(nogroup) groups=65534(nogroup)
root@myhost:/home/liz# cat /proc/$$/uid_map
         0          0          1
root@myhost:/home/liz# hostname new
root@myhost:/home/liz# hostname
new
```

If you're running containers as a root user, you've seen that it's easy to get root privileges inside the container, and it's also easy to set up the container with a user namespace so that it doesn't automatically get root privileges. This is a security benefit because fewer containers need to run as "real" root (that is, root from the perspective of the host).

If you want to run containers as an unprivileged user *and* get root privileges inside the container, that's a bit more tricky. This concept is called *rootless containers*, and we'll cover this topic in Chapter 11.

Inter-Process Communications Namespace

In Linux it's possible to communicate between different processes by giving them access to a shared range of memory, or by using a shared message queue. The two processes need to be members of the same inter-process communications (IPC) namespace for them to have access to the same set of identifiers for these mechanisms.

Generally speaking, you *don't* want your containers to be able to access one another's shared memory, so they are given their own IPC namespaces.

You can see this in action by creating a shared memory block and then viewing the current IPC status with `ipcs`:

```
$ ipcmk -M 1000
Shared memory id: 0
$ ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x74e9655a 0          root       644        1000       0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

In this example, the newly created shared memory block (with its ID in the `shmid` column) appears in the "Shared Memory Segments" block. A process with its own IPC namespace does not see any of these IPC objects:

```
$ sudo unshare --ipc sh
$ ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

Cgroup Namespace

The cgroup namespace is a little bit like a `chroot` for the cgroup filesystem; it stops a process from seeing the cgroup configuration higher up in the hierarchy of cgroup directories than its own cgroup.

This section assumes you're using cgroups v2. If you need to revisit how they work, you'll find them discussed in Chapter 2.

You can see the cgroup namespace in action by comparing the contents of `/proc/self/cgroup` outside and then inside a cgroup namespace:

```
liz@myhost:~$ cat /proc/self/cgroup
0::/user.slice/user-1001.slice/session-357.scope

liz@myhost:~$ sudo unshare --cgroup bash
root@myhost:/home/liz# cat /proc/self/cgroup
0::/
```

The process sees a root-level cgroup. However, this process has full access to the root filesystem, so looking at `/sys/fs/cgroup` shows the host's cgroup hierarchy. For example, looking at the contents of `/sys/fs/cgroup/cgroup.procs` would show a lot of processes that have nothing to do with this process and its own control group. This is similar to how a container needs its own view of `/proc` to get a correct view of the processes inside its process ID namespace; it also needs its own version of `/sys/fs/cgroup`. As before, you'll need to create a mount namespace and change the root directory. In this example, I am using the `alpine` root filesystem that we used earlier:

```
liz@myhost:~$ sudo unshare --cgroup --mount chroot alpine sh
# mkdir -p /sys/fs/cgroup
# mount -t cgroup2 none /sys/fs/cgroup
# ls /sys/fs/cgroup
cgroup.controllers      cpu.stat.local          memory.reclaim
cgroup.events           cpu.uclamp.max          memory.stat
cgroup.freeze           cpu.uclamp.min          memory.swap.current
cgroup.kill             cpu.weight              memory.swap.events
cgroup.max.depth        cpu.weight.nice         memory.swap.high
cgroup.max.descendants  io.pressure             memory.swap.max
cgroup.pressure         memory.current          memory.swap.peak
cgroup.procs            memory.events           memory.zswap.current
cgroup.stat             memory.events.local     memory.zswap.max
cgroup.subtree_control  memory.high             memory.zswap.writeback
cgroup.threads          memory.low              pids.current
cgroup.type             memory.max              pids.events
cpu.idle                memory.min              pids.events.local
cpu.max                 memory.numa_stat        pids.max
```

```
cpu.max.burst           memory.oom.group        pids.peak
cpu.pressure            memory.peak
cpu.stat                memory.pressure
```

The cgroup pseudo-filesystem has been populated with all the information you might expect, but there is still one problem: the process IDs shown here reflect the host's view of processes. You'll need a process ID namespace, too, for this to work completely as expected:

```
liz@myhost:~$ sudo unshare --mount --pid --fork --cgroup bash
root@myhost:/home/liz# mount -t proc proc alpine/proc
root@myhost:/home/liz# mount -t cgroup2 none alpine/sys/fs/cgroup
root@myhost:/home/liz# chroot alpine sh
# ps
PID   USER     TIME  COMMAND
    1 root      0:00 bash
   22 root      0:00 sh
   23 root      0:00 ps
# cat /sys/fs/cgroup/cgroup.procs
0
0
0
0
0
0
1
22
26
```

This looks pretty consistent with what you might expect, but what are those 0 entries in the `cgroup.procs` file? The answer is that these are processes in this cgroup that are outside the process ID namespace. The child process has its own view of cgroups, but it is still a member of the cgroup of its parent. The parent process can create a cgroup for the child by creating a new directory in `/sys/fs/cgroup` and writing the child's process ID into `cgroup.procs`.

Time Namespace

Using the time namespace, a process can adjust its own `CLOCK_MONOTONIC` and `CLOCK_BOOTTIME`, making it seem as if the system booted at a different time. It's intended for seamless process migration between systems, allowing timers and sleeps to pick up where they left off, and it can be used to reproduce issues that are time dependent, if a variable's value is generated based on time.

But can you imagine the confusion caused if different containers in a distributed system all have a different view of time? For starters, trying to coordinate logs and metrics across different containers would get really complex! It could also open up opportunities for an attacker to obfuscate malicious activity by making it appear to

happen in the past or the future. For this reason I'm not aware of any container systems that make use of the time namespace.

You have now explored all the different types of namespace and have seen how they are used along with `chroot` to isolate a process's view of its surroundings. Combine this with what you learned about cgroups in the Chapter 3, and you should have a good understanding of everything that's needed to make what we call a *container*.

Kubernetes Pods and Container Namespaces

A *pod* in Kubernetes is a group of one or more containers. Each of these containers has its own root filesystem, so they can have separate sets of dependencies, but they are not entirely isolated from each other using namespaces:

- The containers in a pod share a network namespace, giving them the same IP address and allowing them to communicate over localhost.
- A pod's containers have a common IPC namespace and can communicate with each other over UNIX domain sockets and shared memory.
- Optionally, a pod can be configured to share the Process ID namespace so that containers can see and send signals to each other's processes.

Before moving on to Chapter 5, it's worth taking a look at a container from the perspective of the host it's running on.

Container Processes from the Host Perspective

Although they are called containers, it might be more accurate to use the term *containerized processes*. A container is still a Linux process running on the host machine, but it has a limited view of that host machine, and it has access to only a subtree of the filesystem and perhaps to a limited set of resources restricted by cgroups. Because it's really just a process, it exists within the context of the host operating system, and it shares the host's kernel, as shown in Figure 4-3.

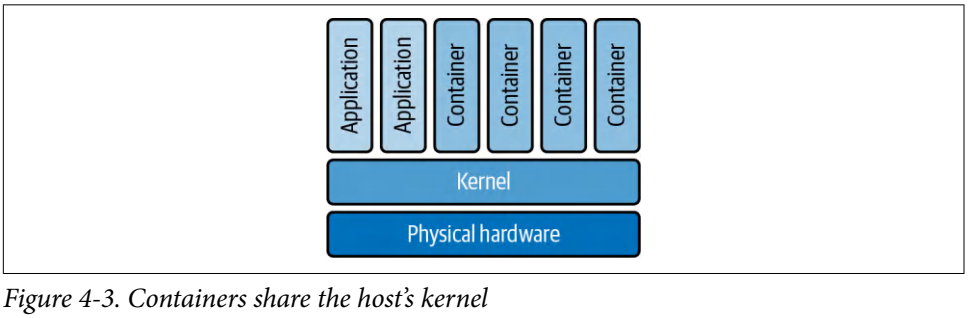


Figure 4-3. Containers share the host's kernel

You'll see how this compares to virtual machines in the Chapter 5, but before that, let's examine in more detail the extent to which a containerized process is isolated from the host, and from other containerized processes on that host, by trying some experiments on a Docker container.

This example runs a shell process in a container based on Ubuntu (or your favorite Linux distribution) and runs the `sleep` command for 1,000 seconds within the shell. Note that the `sleep` command is running as a process inside the container. When you press Enter at the end of the `sleep` command, this triggers Linux to clone a new process with a new process ID and to run the `sleep` executable within that process.

You can put the `sleep` process into the background (`Ctrl-Z` to pause the process, and `bg %1` to background it). Now run `ps` inside the container to see that `sleep` process from the container's perspective:

```
$ docker run --rm -it ubuntu bash
root@ab6ea36fce8e:/$ sleep 1000
^Z
[1]+  Stopped                 sleep 1000
root@ab6ea36fce8e:/$ bg %1
[1]+ sleep 1000 &
root@ab6ea36fce8e:/$ ps
  PID TTY          TIME CMD
    1 pts/0    00:00:00 bash
   10 pts/0    00:00:00 sleep
   11 pts/0    00:00:00 ps
root@ab6ea36fce8e:/$
```

While that `sleep` command is still running, open a second terminal into the same host and look at the same `sleep` process from the host's perspective:

```
$ ps -C sleep
  PID TTY          TIME CMD
30591 pts/0    00:00:00 sleep
```

The `-C sleep` parameter specifies that we are interested only in processes running the `sleep` executable.

The container has its own process ID namespace, so it makes sense that its processes would have low numbers, and that is indeed what you see when running `ps` in the container. From the host's perspective, however, the `sleep` process has a different, high-numbered process ID. There is just one process running `sleep`, and it has ID 30591 on the host and 10 in the container. (The actual number on the host will vary according to what else is and has been running on the machine, but it's likely to be a much higher number than the ID in the container.)

To get a good understanding of containers and the level of isolation they provide, it's really key to get to grips with the fact that although there are two different process

IDs, they both refer to *the same process*. It's just that from the host's perspective, it has a higher process ID number.

The fact that container processes are visible from the host is one of the fundamental differences between containers and virtual machines. An attacker who gets access to the host can observe and affect *all the containers running on that host*, especially if they have root access. And as you'll see in Chapter 11, there are some remarkably easy ways you can inadvertently make it possible for an attacker to move from a compromised container onto the host.

Container Host Machines

As you have seen, containers and their host share a kernel, and this has some consequences for what are considered best practices relating to the host machines for containers. If a host gets compromised, all the containers on that host are potential victims, especially if the attacker gains root or otherwise elevated privileges (such as being a member of the `docker` group that can administer containers where Docker is used as the runtime).

It's highly recommended to run container applications on dedicated host machines (whether they be VMs or bare metal), and the reasons mostly relate to security:

- Using an orchestrator to run containers means that humans need little or no access to the hosts. If you don't run any other applications, you will need a very small set of user identities on the host machines. These will be easier to manage, and attempts to log in as an unauthorized user will be easier to spot.
- You can use any Linux distribution as the host OS for running Linux containers, but there are several "Thin OS" distros specifically designed for running containers. These reduce the host attack surface by including only the components required to run containers. Examples include Flatcar, Talos, and Bottlerocket. With fewer components included in the host machine, there is a smaller chance of vulnerabilities (see Chapter 8) in those components.
- All the host machines in a cluster can share the same configuration, with no application-specific requirements. This makes it easy to automate the provisioning of host machines, and it means you can treat host machines as immutable. If a host machine needs an upgrade, you don't patch it; instead, you remove it from the cluster and replace it with a freshly installed machine. Treating hosts as immutable makes intrusions easier to detect.

I'll come back to the advantages of immutability in Chapter 8.

Using a Thin OS reduces the set of configuration options but doesn't eliminate them completely. For example, you will have a container runtime (perhaps `containerd`) plus orchestrator code (perhaps the Kubernetes `kubelet`) running on every host.

These components have numerous settings, some of which affect security. The Center for Internet Security (CIS) publishes benchmarks for best practices for configuring and running various software components, including Docker, Kubernetes, and Linux.

In an enterprise environment, look for a container security solution that also protects the hosts by reporting on vulnerabilities and worrisome configuration settings. You will also want logs and alerts for logins and login attempts at the host level.

Summary

Congratulations! Since you've reached the end of this chapter, you should now know what a container really is. You've seen the three essential Linux kernel mechanisms that are used to limit a process's access to host resources:

- Namespaces limit what the container process can see—for example, by giving the container an isolated set of process IDs.
- Changing the root limits the set of files and directories that the container can see.
- Cgroups control the resources the container can access.

As you saw in Chapter 1, isolating one workload from another is an important aspect of container security. You now should be fully aware that all the containers on a given host (whether it is a virtual machine or a bare-metal server) share the same kernel. Of course, the same is true in a multiuser system where different users can log in to the same machine and run applications directly. However, in a multiuser system, the administrators are likely to limit the permissions given to each user; they certainly won't give them all root privileges. With containers—at least at the time of writing—they all run as root by default and are relying on the boundary provided by namespaces, changed root directories, and cgroups to prevent one container from interfering with another.

In Chapter 10 we'll explore options for strengthening the security boundary around each container, but next let's delve into how virtual machines work. This will allow you to consider the relative strengths of the isolation between containers and between VMs, especially through the lens of security.

CHAPTER 5

Virtual Machines

Containers are often compared with VMs, especially in terms of the isolation that they offer. Let's make sure you have a solid understanding of how VMs operate so that you can reason about the differences between them and containers. This will be particularly useful when you want to assess the security boundaries around your applications when they run in containers or in different VMs. When you are discussing the relative merits of containers from a security perspective, understanding how they differ from VMs can be a useful tool.

This isn't a clear-cut distinction, really. As you'll see in Chapter 10, there are several sandboxing tools that strengthen the isolation boundaries around containers, making them more like VMs. If you want to understand the security pros and cons of these approaches, it's best to start with a firm understanding of the difference between a VM and a "normal" container.

The fundamental difference is that a VM runs an entire copy of an operating system, including its kernel, whereas a container shares the host machine's kernel. To understand what that means, you'll need to know something about how virtual machines are provided by a virtual machine monitor (VMM). Let's start to set the scene for that by thinking about what happens when a computer boots up.

Booting Up a Machine

Picture a physical server. It has some CPUs, memory, and networking interfaces. When you first boot up the machine, an initial program runs what's called the Basic Input Output System (BIOS). It scans how much memory is available, identifies the network interfaces, and detects any other devices such as displays, keyboards, attached storage devices, and so on.

In practice, a lot of this functionality has been superseded nowadays by Unified Extensible Firmware Interface (UEFI), but for the sake of argument, let's just think of this as a modern BIOS.

Once the hardware has been enumerated, the system runs a bootloader that loads and then runs the operating system's kernel code. The operating system could be Linux, Windows, or some other OS. As you saw in Chapter 2, kernel code operates at a higher level of privilege than your application code. This privilege level allows it to interact with memory, network interfaces, and so on, whereas applications running in user space can't do this directly.

On an x86 processor, privilege levels are organized into *rings*, with Ring 0 being the most privileged and Ring 3 being the least privileged. For most operating systems in a regular setup (without VMs), the kernel runs at Ring 0, and user space code runs at Ring 3, as shown in Figure 5-1.

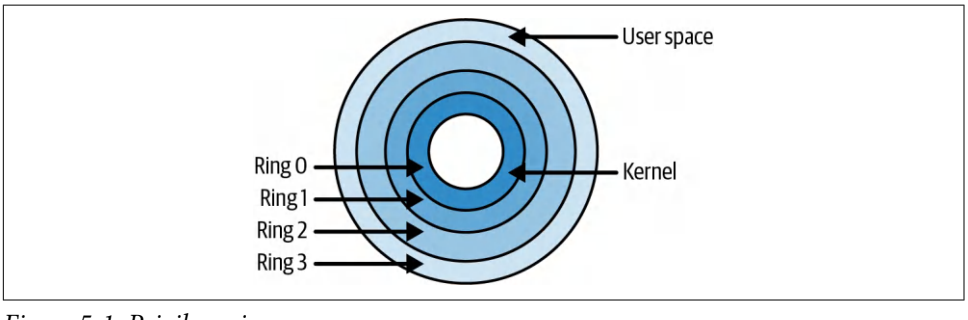


Figure 5-1. Privilege rings

Kernel code (like any code) runs on the CPU in the form of machine code instructions, and these instructions can include privileged instructions for accessing memory, starting CPU threads, and so on. The details of everything that can and will happen while the kernel initializes are beyond the scope of this book, but essentially the goal is to mount the root filesystem, set up networking, and bring up any system daemons. If you want to dive deeper, there is a lot of great information written by Alex Kuleshov on Linux kernel internals, including the bootstrap process, on GitHub.

Once the kernel has finished its own initialization, it can start running programs in user space. The kernel is responsible for managing everything that the user space programs need. It starts, manages, and schedules the CPU threads that these programs run in, and it keeps track of these threads through its own data structures that represent processes. One important aspect of kernel functionality is memory management. The kernel assigns blocks of memory to each process and makes sure that processes can't access one another's memory blocks.

Enter the VMM

As you have just seen, in a regular setup, the kernel manages the machine's resources directly. In the world of virtual machines, a *virtual machine monitor* (VMM) does the first layer of resource management, splitting up the resources and assigning them to virtual machines. Each virtual machine gets a kernel of its own.

You might also see the term *virtual machine manager*. This usually refers to a user-facing GUI or CLI tool used to manage virtual machines rather than the component that provides the virtualization. One well-known example is `virt-manager`, the vSphere client. For developers running Linux VMs on macOS, `lima` is a handy CLI tool offering VM management, while QEMU provides the virtualization.

For each virtual machine that it manages, the VMM assigns some memory and CPU resources, sets up virtual network interfaces and other virtual devices, and starts a guest kernel with access to these resources.

In a regular server, the BIOS gives the kernel the details of the resources available on the machine; in a virtual machine situation, the VMM divides up those resources and gives each guest kernel only the details of the subset that it is being given access to. From the perspective of the guest OS, it thinks it has direct access to physical memory and devices, but in fact it's getting access to an abstraction provided by the VMM.

The VMM is responsible for making sure that the guest OS and its applications can't breach the boundaries of the resources it has been allocated. For example, the guest operating system is assigned a range of memory on the host machine. If the guest somehow tries to access memory outside that range, this is forbidden.

There are two main forms of VMM, often called, not very imaginatively, Type 1 and Type 2. And there is a bit of gray area between the two, naturally!

Type 1 VMMs, or Hypervisors

In a regular system, the bootloader runs an operating system kernel like Linux or Windows. In a pure Type 1 virtual machine environment, a dedicated kernel-level VMM program runs instead.

Type 1 VMMs are also known as *hypervisors*, and examples include Hyper-V, Xen, and ESX/ESXi. As you can see in Figure 5-2, the hypervisor runs directly on the hardware (or *bare metal*), with no operating system underneath it.

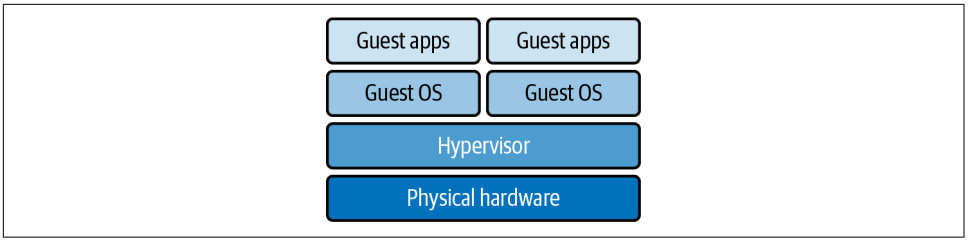


Figure 5-2. Type 1 VMM, also known as a hypervisor

In saying "kernel level," I mean that the hypervisor runs at Ring 0. (Well, that's true until we consider hardware virtualization later in this chapter, but for now let's just assume Ring 0.) The guest OS kernel runs at Ring 1, as depicted in Figure 5-3, which means it has less privilege than the hypervisor.

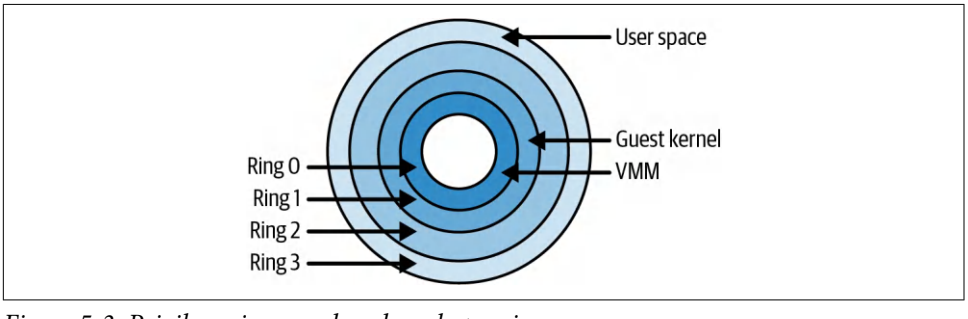


Figure 5-3. Privilege rings used under a hypervisor

Type 2 VMM

When you run virtual machines on your laptop or desktop machine, perhaps through something like QEMU, Parallels, or VirtualBox, they are "hosted," or Type 2, VMs. Your laptop might be running, say, macOS, which is to say that it's running a macOS kernel. You could install QEMU and use it to run guest VMs that coexist with your host operating system. Those guest VMs could be running a completely different operating system like Linux or Windows. Figure 5-4 shows how the guest OS and host OS coexist.

Consider that for a moment and think about what it means to run, say, Linux within macOS. By definition this means there has to be a Linux kernel, and that has to be a different kernel from the host's macOS kernel.

The VMM application has user space components that you can interact with as a user, but it also installs privileged components allowing it to provide virtualization. You'll see more about how this works later in this chapter.

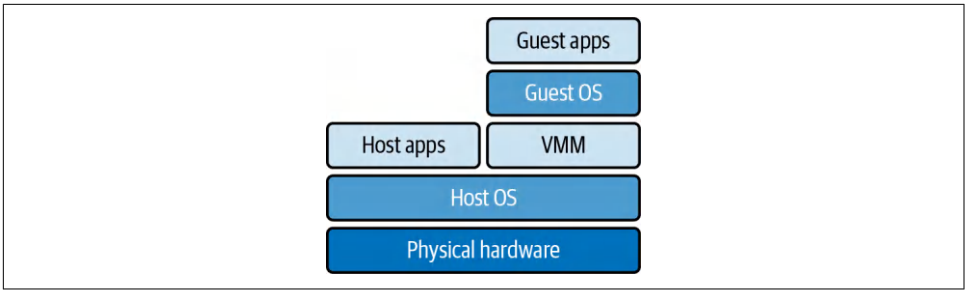


Figure 5-4. Type 2 VMM

Kernel-Based Virtual Machines

I promised that there would be some blurred boundaries between Type 1 and Type 2. In Type 1, the hypervisor runs directly on bare metal; in Type 2, the VMM runs in user space on the host OS. What if you run a virtual machine monitor within the kernel of the host OS?

This is exactly what happens with a Linux kernel module called KVM, or kernel-based virtual machines, as shown in Figure 5-5.

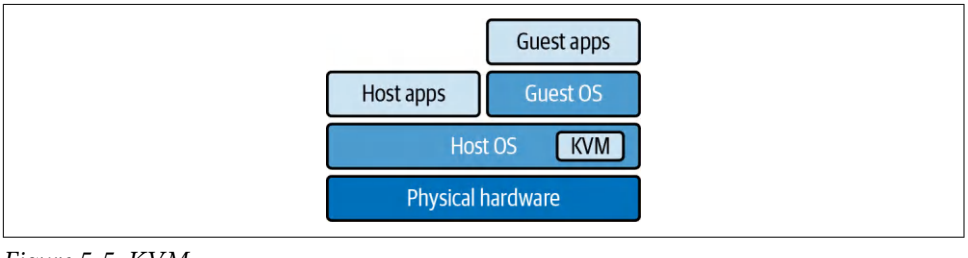


Figure 5-5. KVM

Generally, KVM is considered to be a Type 1 hypervisor because the guest OS doesn't have to traverse the host OS, but I'd say that this categorization is overly simplistic.

KVM is often used with QEMU (Quick Emulator), which I listed earlier as a Type 2 hypervisor. QEMU dynamically translates system calls from the guest OS into host OS system calls. It's worth a mention that QEMU can take advantage of hardware acceleration offered by KVM.

Whether Type 1, Type 2, or something in between, VMMs employ similar techniques to achieve virtualization. The basic idea is called *trap-and-emulate*, though as we'll see, x86 processors provide some challenges in implementing this idea.

Trap-and-Emulate

Some CPU instructions are *privileged*, meaning they can be executed only in Ring 0; if they are attempted in a higher ring, this will cause a *trap*. You can think of the trap as being like an exception in application software that triggers an error handler; a trap will result in the CPU calling a handler in the Ring 0 code.

If the VMM runs at Ring 0 and the guest OS kernel code runs at a lower privilege, a privileged instruction run by the guest can invoke a handler in the VMM to emulate the instruction. In this way the VMM can ensure that the guest OSs can't interfere with each other through privileged instructions.

Unfortunately, privileged instructions are only part of the story. The set of CPU instructions that can affect the machine's resources is known as *sensitive*. The VMM needs to handle these instructions on behalf of the guest OS, because only the VMM has a true view of the machine's resources. There is also another class of sensitive instructions that behaves differently when executed in Ring 0 or in lower-privileged rings. Again, a VMM needs to do something about these instructions because the guest OS code was written assuming the Ring 0 behavior.

If all sensitive instructions were privileged, this would make life relatively easy for VMM programmers, as they would just need to write trap handlers for all these sensitive instructions. Unfortunately, not all x86 sensitive instructions are also privileged, so VMMs need to use different techniques to handle them. Instructions that are sensitive but not privileged are considered to be "non-virtualizable."

Handling Non-Virtualizable Instructions

There are a few different techniques for handling these non-virtualizable instructions:

- One option is *binary translation*. All the nonprivileged, sensitive instructions in the guest OS are spotted and rewritten by the VMM in real time. This is complex, and newer x86 processors support hardware-assisted virtualization to simplify binary translation.
- Another option is *paravirtualization*. Instead of modifying the guest OS on the fly, the guest OS is rewritten to avoid the non-virtualizable set of instructions, effectively making system calls to the hypervisor. This is the technique used by the Xen hypervisor.
- Hardware virtualization (such as Intel's VT-x) allows hypervisors to run in a new, extraprivileged level known as *VMX root mode*, which is essentially Ring –1. This allows the VM guest OS kernels to run at Ring 0 (or VMX non-root mode), as they would if they were the host OS.

Nested Virtualization

Running a virtual machine within another virtual machine is called *nested virtualization*. It's not uncommon to see a Type 2 hypervisor running in a machine hosted by a Type 1 hypervisor on bare metal. Performance can be significantly affected, especially when nesting Type 2 hypervisors, and features like device passthrough or GPU acceleration might not be available.

KubeVirt

The KubeVirt project allows running a virtual machine within a container in a Kubernetes pod. When I wrote the first edition of this book, this idea seemed like a science experiment rather than a practical use case, but it has become more common, largely as a step that enterprises are taking to move VM-based workloads into cloud native environments.

KubeVirt uses KVM virtualization, so the guest virtual machine in the container is really running directly on the host kernel. If that host is actually a virtual machine—say, a managed Kubernetes node running in the public cloud—then this is nested virtualization, which may need to be explicitly configured for the instance (and may not be available on all instance types).

If you would like to dig deeper into how virtualization works, Keith Adams and Ole Agesen provide a useful comparison and describe how hardware enhancements enable better performance.

Now that you have a picture of how virtual machines are created and managed, let's consider what this means in terms of isolating one process, or application, from another.

Process Isolation and Security

Making sure that applications are safely isolated from each other is a primary security concern. If my application can read the memory that belongs to your application, I will have access to your data.

Physical isolation is the strongest form of isolation possible. If our applications are running on entirely separate physical machines, there is no way for my code to get access to the memory of your application.

As we have just discussed, the kernel is responsible for managing its user space processes, including assigning memory to each process. It's up to the kernel to make sure that one application can't access the memory assigned to another. If there is a bug in

the way that the kernel manages memory, an attacker might be able to exploit that bug to access memory that they shouldn't be able to reach. And while the kernel is extremely battle-tested, it's also extremely large and complex, and it is still evolving. Even though we don't know of significant flaws in kernel isolation as of this writing, I wouldn't advise you to bet against someone finding problems at some point in the future.

These flaws can come about due to increased sophistication in the underlying hardware. In recent years, CPU manufacturers developed *speculative processing*, in which a processor runs ahead of the currently executing instruction and works out what the results are going to be ahead of actually needing to run that branch of code. This enabled significant performance gains, but it also opened the door to the famous Spectre and Meltdown exploits.

You might be wondering why people consider hypervisors to give greater isolation to virtual machines than a kernel gives to its processes; after all, hypervisors are also managing memory and device access and have a responsibility to keep virtual machines separate. It's absolutely true that a hypervisor flaw could result in a serious problem with isolation between virtual machines. The difference is that hypervisors have a much, much simpler job. In a kernel, user space processes are allowed some visibility of each other; as a very simple example, you can run `ps` and see the running processes on the same machine. You can (given the right permissions) access information about those processes by looking in the `/proc` directory. You are allowed to deliberately share memory between processes through `IPC` and, well, shared memory. All these mechanisms, where one process is legitimately allowed to discover information about another, make the isolation weaker, because of the possibility of a flaw that allows this access in unexpected or unintended circumstances.

There is no similar equivalent when running virtual machines; you can't see one machine's processes from another. There is less code required to manage memory simply because the hypervisor doesn't need to handle circumstances in which machines might share memory—it's just not something that virtual machines do. As a result, hypervisors are far smaller and simpler than full kernels. There are well over 20 million lines of code in the Linux kernel; by contrast, the Xen hypervisor is around 50,000 lines.

Where there is less code and less complexity, there is a smaller attack surface, and the likelihood of an exploitable flaw is less. For this reason, virtual machines are considered to have strong isolation boundaries.

That said, virtual machine exploits are not unheard of. Darshan Tank, Akshai Aggarwal, and Nirbhay Chaubey describe a taxonomy of the different types of attack, and the National Institute of Standards and Technology (NIST) has published security guidelines for hardening virtualized environments.

Disadvantages of Virtual Machines

At this point you might be so convinced of the isolation advantages of virtual machines that you might be wondering why people use containers at all! There are some disadvantages of VMs compared to containers:

- Virtual machines have startup times that are several orders of magnitude greater than a container. After all, a container simply means starting a new Linux process, not having to go through the whole startup and initialization of a VM. The relatively slow startup times of general-purpose VMs means that they are sluggish for autoscaling, not to mention that fast startup times are important when an organization wants to ship new code frequently, perhaps several times per day. However, there are now several lightweight "micro-VMs" with very fast startup times, which I'll discuss in Chapter 10.
- Containers give developers a convenient ability to "build once, run anywhere" quickly and efficiently. It's possible, but very slow, to build an entire machine image for a VM and run it on one's laptop, but this technique hasn't taken off in the developer community in the way containers have.
- In today's cloud environments, when you rent a virtual machine, you have to specify its CPU and memory, and you pay for those resources regardless of how much is actually used by the application code running inside it.

When choosing whether to use VMs or containers, there are many trade-offs to be made among factors such as performance, price, convenience, risk, and the strength of security boundary required between different application workloads.

Container Isolation Compared to VM Isolation

As you saw in Chapter 4, containers are simply Linux processes with a restricted view. They are isolated from each other by the kernel through the mechanisms of namespaces, cgroups, and changing the root. These mechanisms were created specifically to create isolation between processes. However, the simple fact that containers share a kernel means that the basic isolation is weaker compared to that of VMs.

However, all is not lost! You can apply additional security features and sandboxing to strengthen this isolation, which I will explain in Chapter 10. There are also very effective security tools that take advantage of the fact that containers tend to encapsulate microservices, and I will cover these in Chapter 15.

Summary

You should now have a good grasp of what virtual machines are. You have learned why the isolation between virtual machines is considered strong compared to container isolation and why containers are generally not considered suitably secure for hard multitenancy environments. Understanding this difference is an important tool to have in your toolbox when discussing container security.

Securing virtual machines themselves is outside the scope of this book, although I touched on hardening container host configuration in "Container Host Machines" on page 56.

Later in this book, you will see some examples in which the weaker isolation of containers (in comparison to VMs) can easily be broken through misconfiguration. Before we get to that, let's make sure you are up to speed on what's inside a container image and how images can have a significant bearing on security.

CHAPTER 6
Container Images

If you have been using Docker or Kubernetes, you are likely to be familiar with the idea of container images that you store in a registry. In this chapter we're going to explore container images, looking at what they contain and how container runtimes like Docker, `containerd`, `podman`, or `CRI-O` use them.

With an understanding of what images are, you're ready to think about the security implications of building, storing, and retrieving images—and there are a lot of attack vectors related to these steps. You'll learn about best practices for ensuring that builds and images don't compromise your overall system.

Root Filesystem and Image Configuration

There are two parts to a container image: the root filesystem and some configuration.

If you followed along with the examples in Chapter 4, you downloaded a copy of the Alpine root filesystem and used this as the contents of root inside your container. In general, when you start a container, you instantiate it from a container image, and the image includes the root filesystem. If you run `docker run -it alpine sh` and compare it to what's inside your hand-built container, you will see the same layout of directories and files, and they will match completely if the version of Alpine is the same.

If, like many people, you have come to containers through the use of Docker, you'll be used to the idea of building images based on the instructions in a Dockerfile. Some Dockerfile commands (like `FROM`, `ADD`, `COPY`, or `RUN`) modify the contents of the root filesystem that's included in the image. Other commands, like `USER`, `PORT`, or `ENV`, affect the configuration information that's stored in the image alongside the root filesystem. You can see this config information by running `docker inspect` on an image. This config information gives Docker instructions on runtime parameters that

should be set up by default when running the image. For example, if an environment variable is specified using an `ENV` command in the Dockerfile, this environment variable will be defined for the container process when it runs.

Overriding Config at Runtime

In Docker, the config information can be overridden at runtime using command-line parameters. For example, if you want to change an environment variable or set a new one, you can do this with `docker run -e <VARNAME>=<NEWVALUE> ...`. Similarly, to override the user ID configured in the image, you can set the `--user` parameter:

```
$ docker run --rm -it alpine whoami
root
$ docker run --rm -it --user 405 alpine whoami
guest
```

In Kubernetes, you can set container configuration in a pod's YAML definition:

```
apiVersion: v1
kind: Pod
metadata:
  name: demo
spec:
  containers:
  - name: demo-container
    image: demo-reg.io/some-org/demo-image:1.0
    env:
    - name: DEMO_ENV
      value: "This overrides the value"
  securityContext:
    runAsUser: 1000
```

The (imaginary) image `demo-image:1.0` was built from a Dockerfile, which might have included the line `ENV DEMO_ENV="The original value"`. This YAML overrides the value for `DEMO_ENV`, and if the container were to log the value of this variable, you would see `This overrides the value`.

This example YAML also overrides the user ID through the `securityContext.runAsUser` setting for the pod. If there were multiple containers in this pod, this setting would apply to all of them. The Kubernetes documentation describes many other settings that you can configure for the container(s) running within a pod.

If the container runtime in your Kubernetes deployment is an OCI-compliant tool like `containerd/runc`, the values from the YAML definition end up in an OCI-compliant *config.json* file. Let's find out more about these OCI standard container files and tools.

OCI Standards

The Open Container Initiative (OCI) was formed to define standards around container images and runtime, taking its lead from a lot of the work that had originally been done in Docker—in particular, a goal of the OCI was for the standards to support the same user experience that Docker users had come to expect, like the ability to run an image with a default set of configuration settings. The OCI Image Format Spec 1.0 was finalized in 2017, followed by the Distribution Spec 1.0 in 2021, which defined how images are pushed and pulled. This spec is now widely implemented across registries and registry projects, including Harbor, Docker Hub, AWS ECR, and GitHub Container Registry.

The OCI Distribution Spec allows for arbitrary content, not just container images, to be stored in a registry. As a result, registries are being used to hold other types of container-adjacent artifacts, including Helm charts, Open Policy Agent policies, Software Bill of Materials and attestations (which we'll discuss in Chapter 7), and more.

Skopeo is useful for manipulating and inspecting OCI images. It can generate an OCI-format image from a Docker image:

```
$ skopeo copy docker://alpine:latest oci:alpine:latest
$ ls alpine
blobs  index.json  oci-layout
```

You'll see similar output generated with Docker tooling later in this chapter.

The `index.json` file contains the *manifest* for the image, including the unique digest that identifies it:

```
cat alpine/index.json | jq
{
  "schemaVersion": 2,
  "manifests": [
    {
      "mediaType": "application/vnd.oci.image.manifest.v1+json",
      "digest": "sha256:08001109a7d679fe33b04fa51d681bd40b975d8f5cea8c3e...",
      "size": 1022,
      "annotations": {
        "org.opencontainers.image.ref.name": "latest"
      }
    }
```

```
    ]
}
```

This digest matches one of the *blobs* in the OCI-format image:

```
$ ls alpine/blobs/sha256/
08001109a7d679fe33b04fa51d681bd40b975d8f5cea8c3ef6c0eccb6a7338ce
cea2ff433c610f5363017404ce989632e12b953114fefc6f597a58e813c15d61
fe07684b16b82247c3539ed86a65ff37a76138ec25d380bd80c869a1a4c73236
```

These blobs represent the layers of the image that we'll come to shortly in this chapter. But an OCI-compliant runtime like `runc` doesn't work directly with the image in this format. Instead, it first has to be unpacked into a runtime filesystem bundle. Let's look at an example, using `umoci` to unpack the image:

```
$ sudo umoci unpack --image alpine:latest alpine-bundle
$ ls alpine-bundle
config.json
rootfs
sha256_08001109a7d679fe33b04fa51d681bd40b975d8f5cea8c3ef6c0eccb6a7338ce.mtree
umoci.json
$ ls alpine-bundle/rootfs
bin  etc   lib    mnt  proc  run   srv  tmp  var
dev  home  media  opt  root  sbin  sys  usr
```

As you can see, this bundle includes a `rootfs` directory with the contents of an Alpine Linux distribution. There is also a *config.json* file that defines the runtime settings. The runtime instantiates a container using this root filesystem and settings.

I ran the `umoci` command under `sudo`, but since it has rootless support, you could choose to run `umoci` as an unprivileged user by specifying the `--rootless` option.

When you're using Docker, you don't get direct access to the config information in the form of a file you can inspect with `cat` or your favorite text editor, but you can see that it's there by using the `docker image inspect` command.

Image Configuration

Now that you know from Chapters 3 and 4 how containers are created, it's worth taking a look at one of these *config.json* files, because a lot of it should look familiar. Here's are some extracts from the configuration file for the Alpine image I just unpacked, as an example:

```
"root": {
                "path": "rootfs"
        },
```

```
        "hostname": "umoci-default",
        "mounts": [
                {
                        "destination": "/proc",
                        "type": "proc",
                        "source": "proc"
                },
...
                {
                        "destination": "/sys/fs/cgroup",
                        "type": "cgroup",
                        "source": "cgroup",
                        "options": [
                                "nosuid",
                                "noexec",
                                "nodev",
                                "relatime",
                                "ro"
                        ]
                }
        ],
...

        "namespaces": [
            {
                "type": "pid"
            },
            {
                "type": "network"
            },
            {
                "type": "ipc"
            },
            {
                "type": "uts"
            },
            {
                "type": "mount"
            }
        ]
    }
```

The configuration information includes a definition of everything `runc` should do to create the container. As you can see from these extracts, this includes a list of file mounts such as those needed for `/proc` and for cgroups, and the namespaces the runtime should create for this container.

You have seen how an image consists of two parts: the root filesystem and some configuration information. Now let's consider how an image gets built.

Building Images

Most people's experience of building container images is to use the `docker build` command. This command follows the instructions from a file called a *Dockerfile* to create an image.

Historically, running `docker build` was a relatively simple and monolithic operation that created a container image suitable for the architecture of the machine where the build is run. More recently, Docker introduced BuildKit from the Moby project. (As you may know, Docker renamed its open source Docker engine code to "Moby" in an attempt to avoid the inevitable confusion when the project and company names were the same.) BuildKit offers more advanced capabilities, including a rootless mode, the ability to produce an image for multiple platforms, the ability to push the image to multiple registries, and more, and is the default builder from Docker v23 onward.

The Dangers of Docker Build

When you run a `docker` command, in the classic Docker architecture, the command-line tool you invoked (`docker`) does very little by itself. Instead, it converts your command into an API request that it sends to the Docker daemon via a socket referred to as the *Docker socket*. Any process that has access to the Docker socket can send API requests to the daemon.

The Docker daemon is a long-running process that actually does the work of running and managing both containers and container images. As you saw in Chapter 4, to create a container, the daemon needs to be able to create namespaces, so it needs to be running as root. You can easily check this using `ps`:

```
$ ps -fC dockerd
UID          PID    PPID  C STIME TTY          TIME CMD
root       22240       1  0 Jun04 ?        00:01:58 /usr/bin/dockerd ...
```

A user can get the daemon to run arbitrary commands by specifying them in a `RUN` command in a Dockerfile. Since the daemon is running as root, any user who can run a build may as well have root privileges on that machine.

Imagine that you want to dedicate a machine (or virtual machine) to build container images and store them in a registry. Using the standard Docker approach, your machine has to run the daemon, which has a lot more capabilities beyond building and interacting with registries. Without additional precautions, any user who can trigger a `docker build` on this machine can also perform a `docker run` to execute any command they like on the machine.

Not only can they run any command they like, but also, if they use this privilege to perform a malicious action, it will be hard to track down who was responsible. You may keep an audit log of actions that users take, but—as illustrated nicely in a post by Dan Walsh—the audit will record the daemon process's ID rather than that of the user.

To avoid these security risks, there are several alternative approaches for building container images without relying on the Docker daemon:

- The BuildKit `docker-container` driver allows running builds within a container, isolated from the host it's running on.
- Docker now supports a rootless mode, where the daemon `buildkitd` runs as a non-root user. This mode is generally available, but at the time of writing, it requires you to opt in.
- Other nonprivileged builds include Red Hat's `podman` and `buildah`. A blog post from Puja Abbassi describes these tools and compares them to (pre-BuildKit) `docker build`.
- Google's Bazel can build many other types of artifact, not just container images. It prides itself on generating images deterministically so that you can reproduce the same image from the same source.
- Another tool with a focus on reproducible builds is Nix, albeit one that is renowned for having a steep learning curve.
- If you're building simple Go applications into containers, you might want to try ko, and for Java applications, there is `jib`.

GitLab has some guidance and examples for running rootless builds.

Whichever tool you use, you might run it manually from the command line, but for production builds, you will likely automate them as part of a continuous integration/continuous deployment (CI/CD) pipeline.

Image Layers

Regardless of which tool you use, the vast majority of container image builds are defined through a Dockerfile. The Dockerfile gives a series of instructions, each of which results in either a filesystem layer or a change to the image configuration. This is described well in the Docker documentation, but if you want to dig into the details, you might enjoy the blog post I wrote about re-creating the Dockerfile from an image.

Sensitive data in layers

Anyone who has access to a container image can access any file included in that image. From a security perspective, you want to avoid including sensitive information such as passwords or tokens in an image. (I'll cover how you should handle this information in Chapter 14.)

The fact that every layer is stored separately means that you have to be careful not to store sensitive data, even if a subsequent layer removes it. Here's a Dockerfile that illustrates what *not* to do:

```
FROM alpine
RUN echo "top-secret" > /password.txt
RUN rm /password.txt
```

One layer creates a file, and then the next layer removes it. If you build this image and then run it, you won't find any sign of the `password.txt` file:

```
$ docker run --rm -it sensitive ls /password.txt
ls: /password.txt: No such file or directory
```

But don't let this fool you—the sensitive data is still included in the image. You can prove this by exporting the image to a tar file using the `docker save` command and then unpacking the tar:

```
$ docker save sensitive > sensitive.tar
$ mkdir sensitive
$ cd sensitive
$ tar -xf ../sensitive.tar
$ ls
blobs  index.json  manifest.json  oci-layout  repositories
```

I'm using Docker version v28.3.2, which outputs both standard OCI format metadata and (for compatibility with older tooling) metadata in legacy Docker format. In the future, or with other tools, you will likely only see the OCI format information. We can use the OCI-format information and simply ignore the legacy data from the `index.json` and `repositories` files.

This should look familiar from earlier in the chapter:

`oci_layout`
: Specifies the OCI specification version for this image.

`manifest.json`
: The top-level file describing the image; it tells you which file represents the configuration for the image, describes any tags for this image, and lists each of the layers.

`blobs`
: A directory holding the data files that make up the image.

Inside the `manifest.json` file, you'll find the location of the configuration information, a list of tags for the image, and a list of layer files that make up the root file system for the image:

```
...
      "Config": "blobs/sha256/258d7...7f1a8",
        "RepoTags": [
            "sensitive:latest"
        ],
       "Layers": [
           "blobs/sha256/1231a...a69f1",
           "blobs/sha256/9e745...67031",
           "blobs/sha256/82316...4bd6e"
       ],
...
```

The config file includes the history of the commands that were run to construct this container. As you can see, in this case, the sensitive data `"top-secret"` is revealed in the step that runs the `echo` command:

```
$ cat blobs/sha256/258d* | jq '.history'
[
  {
    "created": "2025-05-30T16:20:41Z",
    "created_by": "ADD alpine-minirootfs-3.22.0-aarch64.tar.gz / # buildkit",
    "comment": "buildkit.dockerfile.v0"
  },
  {
    "created": "2025-05-30T16:20:41Z",
    "created_by": "CMD [\"/bin/sh\"]",
    "comment": "buildkit.dockerfile.v0",
    "empty_layer": true
  },
  {
    "created": "2025-07-26T05:02:59.245427668-05:00",
    "created_by": "RUN /bin/sh -c echo \"top-secret\" > /password.txt # buildkit",
    "comment": "buildkit.dockerfile.v0"
  },
```

```
  {
    "created": "2025-07-26T05:02:59.348379231-05:00",
    "created_by": "RUN /bin/sh -c rm /password.txt # buildkit",
    "comment": "buildkit.dockerfile.v0"
  }
]
```

Inside each layer's directory, there is another tar file holding the contents of the filesystem at that layer. It's easy to reveal the `password.txt` file from the appropriate layer:

```
$ cd blobs/sha256
$ tar -xvf 9e7*
etc/
password.txt
$ cat password.txt
top-secret
```

As this shows, even if a subsequent layer deletes a file, any file that ever existed in any layer can easily be obtained by unpacking the image. Don't include anything in any layer that you're not prepared to show anyone who has access to the image.

In Chapter 8 we will discuss scanning container images for vulnerabilities, and some of these scanning tools can also check for secrets inadvertently included in image layers. If you do detect one of your secrets in an image, you should not only correct the build so that future image versions don't include it but also rotate the secret so that it's of no use to anyone who has access to older image versions.

Sometimes it is necessary to use a secret during the build process, so let's see how this can be done safely, without leaving the secret in the image.

BuildKit secret mounts

BuildKit provides an alternative approach that allows passing sensitive data at build time that never gets written into the image layers. You can specify an identifier for the secret along with a file containing the sensitive information, like this:

```
$ docker build --secret id=<id>,src=secret.txt .
```

The secret can be made available to the build, as a file mounted at the location `/run/secret/<id>`. The Dockerfile refers to this secret where it's needed, like this:

```
RUN --mount=type=secret,id=<id> <command that needs the secret>
```

The secret is temporarily mounted for the individual build step where it's needed and only ever lives in memory.

As an example, suppose you need a bearer token to access an API that you need to call during the build. Your Dockerfile would contain a command, like this:

```
RUN --mount=type=secret,id=API_TOKEN \
    sh -c 'curl -H "Authorization: Bearer $(cat /run/secrets/API_TOKEN)"
    https://api.your-domain.com/data'
```

You could then store the token temporarily in a file and access it during the build like this:

```
$ echo <secret token> > token.txt
$ docker build --secret id=API,TOKEN,src=token.txt -t myimage:v1.0.0 .
```

You wouldn't want to leave that `token.txt` file lying around on the build machine containing the secret token, of course, so you would delete it, safe in the knowledge that it hasn't been stored in any of the layers of `myimage:v1.0.0`.

This sounds great, but unfortunately, it's all too easy to lose the benefit of BuildKit's secret mount by *running a command that persists the secret value into the image*! Avoid patterns like this, which write the secret to a file:

```
RUN --mount=type=secret,id=my-secret cat /run/secret/my-secret > out.txt
```

This command creates a file `out.txt` containing the sensitive information. And that file is going to exist in the image that gets built.

There is more information about mounting secrets, including the option to pass secret values into the build from environment variables, or passing SSH mounts, in the Docker documentation.

Secret mounts are just one of the capabilities that BuildKit adds over the original Docker build approach. Another very useful feature is the ability to create images for multiple chip architectures.

Multiplatform Images

Container images can be built to support multiple CPU architectures, including the common options `amd64` typically used for Intel-based chips, and `arm64` for ARM-based machines. A multiplatform image contains a manifest list that references architecture-specific machines. You can use Docker to inspect a multiplatform image, like this:

```
$ docker manifest inspect alpine
{
   "schemaVersion": 2,
   "mediaType": "application/vnd.oci.image.index.v1+json",
   "manifests": [
      {
         "mediaType": "application/vnd.oci.image.manifest.v1+json",
         "size": 1022,
         "digest": "sha256:08001109a7d679fe33b04fa51d681bd40b975d8f5ce...",
```

```
            "platform": {
               "architecture": "amd64",
               "os": "linux"
            }
         },
...
         {
           "mediaType": "application/vnd.oci.image.manifest.v1+json",
            "size": 1025,
            "digest": "sha256:008448246686fe28544e36ba89e7bc7fbe6dad8a2cc...",
            "platform": {
               "architecture": "arm64",
               "os": "linux",
               "variant": "v8"
            }
         },
...
```

The manifest includes a digest identifying the image for each supported platform. When pulling an image, the container runtime retrieves the image that matches the platform it's running on.

The contents of these per-platform images are independent of each other. If your organization is running images on a mix of different platforms, you'll need to make sure that the images are scanned for vulnerabilities and insecure configurations for all the different platforms, because there's no guarantee that the content will be the same. We'll discuss image scanning in Chapter 8.

Earlier in this chapter you saw what's inside an OCI-compliant container image, and you now know what is happening when these images are built from a Dockerfile. Now let's consider how images are stored.

Storing Images

Container images are stored in container registries. If you use Docker, you've probably used the Docker Hub registry, and if you're working with containers using the services of a cloud provider, it's likely you're familiar with one of the cloud provider's registries—AWS Elastic Container Registry, for example, or Google Container Registry. There are also commercially available registries and open source solutions such as Harbor.

Running Your Own Registry

Many organizations maintain their own registries or use managed registries from their cloud provider and require that only images from those permitted registries can be used. Running your own registry (or your own instance of a managed registry) gives you more control and visibility over who can push and pull images. It also reduces the possibility of a DNS attack that allows an attacker to spoof the registry

address. If the registry lives within a VPC, it is highly unlikely that an attacker can do this. Most managed registries (such as AWS ECR, Microsoft ACR, and Google GCR) support private VPC endpoints.

Care should be taken to restrict direct access to the registry's storage media. For example, a registry running in AWS might use S3 to store images, and the S3 bucket(s) should have restrictive permissions so that a bad actor can't directly access stored image data.

Pushing and Pulling

Storing an image in a registry is generally referred to as a *push*, and retrieving it is a *pull*.

Each layer is stored separately as a *blob* of data in the registry, identified by a hash of its contents. To save storage space, a given blob needs to be stored only once, although it may be referenced by many images. The registry also stores an image *manifest* that identifies the set of image layer blobs that make up the image. Taking a hash of the image manifest gives a unique identifier for the entire image, which is referred to as the image *digest*. If you rebuild the image and anything about it changes, this hash will also change.

If you're using Docker, you can easily see the digests for images held locally on your machine by using the following command:

```
$ docker image ls --digests
REPOSITORY   TAG      DIGEST              IMAGE ID       CREATED      SIZE
nginx        latest   sha256:50cf...8566  231d40e811cd   2 weeks ago  126MB
```

When you push or pull an image, you can use this digest to precisely reference this particular build, but this isn't the only way you can refer to an image. Let's review the different ways of identifying container images.

Identifying Images

The first part of an image reference is the URL of the registry where it is stored. (If the registry address is omitted, this implies either a locally stored image or an image stored on Docker Hub, depending on the command context.)

The next part of an image reference is the name of the user or organization account that owns this image. This is followed by an image name and then either the digest that identifies its contents or a human-readable tag.

Putting this together gives us an address that looks like one of these options:

```
<Registry URL>/<Organization or user name>/<repository>@sha256:<digest>
<Registry URL>/<Organization or user name>/<repository>:<tag>
```

If the registry URL is omitted, it defaults to Docker Hub's address, `docker.io`. Figure 6-1 shows an example version of an image as it appears on Docker Hub.

You could pull this image with either of the following commands (edited for brevity):

```
$ docker pull aquasec/trivy:0.66.0
$ docker pull aquasec/trivy@sha256:08697...dfb1e
```

Referring to an image by digest is unwieldy for humans to deal with, resulting in the commonplace use of *tags*, which are just arbitrary labels applied to the image. A single image can be given any number of tags, and the same tag can be moved from one image to another. Tags are often used to indicate the version of software contained in the image—as in the example just shown, which is version 0.66.0.

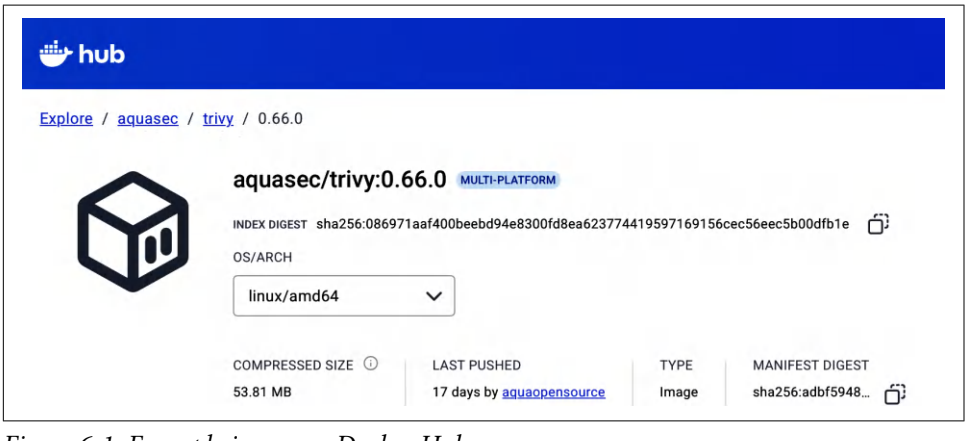


Figure 6-1. Example image on Docker Hub

Because tags can be moved from image to image, there is no guarantee that specifying an image by tag today will give you the same result as it does tomorrow. In contrast, using the digest will give you the identical image, because it is a hash derived from the contents of the image. Any change to the image results in a different hash.

This effect may be exactly what you intend. For example, you might refer to an image using a tag that refers to the major and minor version numbers in a semantic versioning schema. If a new patched version is released, you rely on the image maintainers to retag the patched image with the same major and minor version numbers so that you get the up-to-date patched version when you next pull the image.

However, there are occasions when the unique reference to an image is important. For example, consider the scanning of images for vulnerabilities (which is covered in Chapter 8). You might have an admission controller that checks that images can be deployed only if they have been through the vulnerability scanning step, and this will need to check records of the images that have been scanned. If these records refer to the images by tag, the information is unreliable, as there's no way of knowing whether

the image has changed and needs to be rescanned. Referencing images by digest is common practice in automated deployments and is generally considered a good idea in production.

Some registries allow you to set tags to be immutable, which prevents pushing a new hash to an existing tag in the registry. This prevents "tag confusion" but also means that you can't reuse the tag for a major version number when using semantic versioning.

Summary

In this chapter, you have seen how images are built and stored. You have learned enough about the format of images to understand how they might inadvertently contain sensitive data, and you have seen how to avoid including sensitive data in images by using secure methods to pass in secrets required during the build process. You have also seen options for referring to images by tag and by digest.

When you pull an image from a registry, you want confidence that it contains the code you expect and doesn't hold anything malicious. Coming up in Chapter 7, we will discuss concerns related to the security and provenance of an image and its contents, commonly known as *supply chain security*.

CHAPTER 7

Supply Chain Security

Anyone deploying container images needs to have confidence that the software within each image is safe to run and has not been tampered with at any point. In this chapter, let's look at how you can securely build and distribute container images along with information that reassures the user about its integrity and provenance.

Supply chain security is concerned with making sure that the software you deploy and run is what you expect it to be, having been built from source code that you trust, by a build system that you trust. There are various potential weak points in the chain, from building and storing an image to running the image, as shown in Figure 7-1.

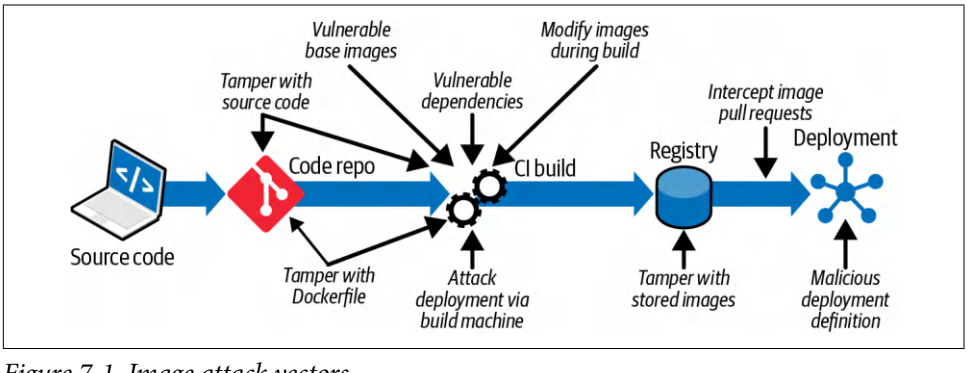


Figure 7-1. Image attack vectors

Container Image Software Components

Container images can contain lots of different software components:

- As you saw in Chapter 4, a container image includes a filesystem, often based on a Linux distribution, containing all the files and directories included in that distribution.
- Distributions typically support a package manager like `apt` or `yum`, and the container image might have some of these packages installed into it—ideally (but not necessarily) only the dependencies that are needed by the application.
- Depending on the language used, the application itself might be a compiled binary, or it might be a series of interpreted scripts.
- There might well be some language-specific libraries needed—for example, Rust crates, Ruby gems, or Node or Python packages.
- There might be other files—for example, for configuration or data—compiled into the image.

Any of these software components might contain vulnerable code that an attacker can exploit.

Distributions, packages, libraries, and application source code evolve over time. Newer versions of any software might contain fixes for vulnerabilities; they could also introduce new, unknown vulnerabilities.

These different components are, generally speaking, written by different developers and are obtained from different sources. For example, the Linux distribution might come from an organization like Alpine or a company like Red Hat. If you pull a base image representing, say, a distribution of Red Hat Enterprise Linux, you want to be sure that it really came from Red Hat and not from a malicious imposter. Similarly, you want confidence that the packages and libraries included in an image come from legitimate providers.

You also need appropriate access controls on your source code repositories to ensure that unauthorized users can't tamper with it or modify what gets built into container images.

Your company or organization might run container images that it builds itself, perhaps for your own business-specific applications. It probably also uses container images built by a vendor or other third party, for common infrastructure components or tools. You might be responsible for building container images that are distributed and used by other organizations and want to give those consumers confidence that your images are safe to use.

Knowing (or working out) what components are included, and what versions of each component, is essential for flagging any known vulnerabilities in a container image. We'll consider how this works in more detail in Chapter 8, but for this chapter, we're concerned with ensuring that every component in the container's creation and delivery path can be verified and traced.

SLSA

Supply Chain Levels for Software Artifacts (SLSA) is a framework from the Open Source Security Foundation (OpenSSF) that defines desirable characteristics for securing a software supply chain. SLSA defines levels from 1 to 3, which represent increasing confidence in the integrity, traceability, and provenance of software artifacts—including container images. The SLSA recommendations include:

- Reproducible builds
- Isolated build environments
- An auditable trail defining how software was built and from what dependencies

The information about what components went into a software artifact is known as a *Software Bill of Materials*, commonly referred to as an SBOM. This tells us what components went into a build; SLSA additionally defines *provenance* information, which describes how the artifact was built and by whom.

To quote from the SLSA website, "a build configuration file (i.e., GitHub workflow) qualifies for SLSA 1. It would be considered unsigned, unformatted provenance." In contrast, the website states that "To achieve SLSA 3, you must:

- Run your build on a hosted platform that generates and signs provenance
- Ensure that build runs cannot influence each other
- Produce signed provenance that can be verified as authentic"

The SBOM and provenance information are created as part of the process that builds the container image. Consumers of the image will want to verify that information when they deploy it. This verification step is also specified in the SLSA framework. Let's consider what risks are avoided by having and verifying an SBOM.

Software Bill of Materials

An SBOM provides a machine-readable inventory of all these different components that go into a container image, including information about the versions used. As you'll see in Chapter 8, the SBOM allows automating the process of identifying which images need updating when a new vulnerability is discovered. The SBOM also holds

license information about the software components, which can be helpful to meet compliance requirements.

The SBOM can play a critical role in combating vulnerabilities related to *dependency confusion* and *package hallucination*.

Dependency Confusion

Dependency confusion can arise by using an unexpected version of a dependency, because it wasn't specified correctly, or by pulling it from the wrong location (the wrong registry, package manager, or cache location). This can be avoided by explicitly specifying the version and location of the dependency.

Package Hallucination

Dependency confusion has become a much bigger problem with the advent of AI-generated code. A 2025 study[1] showed that code created by large language models (LLMs) has a tendency to hallucinate the names of imported packages, often following predictable naming patterns. It's clearly a problem if generated code doesn't work because it tries to import a package that doesn't exist. It's arguably a bigger problem if a malicious actor populates those missing packages that are commonly hallucinated so that the generated code seems to be working but has incorporated exploit-ridden dependencies.

In container builds, we have to cope with language-specific dependencies referred to by source code that developers write, and container image dependencies specified in Dockerfiles.

Language-Specific SBOMs

Source code is often quite nonspecific about the exact versions of dependencies that it incorporates (for example, `import antigravity` in Python[2] doesn't mention a version number). Approaches like lock files, Go's *go.sum* files, and Python's *requirements.txt* are all language-specific approaches to using the right versions, but they are optional. As you saw in Chapter 6, container image tags only very loosely indicate a version. During the build process, all these loose specifications are resolved to some specific versions that get used in the construction of the image.

1 Joseph Spracklen et al., "We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs," arXiv, March 2, 2025, *https://arxiv.org/abs/2406.10279*.

2 See "Python," XKCD, accessed August 8, 2025, *https://xkcd.com/353*.

The ideal SBOM records precisely what versions are resolved during the build process. This can be done using reproducible build tools like Bazel or the OWASP CycloneDX ecosystem, which has language-specific "plug-ins." For example, in a Java project build, you might run the following command during the build:

```
mvn org.cyclonedx:cyclonedx-maven-plugin:makeAggregateBom
```

Or in Go:

```
cyclonedx-gomod mod -licenses -json -output sbom.json
```

These generate SBOMs including all the resolved dependencies, based on the project's *pom.xml file* or *go.mod* file, respectively. CycloneDX plug-ins exist for many other languages too.

If the SBOM isn't generated at build time, it's possible to generate one using tools that inspect the image and reverse-engineer its contents, but this will likely be less complete and accurate, particularly when it comes to language-specific packages. Creating an SBOM is essentially the same problem that vulnerability scanners such as `trivy` and `syft` tackle—in fact, both these scanning tools can generate SBOMs as well as vulnerability reports. The best practice is to pair language-specific SBOMs with container-level SBOMs for a complete picture. We'll discuss vulnerabilities and image scanning in more detail in Chapter 8, and you'll read about creating a container-level SBOM later in this chapter.

The OpenSSF has a Working Group on securing software repositories, focusing on recommendations and best practices that can be shared among different package manager communities, aiming to better enable signing and provenance information for open source software.

Minimal Base Images

The first line of a Dockerfile has a `FROM` command that specifies the base image on which the rest of the image will be built. As you saw in Chapter 6, the image includes a filesystem, and the base image includes that starting point for that filesystem, which will appear in the first layer of the image.

The smaller the base image, the less likely that it includes unnecessary code, and the smaller the attack surface. Smaller images also have the benefit of being quicker to send over the network. There are a few different approaches for minimizing the base image:

- If you can, consider building from scratch—a completely empty image suitable for standalone binaries. You can use multistage builds to help you achieve this, by using a "full" base image for the initial stage(s) that perform the compilation, and then copying the resulting binary or binaries into a scratch base image in the final stage. (Multistage builds are further discussed later in this chapter.)
- Use a minimal base image such as Google's distroless or hardened images from suppliers such as Chainguard or Docker. Amazon provides a minimal container image for its Amazon Linux distribution, Microsoft supplies minimal Azure Linux images, and there is a project from Canonical called chisel for minimizing the set of packages installed into an Ubuntu image.
- The Slim Toolkit can analyze an image to identify redundant components that can be eliminated.

Dependencies and base images aren't the only route for introducing vulnerabilities. Application developers can affect security through the code they write themselves. Static and dynamic security analysis tools, peer review, security assessments, and penetration testing can all help to identify insecurities added during development. This all applies for containerized applications just as it does without containers, and SLSA recommendations can apply to any software, not just containerized code. But since this book is concerned with containers, let's focus on that, starting with the Dockerfile.

Dockerfile Security

The build step takes a Dockerfile and converts it into a container image. Within that step, there are a number of potential security risks.

Provenance of the Dockerfile

The instructions for building an image come from a Dockerfile. Each stage of the build involves running one of these instructions, and if a bad actor is able to modify the Dockerfile, it's possible for them to take malicious actions, including:

- Adding malware or cryptomining software into the image
- Accessing build secrets
- Enumerating the network topology accessible from the build infrastructure
- Attacking the build host

It may seem obvious, but the Dockerfile (like any source code) needs appropriate access controls to protect against attackers adding malicious steps into the build.

The contents of the Dockerfile also have a huge bearing on the security of the image that the build produces. Let's turn to some practical steps you can take in the Dockerfile to improve image security.

Dockerfile Best Practices for Security

These recommendations all improve the security of the image and reduce the chances that an attacker can compromise containers running from this image:

Base image
: The first line of the Dockerfile is a `FROM` instruction indicating a base image that the new image is built from. Here are some considerations to take into account when choosing the base image:

  - Refer to an image from a trusted registry (see "Storing Images" on page 80).
  - Minimize the contents of the base image, as discussed in "Minimal Base Images" on page 89.
  - Arbitrary third-party base images might include malicious code, so some organizations mandate the use of preapproved or "golden" base images.

  As you saw in Chapter 6, you can reference image versions using tags or digests, and you should be thoughtful about using a tag or a digest to reference the base image. The build will be more reproducible if you use a digest, but it might mean you are less likely to pick up new versions of a base image that might include security updates. (That said, you should pick up missing updates through a vulnerability scan of your complete image.) Avoid dependency confusion by explicitly identifying the registry from which the base image should be obtained.

Use multistage builds
: The multistage build is a way of eliminating unnecessary contents in the final image. An initial stage can include all the packages and toolchain required to build an image, but a lot of these tools are not needed at runtime. As an example, if you write an executable in Go, it needs the Go compiler to create an executable program. The container that runs the program doesn't need to have access to the Go compiler. In this example, it would be a good idea to break the build into a multistage build: one stage does the compilation and creates a binary executable —this stage needs to run in an image that includes the compiler. The next stage copies the standalone executable into a smaller base image—perhaps even the Scratch image, which is an empty file system. The image that gets deployed has a much smaller attack surface, with a lower likelihood of vulnerabilities; a nonsecurity benefit is that the image itself will also be smaller, so the time to pull the image is reduced.

Capital One has several multistage build examples for node applications on its blog, showing how you can even run tests as different steps within a multistage build without impacting the contents of the final image.

Non-root `USER`
: The `USER` instruction in a Dockerfile specifies that the default user identity for running containers based on this image isn't root. In Chapter 11 I'll cover some very good reasons why you should avoid running as root and should specify a non-root user in all your Dockerfiles wherever possible.

`RUN` *commands*
: Let's be absolutely clear: a Dockerfile `RUN` command lets you run any arbitrary command. If an attacker can compromise the Dockerfile with the default security settings, that attacker can run *any code of their choosing*. If you have any reason not to trust people who can run arbitrary container builds on your system, I can't think of a better way of saying this: you have given them privileges for remote code execution. Make sure that privileges to edit Dockerfiles are limited to trusted members of your team, and pay close attention to code reviewing these changes. You might even want to institute a check or an audit log when any new or modified `RUN` commands are introduced in your Dockerfiles.

Volume mounts
: We often mount host directories into a container through volume mounts. As you will see in Chapter 11, it's important to check that Dockerfiles don't mount sensitive directories like `/etc` or `/bin` into a container.

Don't include sensitive data in the Dockerfile
: In Chapter 6, you saw some mechanisms for safely passing secrets during the build process, and we'll discuss sensitive data and secrets for runtime in more detail in Chapter 14. Please understand that including credentials, passwords, or other secret data in an image makes it easier for those secrets to be exposed.

Avoid setuid binaries
: As discussed in Chapter 2, it's a good idea to avoid including executable files with the `setuid` bit, as these could potentially lead to privilege escalation.

Avoid unnecessary code
: The smaller the amount of code in a container, the smaller the attack surface. Avoid adding packages, libraries, and executables into an image unless they are absolutely necessary. For the same reason, if you can base your image on the scratch image or one of the distroless options, you're likely to have dramatically less code—and hence less vulnerable code—in your image.

Include everything that your container needs
: If the previous point exhorted you to exclude superfluous code from a build, this point is a corollary: *do* include everything that your application needs to operate. If you allow for the possibility of a container installing additional packages at runtime, how will you check that those packages are legitimate? It's far better to do all the installation and validation when the container image is built and create an immutable image. See "Immutable Containers" on page 111 for more on why this is a good idea.

Avoid dependency confusion
: Here are some routes to consider for avoiding dependency confusion in your Dockerfiles:

  - As mentioned already, the base image is specified in the Dockerfile's `FROM` command. Identify the version you want to use, preferably by hash, but at least by version tag rather than relying on `latest`. Specify the registry explicitly, too, to make sure that the system doesn't fall back to an unexpected default location.
  - Only use base images from sources that you trust. Many organizations insist that they should be pulled from a private registry to ensure everyone is using an approved version. The images might originate from a public registry and are scanned and stored in the private registry if the organization's security team considers it safe to use. Another option is to build base images from source.
  - Make sure package managers are pulling from the correct registries—for example, using `--index-url` on `RUN pip install` commands—or using `set @my-org:registry` on `RUN npm config`.
  - Consider pinning package dependency versions precisely, specifying versions in commands like `RUN apt-get install`.
  - Avoid letting the build automatically or implicitly upgrade dependencies. For example, use `--require-hashes on RUN pip install`.
  - Ideally, all the packages you need should be installed explicitly—for example, using the `--no-install-recommends` flag on `RUN apt-get install`.

Specifying all dependency versions explicitly ensures you're using a precisely defined set of dependencies, avoiding dependency confusion, but there are some trade-offs. Unless you keep the versions updated, you might be missing important security updates that would be picked up automatically if you were to specify the versions more loosely. On the other hand, if you specify versions too loosely, you could easily find "breaking changes" in new versions of packages that are no longer compatible with your code. Vulnerability scanning can be used to spot when your image needs an important security update, and testing should spot when you encounter a breaking

change, but neither of these is 100% perfect. Finding the right balance between automatic updates and explicitly defined dependencies is a careful balance.

Following these recommendations will help you build images that are harder to exploit. Now let's turn to the risk that an attacker will attempt to find weaknesses in your container build system.

Attacks on the Build Machine

The machine that builds the image is a concern for two main reasons:

- If an attacker can breach the build machine and run code on it, can they reach other parts of your system? As you saw in "The Dangers of Docker Build" on page 74, there are reasons to explore using a build tool that doesn't require a privileged daemon process. Use rootless/unprivileged builders if possible.
- Can an attacker influence the outcome of a build so that you end up building, and ultimately running, malicious images? Any unauthorized access that interferes with a Dockerfile's instructions or that triggers unexpected builds can have disastrous consequences. For example, if an attacker can influence the code that's built, they could insert a backdoor into containers that run on your production deployment.

Given that your build machines create the code that you will ultimately run in your production cluster, it's critical to harden them against attack as if they were as important as the production cluster itself. It is a good idea to run builds on a separate machine or cluster of machines from the production environment to limit the blast radius of a build attack. It would be even better, if possible, to use ephemeral infrastructure, spinning up fresh virtual machines for each build: it's harder for an attacker to establish a foothold in the build process if there is no opportunity for (potentially malicious) files to "hang around" from one build to the next.

Many build services and CI/CD platforms like GitHub Actions, GitLab CI/CD, and Bitbucket Pipelines support the concept of *runners*—temporary virtual machines that are brought up specifically to run a build job. GitHub offers runners hosted on Microsoft Azure, or you can provide your own infrastructure in a public or private cloud. The job might run directly on the runner or within a container on that runner.

Philipp Garbe wrote a post about different options for configuring GitHub Actions with self-hosted runners on AWS CodeBuild.

Unprivileged builders protect against the likelihood of being able to run code on the host, and for defense in depth, you should limit network and cloud service access from build machines to prevent an attacker from accessing other elements of your deployment.

Reduce the attack surface by eliminating unnecessary tools from build machines. Restrict direct user access to these machines, and protect them from unauthorized network access using VPCs and firewalls.

Generating an SBOM

As discussed previously, it's good practice to create an SBOM to enumerate the contents of an image. Ideally you'll have a language-specific SBOM for your application code, but you'll also want to record information about the base image and installed OS packages.

Ideally, the SBOM should be generated at build time, for example, with `docker build --sbom=true`. You can also generate an SBOM for an existing image, and there are several tools commonly used for this, including `syft` and `trivy`. SBOM information is often generated in common formats SPDX or CycloneDX. To generate an SBOM in SPDX format for the latest `nginx` image, I can run this command:

```
$ trivy image --format spdx-json nginx
```

The output contains information about the packages in the image, licensing information, relationships between packages, and data about the tool that generated the report. When I ran this tool, the output was more than 8,000 lines long, so I won't include it all here, but just to give you a flavor, here's an extract (with some lines removed or shortened for brevity) describing the `bash` package that Trivy identified within the `nginx` container:

```
"name": "bash",
"SPDXID": "SPDXRef-Package-a592647b46e04269",
"versionInfo": "5.2.15-2+b8",
"supplier": "Organization: Matthias Klose \u003cdoko@debian.org\u003e",
"downloadLocation": "NONE",
"filesAnalyzed": false,
"sourceInfo": "built package from: bash 5.2.15-2",
"licenseConcluded": "GPL-3.0-or-later AND GPL-3.0-only AND LicenseRef-4f0e9e1...",
"licenseDeclared": "GPL-3.0-or-later AND GPL-3.0-only AND LicenseRef-4f0e9e1...",
"externalRefs": [
  {
    "referenceCategory": "PACKAGE-MANAGER",
    "referenceType": "purl",
    "referenceLocator": "pkg:deb/debian/bash@5.2.15-2%2Bb8?arch=amd64\u0026distro=
    debian-12.11"
  }
],
```

```
    "primaryPackagePurpose": "LIBRARY",
    "annotations": [
      {
        "annotator": "Tool: trivy-0.63.0",
        "annotationDate": "2025-06-23T10:06:37Z",
        "annotationType": "OTHER",
        "comment": "LayerDiffID: sha256:7fb72a7d1a8e984..."
      },
```

The SBOM can be used as input into a vulnerability scanner for cross-referencing with known vulnerabilities (which we'll come to in Chapter 8), and your SBOMs can be indexed so that you can easily identify components affected by newly disclosed vulnerabilities. SBOMs can also be used to check for license compliance. For example, if you are building commercial, proprietary software, you may well be concerned about including GPL licenses. Similarly, an organization can use the SBOM to enforce policies about which components are permitted in their images.

You almost certainly want to store the SBOM along with the image that it refers to. You can either upload it to the registry using OCI artifact support with a reference to the image, or you can sign and attach it to the image. Let's consider how you can sign images and other artifacts.

Signing Images and Software Artifacts

Image signing associates a cryptographic identity with an image (in much the same way as certificates are signed, which is covered in Chapter 11). This makes it possible to verify that an image was created by a trusted party and has not been modified since it was built. Other artifacts—including SBOMs, as mentioned earlier—can also be signed to prove who supplied them.

Docker Content Trust, built on the `notary` open source project, was the original approach to image signing. The notary approach was widely considered to be too complex and was superseded by a version 2 project whose CLI tool is called `notation`. This project is backed by major players in the cloud ecosystem, including AWS, Microsoft, and Docker.

An alternative approach originated at Google is `sigstore`, now owned by the OpenSSF (a sister foundation to the CNCF). This has become the de facto open source standard tooling for signing container images and other artifacts in the Kubernetes ecosystem. It consists of three main components:

`cosign`
: Used to sign and verify files and artifacts such as container images, Helm charts, and SBOM files

`fulcio`
: A certificate authority that issues short-lived certificates

`rekor`
: Records signing operations in an immutable log

Perhaps the biggest advance in Sigstore over prior approaches to content trust is that it supports *keyless signing*. Instead of using private keys, which would typically require an organization to have a whole private key infrastructure in place, Sigstore uses ephemeral certificates that are issued automatically, based on common OIDC identity providers such as Google, Microsoft, or GitHub accounts. Users don't need to worry about rotating and distributing keys.

You might well want to attach the SBOM for the image before signing it. Attaching the SBOM can be done like this:

```
$ cosign attach sbom --sbom sbom.json my-image:v0.0.1
```

Before you sign and store that image in a registry, you might also want to attach a build attestation.

Build Attestations

Verifying the signature of an image confirms that it comes from a trusted party, but how do you know whether anyone tampered with the build process during the creation of the image and its component parts? A build attestation describes how an image was built, including information such as:

- Source repository and commit
- Build environment
- Toolchain versions
- Build parameters and outputs
- Details about who or what triggered the build and when

Attestations can be stored in a container registry as OCI metadata that accompanies a container image and then verified when the image is being deployed.

There are tools within the SLSA project for generating provenance records on GitHub actions (`slsa-github-generator`) or on the command line (`slsa-provenance`). Best practice would be to sign the attestation using a tool such as `cosign`.

As well as tooling provided by the OpenSSF through the SLSA project, another project that addresses concerns about the supply chain for container images is `in-toto`. This framework ensures that each of an expected set of build steps ran completely, produced the correct output given the correct input, and was performed in the right order by the right people. Multiple steps are chained together, with `in-toto`

carrying security-related metadata from each step through the process. The result is to ensure that software in production is verifiably running the same code as the developer shipped from their laptop.

You can use `cosign` to attach the attestation to an image like this:

```
$ cosign attach attestation my-image:v0.0.1 --attestation ./my-image-sbom.att.json
```

Attaching SBOM and attestations is optional but is a good security practice and will help you achieve a higher SLSA rating for your processes. Once you have the attachments in place, you are ready to sign the image, like this:

```
$ cosign sign my-image:latest
```

Signed images can be stored in an OCI registry.

What if you want to use a container image from a third party, either directly as an application or as a base image in your builds? You can check that the image came from the supplier you expect using `cosign verify`. I'll discuss this again later in this chapter.

An interesting public registry to be aware of is ttl.sh, where you can push images anonymously for temporary use. This could be useful as part of a CI/CD pipeline, where testing might be parallelized across multiple machines, all of which need access to a newly built image. You specify the length of time, up to 24 hours, that the image will be available. Images can be obfuscated behind a Universally Unique Identifier (UUID) name (which is long enough to be highly likely to be unique). A UUID name makes it a bit harder for an attacker to find an image, but it is publicly available, so I would not recommend this for anything you really need to keep confidential.

Now that you know about signing images, you probably agree that it's a good idea to sign an image before pushing to this registry and verifying the signature is intact when you pull the image again.

Image Manifests

When signatures and attestations are included in an image, they are part of the *image manifest*, which you read about in Chapter 6, where you saw `docker inspect` being used to reveal the manifest. Another tool to help if you're curious to look at the manifest for an image is the ORAS project from the CNCF. The project website has a lot of information on manifest formats if you want to dive into this topic further.

For this example, I'm using Sigstore's `rekor` image from GitHub's container registry as it has a lot of the recommended information attached. Unfortunately, at the time of writing at least, a lot of standard, freely available library images do not come with signatures and attestations.

Here's an example of fetching the manifest, this time for the alpine image stored on Docker Hub:

```
$ oras manifest fetch docker.io/library/alpine:latest | jq
{
  "manifests": [
    {
      "annotations": {
        "com.docker.official-images.bashbrew.arch": "amd64",
        "org.opencontainers.image.base.name": "scratch",
        "org.opencontainers.image.created": "2025-05-30T18:04:04Z",
...
        "org.opencontainers.image.version": "3.22.0"
      },
      "digest": "sha256:08001109a7d6...6a7338ce",                ❶
      "mediaType": "application/vnd.oci.image.manifest.v1+json",
      "platform": {
        "architecture": "amd64",
        "os": "linux"
      },
      "size": 1022
    },
    {
      "annotations": {
        "com.docker.official-images.bashbrew.arch": "amd64",
        "vnd.docker.reference.digest": "sha256:08001109a7d6...6a7338ce",
        "vnd.docker.reference.type": "attestation-manifest"       ❷
      },
      "digest": "sha256:bd4199eb785a...859f0827f",
      "mediaType": "application/vnd.oci.image.manifest.v1+json",
      "platform": {
        "architecture": "unknown",
        "os": "unknown"
      },
      "size": 838
    },
...
}
```

This is similar to the `docker manifest inspect` output you've seen before. For clarity I only included (abbreviated) output related to the `amd64` platform; if you try this for yourself, you'll see similar content for each of several other chip architectures. This time I want to point out an attestation included in this manifest:

❶ The first item in this list refers to the image for the `amd64` platform, with a digest beginning with 080011.

❷ The next item is the "attestation-manifest" for that image, referring to it by digest.

Let's take a look at that attestation manifest (I have abbreviated the digests for clarity):

```
$ oras copy docker.io/library/alpine@sha256:bd419...f0827f --to-oci-layout
alpine-attestation

✓ Copied  application/vnd.in-toto+json
79.1/79.1 KB 100.00%    4ms
  └─ sha256:5ff8d...7ea6a
✓ Copied  application/vnd.in-toto+json
5.55/5.55 KB 100.00%    2ms
  └─ sha256:0775b...09944
✓ Copied  application/vnd.oci.image.config.v1+json
241/241  B 100.00%    2ms
  └─ sha256:7a444...41d36
✓ Copied  application/vnd.oci.image.manifest.v1+json
838/838  B 100.00%    3ms
  └─ sha256:bd419...0827f
Copied [registry] docker.io/library/alpine@sha256:bd419...f0827f => [oci-layout]
alpine-attestation
Digest: sha256:bd419...f0827f
```

You can see that the first two items in the output refer to `in-toto` attestations. The files have been extracted as an OCI layout to a directory called `alpine-attestation`. You've seen OCI layouts earlier in Chapter 4. Inside this layout are some blobs that correspond to the digests for each of the items:

```
$ file blobs/sha256/*
blobs/sha256/0775b...09944: ASCII text, with very long lines (5682), with no line
terminators
blobs/sha256/5ff8d...7ea6a: JSON data
blobs/sha256/7a444...41d36: JSON data
blobs/sha256/bd419...0827f: JSON data
```

The JSON data file with the SHA starting `5ff8d` is one of the `in-toto` attestations. The content is long, so I'll just show a few parts to give you an idea of the contents:

```
  "_type": "https://in-toto.io/Statement/v0.1",
  "predicateType": "https://spdx.dev/Document",
  "subject": [
    {
      "name": "pkg:docker/alpine@3.22.0?platform=linux%2Famd64",
      "digest": {
        "sha256": "08001...338ce"
...
  "predicate": {
```

```
    "spdxVersion": "SPDX-2.3",
    "dataLicense": "CC0-1.0",
    "SPDXID": "SPDXRef-DOCUMENT",
    "name": "sbom",
    "documentNamespace": "https://docker.com/docker-scout/fs/sbom-15dbb…9910858",
    "creationInfo": {
      "creators": [
        "Organization: Docker, Inc",
        "Tool: docker-scout-1.18.0",
        "Tool: buildkit-0.16.0-tianon"
      ],
      "created": "2025-05-30T18:04:21Z"
    },
    "packages": [
...
     {
        "name": "busybox-binsh",
        "SPDXID": "SPDXRef-Package-1cf4a...06f01",
        "versionInfo": "1.37.0-r18",
...
        "licenseDeclared": "GPL-2.0-only",
        "description": "busybox ash /bin/sh",
        "externalRefs": [
          {
            "referenceCategory": "PACKAGE-MANAGER",
            "referenceType": "purl",
            "referenceLocator": "pkg:apk/alpine/busybox-binsh@1.37.0-r18?os_name
            =alpine&os_version=3.22"
...
```

For the purposes of this book, we don't need to dig into all the details of this attestation, but there is enough here to see that it refers to an `alpine` package, and it includes an SBOM created by `docker-scout` and `buildkit` tools, and one of the packages listed in that SBOM is a `busybox` shell, licensed under GPL-2.0.

In day-to-day operations, you won't need to look at manifests, attestations, or signatures like this, but one day it could be helpful to know that you have this information in hand, particularly if you are trying to get to the root cause of a supply chain security problem.

Image Deployment Security

The main security concern at deployment time is ensuring that the correct image gets pulled and run, although there are additional checks you might want to make through what is called *admission control*.

Deploying the Right Image

As you saw in "Identifying Images" on page 81, container image tags are generally not immutable—they can be moved to different versions of the same image (though some registries, including AWS Elastic Container Registry, support immutable tags). Referring to an image by its digest, rather than by tag, can help ensure that the image is the version that you think it is. However, if your build system tags images with semantic versioning and this is strictly adhered to, this may be sufficient and easier to manage since you don't necessarily have to update the image reference for every minor update.

If you refer to images by tag, you should always pull the latest version before running in case there has been an update. Fortunately, this is relatively efficient since the image manifest is retrieved first, and image layers have to be retrieved only if they have changed.

In Kubernetes, this is defined by the `imagePullPolicy`. An image policy to pull every time is unnecessary if you refer to images by digest, since any update would result in a change to the digest.

Malicious Deployment Definition

When you are using a container orchestrator, there typically are configuration files—YAML for Kubernetes, for instance—that define the containers that make up each application. It's just as important to verify the provenance of these configuration files as it is to check the images themselves.

If you download YAML from the internet, please check it *very* carefully before running it in your production cluster. Be aware that any small variations—such as the replacement of a single character in a registry URL—could result in a malicious image running on your deployment.

Verifying the Image Signature and Provenance

There's no point having signed images if you don't check the signature! Verifying the signature on the image ensures that it comes from the trusted source you are expecting, whether that is your own build pipeline or a vendor. Verifying the provenance assures that all the steps in the image build process were carried out as expected. If you are following SLSA standards, and/or a regulatory frameworks like the US Executive Order 14028,[3] you'll need to perform these verifications for compliance reasons.

3 In 2021, following the SolarWinds incident, where software was found to have been tampered with through exploits in the build process, a US Presidential Executive Order set requirements related to software supply chain security and integrity.

I've just suggested (in "Deploying the Right Image" on page 102) that you might want to pull images by tag so that you get the latest updates for that version. Checking the signature allows you to confirm that this is an intended change and that you have not pulled an image that has been tampered with.

You can verify an image signature using `cosign`, like this:

```
cosign verify --key supplier.pub myimage:tag
```

In this instance, `supplier.pub` is the public key for the supplier of the image. You can similarly use `cosign verify-attestation` to verify build attestations. The SLSA project also supplies the slsa-verifier tool for verifying provenance information.

These verification steps can be part of an automated predeploy step, built into a CI/CD pipeline or using a Kubernetes admission controller.

Admission Control

An admission controller such as Kyverno or OPA Gatekeeper can perform checks at the point where you are about to deploy a resource into a cluster. In Kubernetes, admission control can evaluate any kind of resource against policies, but for the purposes of this chapter, I will just consider an admission controller that is checking whether to permit a container based on a particular container image, as defined by some admission control policy. If the admission control checks fail, the container does not run.

Admission control policies can include several vital security checks on the container image before it is instantiated into a running container:

- Has the image been scanned for vulnerabilities/malware/other policy checks?
- Does the image come from a trusted registry?
- Is the image signed by a trusted party?
- Is the image approved?
- Does the image run as root?

These checks ensure that no one can bypass checks earlier in the system. For example, there is little advantage in introducing vulnerability scanning into your CI pipeline if it turns out that people can specify deployment instructions that refer to images that haven't been scanned.

Summary

You have seen how the container runtime needs a root filesystem and some configuration information. You can override the config using parameters that can be passed in at runtime or configured in Kubernetes YAML. Some of these configuration settings have a bearing on application security. There also will be plenty of opportunities to introduce malicious code into container images if you don't follow the best practices listed in "Dockerfile Best Practices for Security" on page 91.

Supply chain security tools allow for images to be signed, along with information about their contents and how they were built. At the point where images are deployed, orchestrators and security tools allow for admission controllers, which present an opportunity to perform security checks on those images, including verifying signatures and provenance information.

Container images encapsulate your application code and any dependencies on third-party packages and libraries. Chapter 8 looks at how these dependencies could include exploitable vulnerabilities and examines tooling to identify and eliminate those vulnerabilities.

CHAPTER 8

Software Vulnerabilities in Images

Patching software for vulnerabilities has long been an important aspect of maintaining the security of deployed code. It has become a more urgent problem as the time for a threat actor to exploit a vulnerability has been getting shorter. A 2024 report from Mandiant put it at just five days, and with AI being used maliciously to invent and automate attacks, it's probably significantly shorter already.

This is just as relevant in the world of containers as in traditional software deployment, but as you will see in this chapter, the patching process has been completely reinvented. But first, let's cover what software vulnerabilities are and how they are published and tracked.

Vulnerability Research

A vulnerability is a known flaw in a piece of software that an attacker can take advantage of to perform some kind of malicious activity. As a general rule, you can assume that the more complex a piece of software is, the more likely it is to have flaws, some of which will be exploitable.

When there is a vulnerability in a common piece of software, attackers may be able to take advantage of it wherever it is deployed, so there is an entire research industry devoted to finding and reporting new vulnerabilities in publicly available software, especially operating system packages and language libraries. You have probably heard of some of the most devastating vulnerabilities, like Shellshock, Meltdown, and Heartbleed, which get not just a name but sometimes even a logo. These are the rock stars of the vulnerability world, but they are a tiny fraction of the thousands of issues that get reported every year.

These days, it's not just human researchers who are finding and reporting new vulnerabilities in existing software: AI is actively discovering them too. Early in January

2025, Microsoft credited unpatched.ai, an automated vulnerability discovery platform, with reporting multiple high-severity issues. DARPA's (the Defense Advanced Research Projects Agency's) Artificial Intelligence Cyber Challenge is a competition for AI-driven tooling to find and patch vulnerabilities in infrastructure code, including the Linux kernel, Jenkins, and SQLite. AI-enhanced fuzzing techniques used by Google's Open Source Security Team are identifying increasing numbers of issues.

Unfortunately, although AI has been used to find real issues, it has also been used to create a lot of "AI-slop" security reports about "vulnerabilities" that aren't real. This wastes inordinate amounts of time from project maintainers and is likely to drive some of them to burnout. If you're thinking of using AI as a shortcut to amassing bug bounties, please think again.

Once a vulnerability is identified, the race is on to get a fix published so that users can deploy that fix before attackers take advantage of the issue. If new issues were announced to the public straightaway, this would create a free-for-all for attackers to take advantage of the problem. To avoid this, the concept of responsible security disclosures has been established. The security researcher who finds a vulnerability contacts the developer or vendor of the software in question. Both parties agree on a time frame, after which the researcher can publish their findings. There is some positive pressure here for the vendor to make efforts to provide a fix in a timely fashion, as it's better for both the vendor and its users that a fix is available before publication.

A new issue will get a unique identifier that begins with "CVE," which stands for Common Vulnerabilities and Exposures, followed by the year. For example, the Shellshock vulnerability was discovered in 2014 and is officially referred to as CVE-2014-6271. The organization that administers these IDs is called MITRE, and it oversees more than 400 CVE Numbering Authorities (CNAs) that can issue CVE IDs within certain scopes. Some large software vendors—for example, Microsoft, Red Hat, and Oracle—are CNAs entitled to assign IDs for vulnerabilities within their own products. GitHub became a CNA toward the end of 2019.

As CVEs are identified, they are recorded at the CVE website, and then these identifiers are used in the National Vulnerability Database (NVD) to keep track of the software packages and versions that are affected by each vulnerability.[1] At first glance, you might be tempted to think that's the end of the story. There's a list of all the package versions that are affected, so if you have one of those versions, you are exposed. Unfortunately, it's not as simple as that, because depending on the Linux distribution you're using, it might have a patched version of the package.

1 The NVD is a US organization, widely used globally, but other vulnerability databases and security advisories exist—for example, from the EU—and there are nation-specific databases such as those run by France, Germany, Japan, and Brazil. They largely use, or at least refer to, the CVE numbering scheme.

As you saw in Chapter 7, best practice is to have an SBOM for each container image that records all the software components and their versions. When new vulnerabilities are disclosed, they can be matched against SBOMs to identify which images are affected.

Vulnerabilities, Patches, and Distributions

Let's take a look at Shellshock as an example. This was a critical vulnerability that affected the GNU `bash` package, and the NVD's page for CVE-2014-6271 has a long list of vulnerable versions ranging from 1.14.0 to 4.3. If you're running a very old installation of Ubuntu 12.04 and you found that your server has `bash` version 4.2-2ubuntu2.2, you might think that it is vulnerable because it's based on `bash` 4.2, which is included in the NVD's list for Shellshock.

In fact, according to the Ubuntu security advisory for the same vulnerability, that exact version has the fix for the vulnerability applied, so it's safe. The Ubuntu maintainers decided that rather than require everyone on 12.04 to upgrade to a whole new minor version of `bash`, they would apply the patch for the vulnerability and make that patched version available.

To get a real picture of whether the packages installed on a server are vulnerable or not, you would need to reference not just the NVD but also the security advisories that apply to your distribution.

So far this chapter has considered packages (like `bash` in the preceding example) that are distributed in binary form through package managers such as `apt`, `yum`, `rpm`, or `apk`. These packages are shared across all the applications in a filesystem, and on a server or virtual machine, the fact that they are shared can cause no end of problems: one application may depend on a certain version of a package that turns out to be incompatible with another application that you want to run on the same machine. This issue of dependency management is one of the problems that containers can address by having a separate root filesystem for each container.

Application-Level Vulnerabilities

There are also vulnerabilities to be found at the application level. Most applications use third-party libraries that are typically installed using a language-specific package manager. Node.js uses `npm`, Python uses `pip`, Java uses `Maven`, and so on. In compiled languages like Go, C, and Rust, your third-party dependencies could be installed as shared libraries, or they could be linked into your binary at build time.

The third-party packages installed by these tools are another source of potential vulnerabilities. In Chapter 7, we covered how language-specific tools can generate detailed SBOMs, which can help catch vulnerabilities in the dependencies pulled in by `pip`, `npm`, `Maven`, etc.

A standalone binary executable by definition (through the word *standalone*) has no external dependencies. It may have dependencies on third-party libraries or packages, but these are built into the executable. In this case you have the option of creating a container image based on the scratch (empty) base image, which holds nothing but your binary executable.

If an application doesn't have any dependencies, it can't be scanned for published package vulnerabilities. It could still have flaws that render it exploitable by attackers, which we will consider in "Zero-Day Vulnerabilities" on page 120.

Vulnerability Risk Management

Dealing with software vulnerabilities is an important aspect of risk management. It's very likely that a deployment of any nontrivial software will include some vulnerabilities, and there is a risk that systems will be attacked through them. To manage this risk, you (or the security team in your organization) need to be able to identify which vulnerabilities are present and assess their severity, prioritize them, and have processes in place to fix or mitigate these issues.

Just because a vulnerability is present, it might not be relevant or exploitable in your application. For example, suppose a library contains two functions called `decent_code()` and `poorly_written()`. An application might import this library because it uses the `decent_code()` function. The `poorly_written()` code might still be present, but if it's never called, it's not reachable through that application.

If vulnerabilities are present but not actually exploitable, they are false positives that create more work for the security team to review. This is where VEX comes in to help.

Vulnerability Exploitability eXchange (VEX) is a machine-readable format for describing the impact of known CVEs. A VEX document covers some number of CVEs and states whether a particular software artifact is one of the following:

`affected`
: The vulnerability is present and could be exploited.

`not_affected`
: The vulnerability is not applicable in this artifact.

`fixed`
: The vulnerability has been addressed in this version.

`under_investigation`
: Work is afoot to determine whether or not this artifact is affected.

The author of a piece of software can create a VEX document to inform consumers of that product about the status of CVEs. VEX documents can be signed to prove their authenticity and stored in OCI registries along with the images they describe.

Vulnerability scanners automate the process of identifying whether known vulnerabilities are present in an image. They provide information about how serious each issue is and about the software package version in which a fix was applied (if a fix has been made available). The results will likely be more accurate if VEX information is provided as an input to the scanner, which can use it to filter out the vulnerabilities that don't affect the product.

Vulnerability Scanning

If you search the internet, you will find a huge range of vulnerability scanning tools encompassing various techniques, including port scanning tools like `nmap` and `nessus`, which attempt to find vulnerabilities on a live running system by probing it from outside. This is a valuable approach, but it's not what we are considering in this chapter. Here, we are more interested in tools that help you find vulnerabilities by examining the software that is installed in a root filesystem. The same approach is taken by tooling that generates SBOMs for a container image. In fact, understanding the components that are included in an image is a required step in vulnerability scanning anyway, so it's not a surprise that many vulnerability scanning tools like `trivy` have been enhanced so that they can also generate SBOMs.

Let's suppose you have a container image that doesn't (yet) have an SBOM. To identify which vulnerabilities are present, the first task is to establish what software is present. Software gets installed through several different mechanisms:

- The root filesystem starts from a distribution of a Linux root filesystem, which could have vulnerabilities within it.
- There could be system packages installed by a Linux package manager like `rpm` or `apk` and language-specific packages installed by tools like `pip` or `RubyGems`.
- You might have installed some software directly using `wget`, `curl`, or even FTP.

Some vulnerability scanners will query package managers to get a list of the installed software. If you're using one of those tools, you should avoid installing software directly as it won't be scanned for vulnerabilities.

Installed Packages

As you have seen in Chapter 6, each container image could include a Linux distribution, possibly with some packages installed, along with its application code. There could be many running instances of each container, each of which has its own copy of the container image filesystem, including any vulnerable packages that might be included therein. This is illustrated in Figure 8-1, where there are two instances of container X and one instance of container Y. In addition, the illustration shows some packages installed directly onto the host machine.

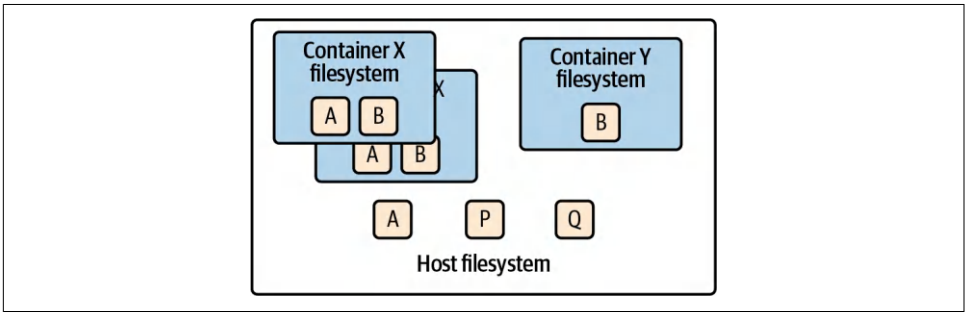


Figure 8-1. Packages on host and in containers

Installing packages directly onto hosts is nothing new. In fact, it is exactly these packages that system administrators have traditionally had to patch for security reasons. This was often achieved by simply SSH-ing into each host and installing the patched package. In the cloud native era, this is frowned upon, because manually modifying the state of a machine in this way means that it can't be automatically re-created in the same state. Instead, it's better either to build a new machine image with the updated packages or to update the automation scripts used to provision images so that new installations include the updated packages.

Container Image Scanning

To know whether your deployment is running containers with vulnerable software, you need to scan all the dependencies within those containers. There are some different approaches you could take to achieve this.

Imagine a tool that can scan each running container on a host (or across a deployment of multiple hosts). In today's cloud native deployments, it's common to see hundreds of instances of containers initiated from the same container image, so a scanner that takes this approach would be very inefficient, looking at the same dependencies hundreds of times. It's far more efficient to scan the container image from which these containers were derived.

However, this approach relies on the containers running only the software that was present in the container image and nothing else. The code running in each container must be *immutable*. Let's see why it's a good idea to treat containers as immutable in this way.

Immutable Containers

There is (usually) nothing to stop a container from downloading additional software into its filesystem after it starts running. Indeed, in the early days of containers, it was not uncommon to see this pattern, as it was considered a way to update the container to the latest version of software without having to rebuild the container image. If this idea hadn't occurred to you before now, please try to wipe it from your memory straightaway, as it's generally considered a very bad idea for several reasons, including these:

- If your container downloads code at runtime, different instances of the container could be running different versions of that code, but it would be difficult to know which instance is running what version. Without a stored version of that container's code, it can be hard (or even impossible) to re-create an identical copy. This is a problem when trying to reproduce field issues.
- An SBOM for the image no longer represents the contents of the container if you change them. Any results based on the SBOM to inform you about licensing or vulnerability issues in your running containers would be inaccurate.
- It's harder to control and ensure the provenance of the software running in each container if it could be downloaded at any time and from anywhere. Any provenance attestations in the container image become practically meaningless as they no longer reflect the actual code being run.
- Building a container image and storing it in a registry is simple to automate in a CI/CD pipeline. It's also easy to add additional security checks—like vulnerability scanning or verification of the software supply chain—into the same pipeline.

A lot of production deployments treat containers as immutable simply as a best practice but without enforcement. There are tools that can automatically enforce container immutability by preventing an executable from running in a container if that executable wasn't present in the image when it was scanned. This is known as *drift prevention* and is discussed further in Chapter 15.

Another way to achieve immutability is to run the container with a read-only filesystem. You can mount a writable temporary filesystem if the application code needs access to writable local storage. This may require changes to the application so that it writes only to this temporary filesystem.

By treating your containers as immutable, you only need to scan each image to find all the vulnerabilities that might be present in all the containers. But unfortunately, scanning just once at a single point in time may not be sufficient. Let's consider why scans have to happen on a regular basis.

Regular Scanning

As discussed at the beginning of this chapter, there is a body of security researchers around the world who are finding previously undiscovered vulnerabilities in existing code. Sometimes they find issues that have been present for years. One of the best-known examples of this is Heartbleed, a critical vulnerability in the widely used OpenSSL package that exploited a problem in the heartbeat request and response flow that keeps a TLS connection alive. The vulnerability was uncovered in April 2014, and it allowed an attacker to send a crafted heartbeat request that asked for a small amount of data in a large buffer. The absence of a length check in the OpenSSL code meant that the response would supply the small amount of data, followed by whatever happened to be in active memory to fill up the rest of the response buffer. That memory might be holding sensitive data, which would be returned to the attacker. Serious data breaches that involved the loss of passwords, Social Security numbers, and medical records were subsequently traced back to the Heartbleed vulnerability.

Cases as serious as Heartbleed are rare, but it makes sense to assume that if you're using a third-party dependency, at some point in the future a new vulnerability will be uncovered in it. And unfortunately there is no way of knowing when that will happen. Even if your code doesn't change, there is a possibility that new vulnerabilities have been uncovered within its dependencies.

Regularly rescanning container images allows the scanning tool to check the contents against its most up-to-date knowledge about vulnerabilities (from the NVD and other security advisory sources). A common approach is to rescan all deployed images every 24 hours, in addition to scanning new images as they are built, as part of an automated CI/CD pipeline.

Scanning Tools

There are numerous container image scanning tools, from open source implementations like Trivy and Grype to commercial solutions from companies like JFrog, Palo Alto, and Aqua. Many container image registry solutions, such as Docker Hub with Docker Scout and the CNCF project Harbor, as well as the registries provided by all the major public clouds, include scanning as a built-in feature.

Most, if not all, of these tools can use common SBOM and VEX formats as input to speed scanning and give more relevant results based on whether CVEs are not merely present but also exploitable.

In a Kubernetes environment, the Trivy operator can automatically scan your deployments for a variety of issues including dependency vulnerabilities, generating Prometheus metrics so that you can spot issues in a dashboard.

Liquid Reply published a walkthrough of installing Trivy and its operator and visualizing the Prometheus output in Grafana dashboards.

Unfortunately, the results you get from different scanners vary considerably, and it's worth considering why.

Sources of Information

As discussed earlier in this chapter, there are various sources for vulnerability information, including per-distribution security advisories. Red Hat even has more than one: its OVAL feed includes only vulnerabilities for which there is a fix, while the Red Hat Security Data API includes the status of unfixed CVEs.

If a scanner doesn't include data from a distribution's security feed and is relying just on the underlying NVD data, it is likely to show a lot of false positives for images based on that distribution. If you prefer a particular Linux distribution for your base images, or a solution like distroless, make sure that your image scanner supports it.

Out-of-Date Sources

Occasionally the distribution maintainers change the way they are reporting vulnerabilities. This happened with Alpine, which stopped updating its advisories at `alpine-secdb` in favor of tracking vulnerability fixes via metadata in the `aports` package build system. All the actively maintained scanning projects that I'm aware of have updated to use the `aports` metadata (or their own scraping approach), but it's an example of how a structural change like this in the future would require significant updates to scanning tools.

Won't Fix Vulnerabilities

Sometimes the maintainers of a distribution will decide that they are not going to fix a particular vulnerability (perhaps because it's a negligible risk and the fix is nontrivial or because the maintainers have concluded that interactions with other packages on their platform mean the vulnerability is impossible to exploit).

Given that the maintainers are not going to provide a fix, it becomes something of a philosophical question for scanner tool developers: considering it's not actionable, do you show the vulnerability in the results or not? While I was at Aqua, we heard from some of our customers that they don't want to see this category of result, so we provided an option to give the user the choice. It just goes to show that there is no such thing as a "correct" set of results when it comes to vulnerability scanning.

VEX Input

As discussed earlier in this chapter, VEX information can supply additional information about whether vulnerabilities are actually exploitable in a given product. If a scanner can take VEX information into account, it can produce fewer false positives.

Subpackage Vulnerabilities

Sometimes a package is installed and reported by the package manager, but in fact it consists of one or more subpackages. A good example of this is the `bind` package on Ubuntu. Sometimes this is installed with only the `docs` subpackage, which, as you might imagine, consists only of documentation. Some scanners assume that if the package is reported, then the whole package (including all its possible subpackages) is installed. This can result in false positives where the scanner reports vulnerabilities that can't be present because the guilty subpackage is not installed.

Package Name Differences

The source name for a package may include binaries that have completely different names. For example, in Debian, the `shadow` package includes binaries called `login`, `passwd`, and `uidmap`. If the scanner doesn't take this into account, it can result in false negative results.

Statically Linked Executables

When developers use a compiled language like Go, C, or Rust, they often make use of libraries. These libraries can be either loaded dynamically at runtime or compiled statically into a single executable file. A statically linked, standalone executable can be built into a container image using the scratch base image, meaning that the image has nothing in its filesystem except the executable file. In Chapter 7, I recommended this as a good approach for minimizing the attack surface.

However, this choice has an impact on what a scanner has to do to correctly detect which libraries and versions are in use. Dynamic libraries are generally installed using a package manager, so the scanner typically has access to version information from the package metadata, which is added into the container image during the image build process. None of this version information will be included in the image if the build simply copies a single standalone executable file into it.

All is not lost! The executable file typically has lots of clues about the modules or libraries that it includes, for example, in its ELF symbols and section headers. There are scanners—for example `syft`—that can perform analysis on this information and output an SBOM enumerating the modules or libraries that are built into the executable.

It's likely to be even more accurate if the compiler generates the SBOM as it is pulling in components. Recent versions of Go, for example, embed module information into a special `.note.go.buildinfo` section of the compiled executable. Scanning tools can simply compare this against their list of currently known vulnerabilities.

Scanning Multiplatform Images

As you know from Chapter 6, container images can be built to include versions for different CPU architectures, with the image manifest listing the platforms that are included. Because the software is different on each platform, the vulnerabilities that exist in, say, the ARM variant might be different from those in the Intel version.

By default, a scanner will typically retrieve the image variant only for the local architecture it's running on, though some allow you to specify a parameter to get the results for another platform. For example, you can run `trivy image --platform linux/arm64 <image>` to scan the ARM version.

Additional Scanning Features

Many scanning tools do more than just detecting image vulnerabilities, scanning for other issues such as:

- Known malware within the image, by looking for malware signatures or using heuristic approaches to detection, such as Yet Another Research Assistant (YARA)
- Executables with the `setuid` bit (which, as you saw in Chapter 2, can allow privilege escalation)
- Images configured to run as root
- Secret credentials such as tokens or passwords
- Sensitive data in the form of credit card or Social Security numbers or such

- Policy violations (e.g., outdated dependencies, noncompliant licenses)
- Poor image configuration—for example, not including a HEALTHCHECK
- Comparing the components included in the image with an SBOM that describes what is expected (this could help spot if a malicious component was added)

Scanner Errors

As I hope this section of the book has made clear, reporting on vulnerabilities is not as straightforward as you might at first imagine. So, it's likely that in any scanner you will find cases in which there is a false positive or false negative due to a bug in the scanner or a flaw in the security advisory data feeds that the scanner reads.

That said, it's better to have a scanner in place than not. If you don't have a scanner in place to use regularly, you really have no way of knowing whether your software is prey to an easy exploit. Time is no healer in this regard—the critical Shellshock vulnerability was discovered in code that was decades old. If you rely on complex dependencies, you should expect that at some point some vulnerabilities will be found within them.

False positives can be irritating, but some tools will let you suppress or mark individual vulnerability reports as acceptable so that you can decide for yourself whether you want to ignore them going forward.

Assuming you are convinced that a scanner would be a good thing to include in your processes, let's turn to the possible options for incorporating it into your team's workflow.

Scanning in the CI/CD Pipeline

Consider a CI/CD pipeline from left to right, with "writing code" at the far left and "deploying to production" at the far right, as in Figure 8-2. It's better to remove issues as early as possible in this pipeline because doing so is quicker and cheaper, in exactly the same way that finding and fixing bugs is much more time-consuming and expensive after deployment than during development.

In a traditional host-based deployment, all the software running on a host shares the same packages. The security team in an organization would typically be responsible for updating those packages with security fixes on a regular basis. This activity is largely decoupled from the development and testing stages of each application's life cycle, and it's way over to the right in the deployment pipeline. There often can be issues where different applications share the same package but need different versions, requiring careful dependency management and, in some cases, code changes.

In contrast, as you saw in Chapter 6, in a container-based deployment, each image includes its own dependencies, so different application containers can have their own

versions of each package as needed. There is no need to worry about compatibility between app code and the set of dependencies they use. This, plus the existence of container image scanning tools, allows vulnerability management to "shift left" in the pipeline.

Teams can include vulnerability scanning as an automated step. When a vulnerability needs to be addressed, developers can do this by updating and rebuilding their application container image to include the patched version. Security teams no longer need to do this manually.

There are a few places where scanning can be introduced, as illustrated in Figure 8-2.

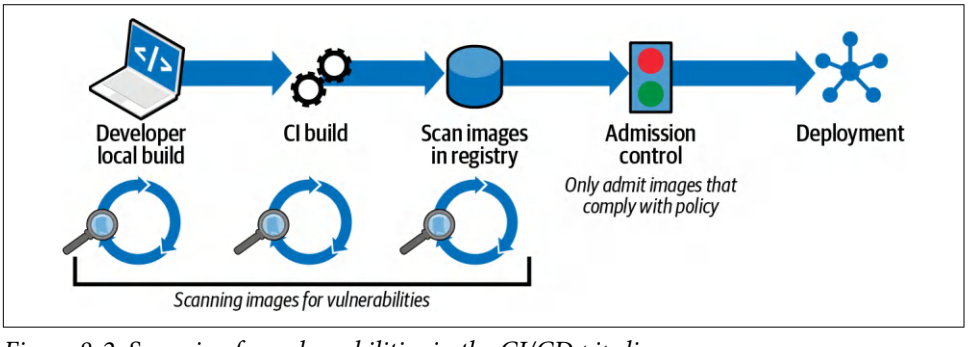


Figure 8-2. Scanning for vulnerabilities in the CI/CD pipeline

Developer scanning
: If you use a scanner that is easy to deploy on the desktop, individual developers can scan their local image builds for issues, giving them the chance to fix them before they push to a source code repository.

Scan on build
: Consider incorporating a scanning step immediately after the container image is built in your pipeline. If the scan reveals vulnerabilities above a certain severity level, you can fail the build to ensure that it never gets deployed. Figure 8-3 shows the output from an AWS CodeBuild project that builds an image from a Dockerfile and then scans it. In this example, a high-severity vulnerability was detected, and this has caused the build to fail.

Registry scans
: After the image has been built, the pipeline typically pushes it to an image registry (ideally with SBOM, build attestations, and signature attached). It's a good idea to regularly scan images in case a new vulnerability has been found in a package that's used by an image that hasn't been rebuilt in a while. If there is a comprehensive SBOM, this can speed up the scan as there is no need to reexamine the (unchanged) image itself, and the scan only involves checking the itemized image contents from the SBOM against vulnerability databases.

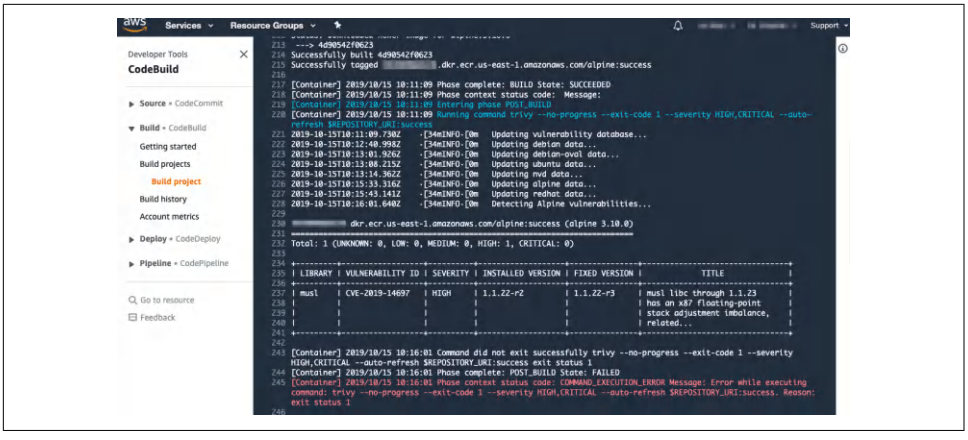


Figure 8-3. Example of failing a build when a high-severity vulnerability is detected

The following articles have useful details on how to incorporate various scanners within different CI/CD pipeline solutions:

- "Scanning Images with Trivy in an AWS CodePipeline"
- "Container Scanning" (from GitLab)

If you have multiple CPU architectures in your deployment, you might also like Andrew Blooman's example of a multiarchitecture build and scan pipeline on GitLab.

You probably don't want to leave the scan step until the point of deployment, for the simple reason that you would scan every instance of the container as it gets instantiated, even though these instances all come from the same container image. Assuming that you can treat the container as immutable, it's the image and not the container that you should scan.

Prevent Vulnerable Images from Running

It's one thing to use a scanner to establish whether an image has any significant vulnerabilities, but you also need to make sure that vulnerable images don't get deployed. This can be done as part of the admission control step that we considered in Chapter 7, as indicated in Figure 8-2. If there isn't a check to ensure that only scanned images can be deployed, it would be relatively easy to bypass the vulnerability scanner.

Generally speaking, commercial vulnerability scanners are sold as part of a broader platform that also correlates admission control with scan results. In a Kubernetes deployment, you can use Kyverno or the Open Policy Agent Gatekeeper to enforce custom admission control checks—which could include checking that images have passed their vulnerability scan—and to enforce thresholds on vulnerability scores (for example, block images with known High or Critical CVEs). As you saw in Chapter 7, you can also add policy checks to ensure that images are signed by trusted suppliers.

With scanning in place, you will find out that you have vulnerable container images. What next?

Updating Images

Once you have identified that your images are vulnerable and that fixed versions of dependencies are available, you need to rebuild those images to use the updated packages and redeploy them. How fast do you need to do this?

Guidelines vary but generally advise a timeline of several weeks for deploying updated or patched software. For example, the FedRAMP Continuous Monitoring Strategy Guide (download) says that "All vendor dependencies at a high risk level must be mitigated to a moderate level through compensating controls within 30 days," and the CISA Binding Directive from 2019 says that "Critical vulnerabilities must be remediated within 15 calendar days of initial detection."

However, do you really want to leave your systems exposed to a critical vulnerability for two weeks or more, when the Mandiant team[2] found that the average time to exploit in 2023 was only five days? The mind boggles at how many attacks could be attempted in the traditional 30- or 45-day "patch cycle."

From the point when a new fixed vulnerability is disclosed, there are several steps:

1. Image or SBOM scanning needs to spot that an update is needed.
2. Code, lockfiles, or scripts have to be updated with the patched version.
3. Container images need to be rebuilt and tested.
4. Updated versions can now be deployed to production. This might be a "rolling upgrade" such that old container instances are gradually retired while new instances are deployed, without causing disruption to the user.

2 Google's Mandiant team publishes a very good, if somewhat terrifying, annual report on the state of the threat landscape.

Automation is helping here. For example, GitHub's Dependabot can suggest pull-request changes to your code to update vulnerable packages in your repositories. This is another fast-moving area where AI is likely to provide some assistance (though I suspect that AI will likely be just as helpful to attackers as to security teams).

While these steps are being carried out, the deployment is still vulnerable, and I hope it's clear to you that the faster you can deploy the updated code, the less likely you are to suffer a serious breach. Unfortunately it's not possible to reduce this time to zero—even if you're cavalier about how much testing you do! While you're waiting for the updated images to be ready to deploy, there are new approaches that use eBPF to mitigate vulnerabilities in existing containers. We'll discuss this in Chapter 15.

So far in this chapter we have discussed known vulnerabilities in dependencies that your application code relies on. But this misses out on an important category of vulnerabilities called *zero days*.

Zero-Day Vulnerabilities

Early in this chapter, "Vulnerability Research" on page 105 discussed how there are security researchers around the world looking for new ways to exploit existing software. It stands to reason that when a new vulnerability is found, some amount of time passes before a fix is published that addresses the problem. Until a fix is made available, the vulnerability is known as a *zero-day* or *0-day* vulnerability because no days have passed since the fix was published. Mandiant's report tells us that 70% of exploited vulnerabilities in 2023 were zero days.

If it's possible for a third-party library to have a bug that an attacker can exploit, the same is true for any code—including the applications that your team is writing. Peer review, static analysis, and testing can all help to identify security issues in your code, but there's a chance that some issues will slip through. Depending on your organization and the value of its data, there may be bad actors in the world for whom it's worthwhile trying to find these flaws.

The good news is that if a vulnerability isn't published, the vast majority of potential attackers in the world don't know about it anymore than you do.

The bad news is that you can bet on the fact that sophisticated attackers and nation-state organizations have libraries of as-yet-unpublished vulnerabilities. We know this to be true from Edward Snowden's revelations. As discussed at the start of this chapter, AI-based tooling is helping to identify new vulnerabilities more easily, and I'd bet that some of them are not being reported, instead being held back for use by malicious actors.

No amount of matching against a vulnerability database is going to be able to identify a vulnerability that hasn't been published yet. Depending on the type and severity of the exploit, sandboxing as described in Chapter 10 may well protect your application and your data. Your best hope for defending against zero-day exploits is to detect and prevent anomalous behavior at runtime, which I will discuss in Chapter 15.

Summary

In this chapter, you read about vulnerability research and the CVE identifiers that are assigned to different vulnerability issues. You saw why it's important to have distributions-specific security advisory information and to not just rely on the NVD. You know why different scanners can produce different results, so you are better armed to make a decision about which tools to use. Whichever scanner you pick, I hope you're now convinced that you need container image scanning built into your CI/CD pipeline.

Some of the scanning tools discussed in this chapter are also capable of scanning for security concerns in the files that describe and configure the infrastructure where your applications run. In Chapter 9, on infrastructure as code and GitOps, let's consider the security implications of automating infrastructure provisioning and configuration.

CHAPTER 9

Infrastructure as Code and GitOps

Software runs on computers, and when there are several computers involved, we start using the term *infrastructure* to describe not just the machines themselves but also the networking that connects them and devices such as storage that are available to those computers. The way that this infrastructure is set up has a bearing on security.

This book focuses on container security, so I won't go into details about the broad topics of network and computer security. But in the cloud deployments where containers typically run in production, the tools and processes used to set up the infrastructure can have a strong influence on the security outcomes for those containers. In this chapter, I'll start by talking briefly about infrastructure as code, and then go on to describe how this approach set the foundations for GitOps, which in turn has some very significant security characteristics.

IaC

When you are going to run software in "the cloud," you will need to provision some infrastructure to run it on, whether that involves spinning up virtual machines or configuring bare metal machines. You'll want to choose the operating system(s) to deploy on those machines and connect them together, perhaps with a virtual private cloud (VPC) to isolate them from other machines and users. You might also want to set up access to cloud services like databases or messaging frameworks that your software is going to access. In a public cloud environment, the infrastructure can include managed Kubernetes services or container orchestration systems like AWS Elastic Container Service.

The idea of *infrastructure as code* (IaC) is that instead of running commands manually to perform this provisioning, you have a set of files that describe your cloud infrastructure—the virtual machines, networking, and managed cloud services that your applications will run on. These files are used as input to tools like Terraform/OpenTofu, CloudFormation, or Pulumi, which provision infrastructure according to the definitions in those files. Automating this provisioning, rather than relying on a human to manually enter commands, makes the process more reliable and reproducible.

You have probably realized that the files that describe an infrastructure configuration can—and should—be stored in source control such as Git, bringing all the usual benefits such as version history (so that faulty changes can easily be backed out), ability for multiple people to collaborate on the infrastructure and review each other's changes, and a historical record of what has been deployed that can be used for auditing or troubleshooting.

Aside from autoscaling the number of VMs in use, infrastructure doesn't typically change terribly often. To create a new deployment of a set of infrastructure (perhaps in a new geographic region or as a test environment), a user will typically run a command, or push a button in a cloud console, to trigger the automated provisioning to run. But one of the benefits of a cloud native approach to application software is the ability to frequently push updates. *GitOps* builds on the concept of IaC, applying the same ideas to workload configuration and enabling automated, dynamic updates.

GitOps

Alexis Richardson coined the term GitOps in 2017, describing a methodology in which all the configuration information about the state of a system is held under source control, just as the application source code is. When a user wants to make an operational change to the system, they don't apply commands directly but instead check in the desired state in code form (for example, in YAML files for Kubernetes). An automated system called the GitOps controller makes sure that the system is updated to reflect the latest state as defined under code control. Tools such as ArgoCD and Flux are open source examples of this and are both mature enough to have reached graduated status in the CNCF.

GitOps impacts security in significantly beneficial ways. Users no longer need direct access to the running system because everything is done at arm's length via the source code control system (typically Git, as the name implies). As shown in Figure 9-1, user credentials allow access to the source control system, but only the automated GitOps operator has permissions to modify the running system. Because Git records every change, there is an audit trail for every operation.

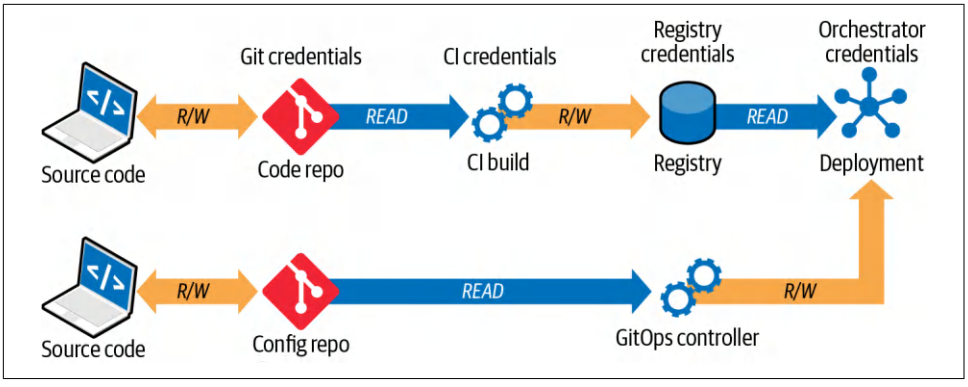


Figure 9-1. GitOps

GitOps encompasses the following principles:

Declarative
: The files used to manage a system using GitOps are declarative; that is, they describe the desired state of the system. This is in contrast to imperative code that lists a set of commands that might achieve that desired state. The great thing about a declarative approach is that it doesn't matter what state you start from: it is up to the tooling to figure out how to get from the starting state to the desired state.

Versioned and immutable
: If the files are stored in Git, there is a version history that can be used as an audit trail to see what changes have been made, by whom, and when (and the commit messages might even convey why). They are immutable—that is, a file stored in the Git repo is used unmodified by the running system, and system changes are permitted only if they are driven by files in the Git repo. Even trusted users aren't allowed to directly change configuration on the fly: they will have to run their changes through the Git repo.

Automatic pulls
: GitOps tools automatically pull the desired state information from a Git repository, with no need for manual input or control.

Continuous reconciliation
: The files stored in Git describe the desired state of the infrastructure and the set of workloads that should be running. GitOps tools repeatedly check whether the actual state of the system matches this desired state and makes changes to reconcile the running system to that desired state. If the state stored in Git changes (by someone making a change to a file and committing it into the repository), then this continuous reconciliation approach will eventually bring the running system into line with this change.

Check out OpenGitOps for further information about GitOps and its principles.

Typically GitOps goes hand in hand with Kubernetes, because Kubernetes has declarative APIs and controllers, which perform continuous reconciliation. Tools like Crossplane and the Cluster API extend the concept to use Kubernetes controllers for managing infrastructure too.

Implications for Deployment Security

Both IaC and GitOps have advantages for repeatability and automation, but let's dive into their implications for security:

Version history
: As mentioned previously, source control provides an audit trail showing who has modified files and when.

Change management
: Only users with the right Git permissions can push changes to the configuration. Git systems can enforce additional rules, for example, requiring sign-off by more than one person before a change is merged. It's good practice to make use of this so that all changes, whether to application source or to infrastructure configuration, are reviewed.

Drift prevention
: Because continuous reconciliation constantly brings the running system into line with the desired configuration, it's hard for the system to "drift" away from that configuration. Even if someone can manually apply a change (perhaps through a compromise to the system), the GitOps controller will soon bring the system back to the desired state.

Configuration scanning
: There are several tools that can automatically scan or test IaC and/or GitOps files for security concerns, including checkov, conftest, kubescape, and trivy. In just the same way that you can configure your CI/CD pipeline to not deploy images that fail your scans, configuration changes can be rejected if they don't meet scanning requirements.

Reduced permission requirements
: In a GitOps model, humans don't need direct write access to Kubernetes clusters. Only the tooling needs the ability to write resource files into the cluster, exactly as they have been pulled from Git. Human use of `kubectl` can be limited to read-only operations using RBAC controls. This reduces the attack surface and provides fewer opportunities for accidental misconfiguration. It also reinforces the immutable desired state model, reducing opportunities for drift.

Secrets management
: As discussed in Chapter 14, secrets should not be stored under source control! GitOps tools integrate with external secret management tools like Vault, Sealed-Secrets, or the External Secrets Operator to retrieve secret values when they are needed.

Rollback
: If something goes wrong for any reason, it's easy enough to use GitOps tooling to roll back to a version that was previously working correctly.

Configuration as documentation
: The configuration files stored in Git are the source of truth about how the system is supposed to operate. There is no need for separate documentation that can easily get out of sync with reality. All changes are associated with a commit, with associated message and/or pull request that explains the change.

In a GitOps model, people no longer need to directly configure what's running on a Kubernetes cluster. This dramatically improves the security posture of the cluster itself, but it pushes the focus to the GitOps tooling and the Git repos themselves. In particular, since the configuration is held in code files, those files are subject to the same kind of *supply chain attacks* as application source code, as described in Chapter 6:

- The most direct risk is called *malicious manifest injection*, where an attacker who gets access to the Git repo can control what gets deployed. If a bad actor can modify the configuration files, they can include workloads of their choice, modify RBAC permissions, or introduce sidecar containers for data exfiltration. Unless other layers of defense prevent it, the GitOps controller faithfully deploys whatever state is described in the repository, including these malicious changes.
- The dependency confusion attack discussed in Chapter 7 can apply in a GitOps context to additional types of dependency. For example, a GitOps pipeline might resolve references to Terraform modules or Helm charts during deployment, so the pipeline might be tricked into pulling malicious versions of dependencies from public registries.

To defend against these attacks and to harden the Git repo, let's consider some security best practices when you're using GitOps.

GitOps Security Best Practices

Some best practices to consider include the following, all of which can be enforced easily within tooling like GitHub or GitLab:

- Ensure that all commits are signed, to prove who made any change to a file: this is a wise move for source code of any kind, not just those repos being used to drive IaC or GitOps.
- It's a good idea to also require release tags to be signed, since if something goes wrong, you might need to roll back to a known good release, and you want that to be identified by someone trusted and authoritative.
- Use Git branch protection rules to require pull requests with approval by at least one reviewer in addition to the person making a change, especially in sensitive areas like security policies or RBAC configuration. Pull requests also provide a record of who made a change and why, along with approval information.
- Disallow force pushes, since these can rewrite history and make audit trails less reliable. Also, a force push that removes commits could conceivably confuse a GitOps controller like Flux or ArgoCD.
- Require multifactor authentication when users are signing in to Git tooling. Even better—since phone SMS codes and other factors can be phished—is to require passwordless Fast IDentity Online (FIDO) authentication that uses cryptographic credentials tied to the user's device.
- Avoid using long-lived Personal Access Tokens for CI/CD workflows, because they are vulnerable to being leaked through reuse or misuse and don't rotate automatically. Instead, have the CI/CD job authenticate itself using OIDC (OpenID Connect), and use temporary, short-lived credentials for actions such as retrieving secrets, pushing images, or deploying code.
- Require status checks from scanning tools to complete successfully before merge.
- Apply least-privilege principles to repository access so that only trusted users can push, merge, or review changes. GitHub and GitLab provide team management capabilities to help with this. Note that GitOps controllers (such as Flux or ArgoCD) need permissions to write to the Kubernetes cluster(s), but they only need permission to read from Git repos.
- Consider separating the repo used for IaC/GitOps manifests from the repo(s) used for application source code. This limits the blast radius if a repo is compromised, and makes it easier to ensure least privilege access.

- As well as audit logging, GitHub and GitLab offer additional paid security features that you might want to consider.
- Consider using admission controllers and/or runtime security tooling to detect and protect against unexpected software being deployed.

Summary

This chapter discussed why infrastructure and configuration information should be stored under code control, and described the advantages and security implications of using IaC and GitOps tools to apply that configuration to a deployment. I hope you'll find the advice on best practices useful when setting up Git repos for use with GitOps tooling.

Both Chapter 7 and this chapter considered ways to avoid deploying insecure container images and configuration, but based on the principle of defense in depth, we should assume that it's possible for some vulnerabilities to make it past these measures. In Chapter 10, we'll consider strengthening the isolation of a running container so that it's harder for a malicious actor to exploit vulnerabilities.

CHAPTER 10
Strengthening Container Isolation

In Chapters 3 and 4, you saw how containers create some separation between workloads even though they are running on the same host. In this chapter, you'll learn about some more advanced tools and techniques that can be used to strengthen the isolation between workloads.

Suppose you have two workloads and you don't want them to be able to interfere with each other. One approach is to isolate them so that they are unaware of each other, which at a high level is really what containers and virtual machines are doing. Another approach is to limit the actions those workloads can take so that even if one workload is somehow aware of the other, it is unable to take actions to affect that workload. Isolating an application so that it has limited access to resources is known as *sandboxing*.

When you run an application as a container, the container acts as a convenient object for sandboxing. Every time you start a container, you know what application code is supposed to be running inside that container. If the application were to be compromised, the attacker might try to run code that is outside that application's normal behavior. By using sandboxing mechanisms, we can limit what that code can do, restricting the attacker's ability to affect the system.

Several of these sandboxing approaches involve applying a profile when you start a container, where that profile defines operations that the container can or can't perform. There are also eBPF-based runtime security tools that effectively sandbox what a container can do, with a profile that can be updated or applied while a container is running. We'll cover these in Chapter 15.

The first sandboxing mechanism we'll consider is *seccomp*.

Seccomp

In "System Calls" on page 15, you saw that system calls provide the interface for an application to ask the kernel to perform certain operations on the application's behalf. Seccomp is a mechanism for restricting the set of system calls that an application is allowed to make.

When it was first introduced to the Linux kernel in 2005, seccomp (for "secure computing mode") meant that a process, once it had transitioned to this mode, could make only a few system calls:

- `sigreturn` (return from a signal handler)
- `exit` (terminate the process)
- `read` and `write`, but only using file descriptors that were already open before the transition to secure mode

Untrusted code could be run in this mode without being able to achieve anything malicious. Unfortunately, the side effect is that lots of code couldn't really achieve anything at all useful in this mode. The sandbox was simply too limited.

In 2012, a new approach called *seccomp-bpf* was added to the kernel. This uses Berkeley Packet Filters to determine whether a given system call is permitted, based on a seccomp profile applied to the process. Each process can have its own profile.

Berkeley Packet Filters are a precursor to eBPF, which we'll discuss in later chapters.

The BPF seccomp filter can look at the system call opcode and the parameters to the call to make a decision about whether the call is permitted by the profile. In fact, it's slightly more complicated than that: the profile indicates what to do when a syscall matches a given filter, with possible actions including returning an error, terminating the process, or calling a tracer. But for most uses in the world of containers, the profile either permits a system call or returns an error, so you can think of it as listing which systems will be allowed or denied.

This can be useful in the container world because there are several system calls that a containerized application really has no business trying to make, except under extremely unusual circumstances. For example, you really don't want any of your containerized apps to be able to change the clock time on the host machine, so it makes sense to block access to the syscalls `clock_adjtime` and `clock_settime`. Unless you want containers to be making changes to kernel modules, there is no need

for them to call `create_module`, `delete_module`, or `init_module`. There is a keyring in the Linux kernel, and it isn't namespaced, so it's a good idea to block containers from making calls to `request_key` or `keyctl`.

The Docker default seccomp profile is part of the Moby open source project and blocks more than 40 of the 400+ syscalls (including all the examples just listed) without ill effects on the vast majority of containerized applications. Unless you have a reason not to do so, it's a good default profile to use.

Kubernetes has supported the ability to configure a `seccompProfile` setting in the `podSecurityContext` for a workload since version 1.22, and the `RuntimeDefault` option for this setting uses the default profile for the container runtime (for example `containerd` uses the Moby/Docker profile). You might want to go even further and limit a container to an even smaller group of syscalls: in an ideal world, there would be a tailored profile for each application that permits precisely the set of syscalls that it needs. There are a few different possible approaches to creating this kind of profile:

- You can use `strace` to trace out all the system calls being called by your application. Jess Frazelle described how she did this to generate and test the default Docker seccomp profile in a blog post.
- For Kubernetes deployments, there is a Security Profiles Operator, which can record the syscalls used by an application and then apply them as a profile. This tool can generate AppArmor and SELinux profiles as well as seccomp—we'll discuss those shortly.
- If creating seccomp profiles yourself seems like a lot of effort, you may want to look at commercial container security tools, some of which have the ability to observe individual workloads to automatically generate custom seccomp profiles.

One thing to be aware of with seccomp profiles is that system calls continue to evolve as Linux develops over time. Since writing the first edition of this book, around 100 syscalls have been added to the kernel. Generally, application developers don't program directly to syscalls, as they are abstracted by programming language libraries, and upgrading those libraries can potentially mean a change to the underlying system calls that are used, without this change being obvious to the application developers. A strict seccomp profile might deny access to a new system call being legitimately used, so whenever the host operating system is upgraded to a new kernel version that includes new system calls, profiles might need to be updated accordingly.

If you are interested in the underlying technology behind `strace`, you might like to watch my talk at GopherCon 2017, where I created a very basic `strace` implementation in a few lines of Go.

AppArmor

AppArmor (short for "Application Armor") is one of a handful of Linux Security Modules (LSMs) that can be enabled in the Linux kernel. You can check the LSMs available on a machine by looking at the contents of the */sys/kernel/security/lsm* file.

In AppArmor, a profile can be associated with an executable file, determining what that file is allowed to do in terms of capabilities and file access permissions. You'll recall that these were both covered in Chapter 2.

Various container runtimes include support for AppArmor, including Docker, `containerd`, and `CRI-O`.

AppArmor and other LSMs implement *mandatory access controls* (MACs). A MAC is set by a central administrator, and once set, other users do not have any ability to modify the control or pass it on to another user. This is in contrast to Linux file permissions, which are *discretionary access controls* (DACs), in the sense that if my user account owns a file, I could grant your user access to it (unless this is overridden by a MAC), or I could set it as unwritable even by my own user account to prevent myself from inadvertently changing it. Using MACs gives the administrator much more granular control of what can happen on their system, in a way that individual users can't override.

AppArmor includes a "complain" mode in which you can run your executable against a profile and any violations get logged. The idea is that you can use these logs to update the profile, with the goal of eventually seeing no new violations, at which point you start to enforce the profile. Once you have a profile, you install it under the `/etc/apparmor.d` directory and run a tool called `apparmor_parser` to load it. See which profiles are loaded by looking at `/sys/kernel/security/apparmor/profiles` or by running `sudo apparmor_status`. This will show you the available profile names.

Running a container using `docker run --security-opt="apparmor:<profile name>" ...` will constrain the container to the behaviors permitted by the profile. By default, Docker will apply a default AppArmor profile, which blocks various operations such as using `ptrace` within a container. You probably won't see the profile in `/etc/apparmor.d`, though, since Docker constructs it in memory and passes it to `apparmor_parser` when the Docker daemon starts.

You can see which AppArmor profile is applied to a running container in the output from `docker inspect <container ID>`, which shows output like this:

```
"AppArmorProfile": "docker-default"
```

You can add annotations to apply an AppArmor profile on a container in a Kubernetes pod. The Security Profile Operator mentioned earlier can build and apply AppArmor profiles specific to a workload.

SELinux

SELinux, or Security-Enhanced Linux, is another LSM option in Linux. History (or at least Wikipedia) relates that it has its roots in projects by the US National Security Agency, and it's now an open source project primarily maintained by Red Hat. If you're running a Red Hat distribution (RHEL, Fedora, or Centos Stream) on your hosts, there is a good chance that SELinux is enabled already.

SElinux lets you constrain what a process is allowed to do in terms of its interactions with files and other processes. Each process runs under an SELinux *domain*—you can think of this as the context that the process is running in—and every file has a type. You can inspect the SELinux information associated with each file by running `ls -lZ`, and similarly you can add `-Z` to the `ps` command to get the SELinux detail for processes.

A key distinction between SELinux permissions and regular DAC Linux permissions (as seen in Chapter 2) is that in SELinux, permissions have nothing to do with the user identity—they are described entirely by labels. That said, they work together, so an action has to be permitted by both DAC and SELinux.

Every file on the machine has to be labeled with its SELinux information before you can enforce policies. These policies can dictate what access a process of a particular domain has to files of a particular type. In practical terms, this means you can limit an application to have access only to its own files and prevent any other processes from being able to access those files. In the event that an application becomes compromised, this limits the set of files that it can affect, even if the normal DACs would have permitted it. When SELinux is enabled, it has a mode in which policy violations are logged rather than enforced (similar to what we saw in AppArmor).

Manually creating an effective SELinux profile for an application takes in-depth knowledge of the set of files that it might need access to, in both happy and error paths, so that task may be best left to the app developer. Some vendors provide profiles for their applications.

SELinux is tightly integrated with Red Hat–maintained container runtimes `podman` and `CRI-O`. Under these runtimes, each container runs in its own SELinux domain, and file volumes can be marked with the `:z` or `:Z` flag to automatically relabel content for container access.

If you are interested in learning more about SELinux, there is a good tutorial on the subject by DigitalOcean, or you might prefer Dan Walsh's visual guide.

The security mechanisms we have seen so far—seccomp, AppArmor, and SELinux—all police a process's behavior at a low level. Generating a complete profile in terms of the precise set of system calls or capabilities needed can be a difficult job, and a small change to an application can require a significant change to the profile in order to run. The administrative overhead of keeping profiles in line with applications as they change can be a burden, and human nature means there is a tendency either to use loose profiles or to turn them off altogether. The default Docker seccomp and AppArmor profiles provide some useful guardrails if you don't have the resources to generate per-application profiles.

It's worth noting, however, that although these protection mechanisms limit what the user space application can do, there is still a shared kernel. A vulnerability within the kernel itself, like Dirty COW, would likely not be prevented by any of these tools (unless they happen to block all possible execution paths that might access the kernel vulnerability).

So far in this chapter, you have seen security mechanisms that can be applied to a container to limit what that container is permitted to do. Now let's turn to a set of sandboxing techniques that fall somewhere between container and virtual machine isolation, starting with gVisor.

gVisor

Google's gVisor sandboxes containers by intercepting system calls, in much the same way that a hypervisor intercepts the system calls of a guest virtual machine. It implements a substantial set of Linux system calls in user space through paravirtualization. As you saw in Chapter 5, paravirtualization means reimplementing instructions that would otherwise be run by the host kernel.

To do this, a component of gVisor called the Sentry intercepts syscalls from the application. Sentry is heavily sandboxed using seccomp, such that it is unable to access filesystem resources itself. When it needs to make system calls related to file access, it off-loads them to an entirely separate process called the Gofer.

Even those system calls that are unrelated to filesystem access are not passed through to the host kernel directly but instead are reimplemented within the Sentry. Essentially it's a guest kernel, operating in user space.

The gVisor project provides an executable called `runsc` that is compatible with OCI-format bundles and acts very much like the regular `runc` OCI runtime that we met in

Chapter 6. Running a container with `runsc` allows you to easily see the gVisor processes. In the following example, I am running the same bundle for Alpine Linux that I used in "OCI Standards" on page 71:

```
$ cd alpine-bundle
$ sudo runsc run sh
```

In a second terminal, you can use `runsc list` to see containers created by `runsc`:

```
$ sudo runsc list
ID  PID    STATUS   BUNDLE                    CREATED              OWNER
sh  32258  running  /home/liz/alpine-bundle   2019-08-26T13:51:21  root
```

Inside the container, run a `sleep` command for long enough that you can observe it from the second terminal. The `runsc ps` *`<container ID>`* shows the processes running inside the container:

```
$ runsc ps sh
UID       PID       PPID      C         TTY       STIME     TIME      CMD
5000      1         0         0         pts/0     11:49     0s        sh
5000      2         1         0         pts/0     11:49     10ms      sleep
```

So far, so much as expected, but things get interesting if you start to look at the processes from the host's perspective (the output here was edited to show the interesting parts):

```
$ ps fax
PID   TTY   STAT TIME COMMAND
...
26162 pts/1 Ss   0:00 \_ sudo runsc run sh
26163 pts/1 Sl+  0:00     \_ runsc run sh
26170 ?     Ssl  0:00         \_ runsc-gofer --root=/var/run/runsc gofer
                                --bundle=/home/liz/alpine-bundle ...
26175 ?     Ssl  0:14         \_ runsc-sandbox --root=/var/run/runsc boot
                                                --apply-caps=false ...
26205 ?     Ss   0:00             \_ [exe]
26211 ?     S    0:00             \_ [exe]
26213 ?     SN   0:00             |   \_ [exe]
26228 ?     S    0:00             \_ [exe]
26229 ?     SN   0:00             |   \_ [exe]
26230 ?     S    0:00             \_ [exe]
26231 ?     SN   0:00                 \_ [exe]

...
```

You can see the `runsc run` process, which has spawned two processes: one is for the Gofer; the other is `runsc-sandbox` but is referred to as the Sentry in the gVisor documentation. Sandbox has several child and grandchild processes, and looking at the process information for these child and grandchild processes from the host's perspective reveals something interesting: they are all running the `runsc` executable. For brevity, the following example shows one child and one grandchild:

```
$ sudo ls -l /proc/26230/exe
lrwxrwxrwx 1 root root 0 Jul 29 11:57 /proc/26230/exe -> /usr/bin/runsc
$ sudo ls -l /proc/26231/exe
lrwxrwxrwx 1 root root 0 Jul 29 11:57 /proc/26231/exe -> /usr/bin/runsc
```

Notably, none of these processes refers to the `sleep` executable that we know is running inside the container because we can see it with `runsc ps`. Trying to find that `sleep` executable more directly from the host is also unsuccessful:

```
$ sudo ps -eaf | grep sleep
liz  3554 3171  0 14:26 pts/2    00:00:00 grep --color=auto sleep
```

This inability to see the processes running inside the gVisor sandbox is much more akin to the behavior you see in a regular VM than it is like a normal container. And it affords extra protection for the processes running inside the sandbox: even if an attacker gets root access on a host, there is still a relatively strong boundary between the host and the running processes. Or at least there would be, were it not for the `runsc` command itself! It offers an `exec` subcommand that we can use, as root on the host, to operate inside a running container:

```
$ sudo runsc exec sh ps
PID   USER     TIME  COMMAND
  1   root     0:00  /bin/sh
 21   root     0:00  sleep 100
 22   root     0:00  ps
```

While this isolation looks very powerful, you might run into limitations:

- The first is that not all Linux syscalls have been implemented in gVisor. The project has a compatibility guide, which notes that many languages examine the available system calls and can call back to alternatives at runtime, so the majority of applications will function within gVisor. These days, gVisor even runs successfully on graphics processing units (GPUs) and tensor processing units (TPUs), commonly used for accelerating machine-learning workloads.
- The second is that performance will likely be impacted. The gVisor project published a performance guide to help you explore this in more detail. Essentially, gVisor deliberately chooses an improved security model, sacrificing some performance to achieve those security goals. There is a KVM-based platform mode for gVisor that may give you better performance on bare-metal deployments.

If you're running on Google's Cloud Platform, gVisor is readily available, and you can also use it on self-managed, vanilla Kubernetes.

As you have seen in this section, gVisor provides an isolation mechanism that more closely resembles a virtual machine than a regular container. However, gVisor affects only the way that an application accesses system calls. Namespaces, cgroups, and changing the root are still used to isolate the container.

Now let's turn to considering approaches that use virtual machine isolation for running containerized applications.

Kata Containers

As you've seen in Chapter 4, when you run a regular container, the container runtime starts a new process within the host. The idea with Kata Containers is to run containers within a separate virtual machine. This approach gives the ability to run applications from regular OCI format container images, with all the isolation of a virtual machine. The Kata Containers project is hosted by the Open Infra Foundation.

For each container, Kata creates a separate virtual machine using a "micro-VMM"—a lightweight virtual machine monitor (VMM) such as Firecracker, QEMU, or Cloud Hypervisor—we'll consider these technologies shortly.

Like gVisor, Kata Containers make a trade-off between security and performance. For many deployments, especially where workloads are essentially trusted (for example, they are all created and operated by the same business), the additional isolation is an unnecessary cost, requiring additional memory, CPU, and impacting performance, and features like shared volumes or GPU support may not be available.

Lightweight/Micro Virtual Machines

As you saw in "Disadvantages of Virtual Machines" on page 67, conventional virtual machines are slow to start, making them unsuitable for the ephemeral workloads that typically run in containers. But what if you had a virtual machine that boots extremely quickly? There are now several options of minimal virtual machines, often called "lightweight VMs" or "micro VMs," offering the benefits of secure isolation through a hypervisor and no shared kernel but designed specifically for containers and with fast startup times.

Firecracker and Cloud Hypervisor are both minimal VMMs written in Rust, which as a language provides memory-safety guarantees that help to avoid vulnerabilities and achieve startup times around 100ms. Edera has applied a similar approach, creating a hypervisor based on Xen but largely rewritten in Rust. Apple recently launched the Apple Containerization open source project, which allows containers to run in lightweight Linux VMs within a hypervisor written in Swift.

These "micro-VMMs" are able to start VMs so fast because they strip out functionality that is generally included in a kernel but that isn't required in a container. Enumerating devices is one of the slowest parts of booting a system, but containerized applications rarely have a reason to use many devices. The main saving comes from a minimal device model that strips out all but the essential devices.

There are some differences in philosophy and background between these projects:

- Firecracker originated in AWS and is used at scale for running Lambda workloads. It is designed to provide the minimal necessary functionality for running and isolating container workloads with the fastest startup times.
- Cloud Hypervisor supports more complex workloads such as nested virtualization, Windows as a guest OS, and GPU support.
- Edera takes the approach of running containers in lightweight VM-like "zones," with an emphasis on security based on stronger isolation than conventional containers.
- Apple Containerization allows containers to run in their own lightweight VMs on a Mac, without requiring a Linux virtual machine to act as the host for those containers.

As with most things in technology, there are trade-offs. The greater isolation provided by these VM-based approaches certainly gives a much stronger security boundary. Each container has its own kernel, so a "container escape" is possible only through a "virtualization escape." The downside is that without a shared kernel, eBPF-based infrastructure tools lose visibility and control over all the containers on a host and would be closer to the sidecar model with an instance per container/VM. The performance impact is likely to be minimal for many workloads, but it might make a difference.

Edera has compared different virtualization approaches in its performance benchmarking.

There is one last approach to isolation that I'd like to mention in this chapter. It's rarely used in practice, but I think it's an interesting approach that takes an even more extreme approach to reducing the size of the guest operating system: unikernels.

Unikernels

The operating system that runs in a virtual machine image is a general-purpose offering that you can reuse for any application. It stands to reason that apps are unlikely to use every feature of the operating system. If you were able to drop the unused parts, there would be a smaller attack surface.

The idea of unikernels is to create a dedicated machine image consisting of the application and the parts of the operating system that the app needs. This machine image can run directly on the hypervisor, giving the same levels of isolation as regular virtual machines but with a lightweight startup time similar to what we see in Firecracker.

Every application has to be compiled into a unikernel image complete with everything it needs to operate. The hypervisor can boot up this machine in just the same way that it would boot a standard Linux virtual machine image.

IBM's Nabla project is mostly inactive now, but it made use of unikernel techniques for containers. Nabla containers use a highly restricted set of just seven system calls, with this policed by a seccomp profile. All other system calls from the application get handled within a unikernel library OS component. By accessing only a small proportion of the kernel, Nabla containers reduce the attack surface.

Unikraft is a unikernels project under the Linux Foundation, aimed at cloud applications.

I should point out that unikernels aren't containers, but they provide a different way of isolating applications from each other on a shared host machine.

Summary

In this chapter, you have seen that there are a variety of ways to isolate instances of application code from one another, which look to some degree like what we understand as a "container":

- Some options use regular containers, with additional security mechanisms applied to bolster basic container isolation: seccomp, AppArmor, SELinux. These are proven and battle-tested but also renowned for how hard they are to manage effectively.
- Where stronger boundaries are needed between containers, micro-VMMs can provide the isolation of a virtual machine but can come with performance penalties.
- There is a third category of sandboxing techniques such as gVisor that fall somewhere between container and virtual machine isolation.

What's right for your applications depends on your risk profile, and your decision may be influenced by the options offered by your public cloud and/or managed solution. You should also consider runtime security tools (which we'll come to in Chapter 15) as they offer a more flexible and dynamic approach to sandboxing that might be more appropriate for your deployments. These might be an alternative to static sandboxing profiles, or they could be combined to provide defense in depth.

Regardless of the container runtime you use and the isolation it enforces, there are ways that a user can easily compromise this isolation. Move on to Chapter 11 to see how.

CHAPTER 11

Breaking Container Isolation

In Chapter 4, you saw how a container is constructed and how it gets a limited view of the machine it is running on. In this chapter, you'll see how easy it is to configure containers to run in such a way that this isolation is effectively broken.

Sometimes you will want to do this deliberately, to achieve something specific such as off-loading networking functionality to a sidecar container. In other circumstances, the ideas discussed in this chapter could be seriously compromising the security of your applications!

To start with, let's talk about what is arguably the most insecure-by-default behavior in the container world: running as root.

Containers Run as Root by Default

Unless your container image specifies a non-root user or you specify a nondefault user when you run a container, by default the container will run as root. And unless you are set up with user namespaces, this is not just root inside the container but also root on the host machine.

This example assumes that you are using the `docker` command provided by Docker. If you have installed `podman`, you may have followed the advice to alias `docker` so that it actually runs `podman` instead. The behavior of `podman` is quite different with regard to root users. I'll come to the differences later in this chapter, but for now be aware that the following example won't work with `podman`.

As a non-root user, run a shell inside an Alpine container using `docker` and check the user identity:

```
$ whoami
liz
$ docker run -it alpine sh
/ $ whoami
root
```

Even though it was a non-root user that ran the `docker` command to create a container, the user identity inside the identity is root. Now let's confirm that this is the same as root on the host by opening a second terminal on the same machine. Inside the container, run a `sleep` command:

```
/ $ sleep 100
```

In the second window, check the identity of this user:

```
$ ps -fC sleep
UID        PID  PPID  C STIME TTY          TIME CMD
root     30619 30557  0 16:44 pts/0    00:00:00 sleep 100
```

This process is owned by the root user from the host's perspective. Root inside the container is root on the host.

If you're using `runc` rather than `docker` to run containers, a similar demo would be less convincing because (aside from rootless containers, which we will discuss shortly) you need to be root on the host to run a container in the first place. This is because only root has sufficient capabilities to create namespaces, generally speaking. In Docker, it's the Docker daemon, running as root, that creates containers on your behalf.

Under Docker, the fact that containers run as root, even when initiated by a non-root user, is a form of privilege escalation. In and of itself, it's not necessarily a problem that the container is running as root, but it does ring alarm bells when thinking about security. If an attacker can escape a container that is running as root, they have full root access to the host, which means free access to everything on the machine, including all the other containers. Do you want to be just one line of defense away from an attacker taking over a host?

Fortunately, it's possible to run containers as non-root users. You can either specify a non-root user ID or use the aforementioned rootless containers. Let's look at both of these options.

Override the User ID

You can override this at runtime by specifying a user ID for the container.

In `runc`, you can do this by modifying the *config.json* file inside the bundle. Change the `process.user.uid`, for example, like this:

```
...
"process": {
        "terminal": true,
        "user": {
                "uid": 5000,
                ...
        }
        ...
}
```

Now the runtime will pick up this user ID and use it for the container process:

```
$ sudo runc run sh
$ whoami
whoami: unknown uid 5000
$ sleep 100
```

Despite using `sudo` to run as root, the user ID for the container is 5000, and you can confirm this from the host:

```
$ ps -fC sleep
UID          PID  PPID  C STIME TTY          TIME CMD
5000     26909 26893  0 16:16 pts/0    00:00:00 sleep 50
```

As you saw in Chapter 6, an OCI-compliant image bundle holds both the root filesystem for an image and the runtime configuration information. This same information is packed into a Docker image. You can override the user config with the `--user` option, like this:

```
$ docker run -it --user 5000 ubuntu bash
I have no name!@b7ca6ec82aa4:/$
```

You can change the user ID that is built into a Docker image with the `USER` command in its Dockerfile. But the vast majority of container images on public repositories are configured to use root because they don't have a `USER` setting. If there is no user ID specified, by default your container will run as root.

No New Privileges

In Chapter 2 you met the `setuid` flag, which allows someone else to assume the identity of that file's owner when they execute it.

Let's take a look at an example of this being used to override the user specified in the following Dockerfile:

```
FROM ubuntu:24.04
RUN cp /usr/bin/bash /tmp/mybash          ❶
RUN chmod 4755 /tmp/mybash                ❷
RUN useradd -ms /tmp/mybash myuser        ❸
USER myuser                               ❹
ENTRYPOINT ["/tmp/mybash"]                ❺
```

❶ Copy the `bash` executable into the `/tmp` directory in the container image.

❷ The `4` in this `chmod` command sets the `setuid` bit on that file, and the `755` allows anyone to read and execute the file and only the owner to write it.

❸ Create a new user called `myuser`.

❹ Make the container run under `myuser`'s identity.

❺ Set the entry point to run the copy of `bash` that has the `setuid` bit.

Running this container image starts a `bash` shell under the `myuser` identity:

```
$ docker run -it nopriv
mybash-5.2$ whoami
myuser
mybash-5.2$ id
uid=1001(myuser) gid=1001(myuser) groups=1001(myuser)
```

If you look at the executable, it's owned by root and has the `setuid` flag:

```
mybash-5.2$ ls -l /tmp/mybash
-rwsr-xr-x 1 root root 1446024 May  8 13:16 /tmp/mybash
```

With that in mind, you might expect the shell to be running as root, until you recall what we learned in Chapter 2 about `bash` checking the user ID and resetting to the original user to avoid this easy route to privilege escalations.

However, `bash` has a `-p` option that allows overriding this check and running as the file owner after all. You can specify this option when running the same container and get a very different result:

```
$ docker run -it nopriv -p
mybash-5.2# whoami
root
mybash-5.2# id
uid=1001(myuser) gid=1001(myuser) euid=0(root) groups=1001(myuser)
```

In the output from `id`, you can see that although the executable was run by `myuser`, it picked up the effective user ID (`euid`) of root. Because this example uses `bash`, you need to specify the `-p` flag to stop it from reverting to the original user, but it required additional work on the part of the developers of `bash` to add that functionality. If the image specified a different executable with the `setuid` flag as the entry point, simply running the container would override the specified `USER` and the container would be running as root.

It would be easy to overlook the `setuid` flag[1] and assume that the `USER` directive would be effective, so to add another layer of defense, you can use the `no-new-privileges` security option at runtime to prevent this kind of privilege escalation:

```
$ docker run -it --security-opt no-new-privileges nopriv -p
myuser@e731e9ac6d6c:/$ whoami
myuser
myuser@e731e9ac6d6c:/$ id
uid=1001(myuser) gid=1001(myuser) groups=1001(myuser)
```

In this example, the user that the container is supposed to run with is specified at build time using the `USER` directive, but the effect is the same if you run the container with the `--user` option.

In Kubernetes you can achieve the same effect by specifying `securityContext.allowPrivilegeEscalation: false` in the container spec for your pods. This is a good default to apply to all applications, unless you have an extremely good reason to allow privilege escalations inside a pod.

Root Requirement Inside Containers

There are many commonly used container images that encapsulate popular software that was originally designed to run directly on servers. Take the Nginx reverse proxy and load balancer, for example; it existed long before Docker became popular, but there are official Nginx container images available on Docker Hub.

At least at the time of writing this book, the standard Nginx container image is configured to run as root by default. If you start an `nginx` container and look at the processes running within it, you will see the master process running as root:

```
$ docker run -d --name nginx nginx
4562ab6630747983e6d9c59d839aef95728b22a48f7aff3ad6b466dd70ebd0fe
$ docker top nginx
UID        PID     PPID    C  STIME   TTY   TIME       CMD
root       24945   24920   0  22:35   ?     00:00:00   nginx: master process nginx
message+   24992   24945   0  22:35   ?     00:00:00   nginx: worker process
message+   24993   24945   0  22:35   ?     00:00:00   nginx: worker process
message+   24994   24945   0  22:35   ?     00:00:00   nginx: worker process
message+   24995   24945   0  22:35   ?     00:00:00   nginx: worker process
```

1 Some image scanners can detect `setuid` executables, as mentioned in Chapter 8.

The main point of this example was to show the process that's running as root, but are you curious about that `message+` username for the worker processes? The first thing to note is that the `+` character means it's a name longer than eight characters, starting with the message. The output from `docker top` is showing usernames as they are defined on the host machine, not how they are defined within the container. If I look at the list of users defined in `/etc/passwd` on my host machine, I see output like this:

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
...
messagebus:x:101:101::/nonexistent:/usr/sbin/nologin
...
lizr:x:501:1000:lizr:/home/lizr.linux:/bin/bash
...
```

The description of the official Nginx image mentions that since version 1.17.0, it has used unprivileged user ID 101 for the worker processes. The user with ID 101 is `messagebus` on this machine, so that corresponds to the output from the `docker top` command.

It makes total sense for the `nginx` code to run as root when it's running on a server. By default it accepts requests on the traditional web port 80. Opening low-numbered ports (under 1024) typically requires the `CAP_NET_BIND_SERVICE` (see Chapter 2), and the simplest way to ensure this is true is to have `nginx` run as the root user. But this requirement makes a lot less sense in a container, where port mapping means that the `nginx` code could listen on any port, with this mapped to port 80 (if required) on the host.

Instead of running as root, another option would be to use `setcap` to add the `CAP_NET_BIND_SERVICE` capability to the `nginx` executable file during the container image build process. The Dockerfile could specify a non-root user, and the executable would still be able to bind to a low-numbered port. Perhaps the Nginx maintainers chose not to do this because `nginx` wouldn't work if capabilities were dropped by the container runtime (for example, `docker --cap-drop=ALL`). It would also require installing the `libcap2-bin` package that provides `setcap`, at least during the build process.

There is another way: you can allow unprivileged users to bind to any port by using `sysctl` to change the kernel setting `net.ipv4.ip_unprivileged_port_start` to 0. Docker now sets this by default, and in Kubernetes you can configure this and other `sysctl` settings for individual workloads using `podSecurityContext`.

Recognizing that running as root is a problem, many vendors provide container images that run as normal, unprivileged users. Nginx has an official unprivileged image that has a few differences from the root-based image, such as using port 8080 by default. This allows it to work even in a container runtime that doesn't support the use of `sysctl` to permit any user to open low-numbered ports.

If Nginx didn't already provide one, it would be relatively straightforward to build an Nginx image that can run as a non-root user (there is a simple example here). For other applications, it can be trickier and may require changes to the code that are more extensive than a few tweaks to the Dockerfile and some configurations. Thankfully, there are organizations, including Bitnami, Chainguard, and the distroless images from Google, that have gone to the trouble of creating and maintaining a series of non-root container images for many popular applications. Many software providers offer unprivileged versions of their distributions (like the one that Nginx provides, as mentioned in the previous paragraph).

Opening low-numbered ports isn't the only reason a piece of software might need to run as root. Another common operation that requires privileges is to install software.

Root for Installing Software

Container images are sometimes configured to run as root is so that they can install software using package managers like yum or apt. It's completely reasonable for this to happen during the build of a container image, but once the packages are installed, a later step in the Dockerfile could easily be a `USER` command so that the image is configured to run under a non-root user ID. Really, you should be doing this as part of a multistage build (see Chapter 7 for more on this).

I strongly recommend you don't allow containers to install software packages at runtime, for several reasons:

- It's inefficient: if you install all the software you need at build time, you do it once only rather than repeating it every time you create a new instance of the container.
- Packages that get installed at runtime haven't been scanned for vulnerabilities (see Chapter 8).
- Related to the fact that the packages haven't been scanned, but arguably worse: it's harder to identify exactly what versions of packages are installed into each different running instance of your containers, so if you do become aware of a vulnerability, you won't know which containers to kill and redeploy.
- Depending on the application, you might be able to run the container as read-only (by using the `--read-only` option in `docker run` or by setting the mount as

readOnly in a Kubernetes pod specification), which would make it harder for an attacker to install code.

- Arguably the most important reason is that adding packages at runtime means you are not treating them as immutable. See "Immutable Containers" on page 111 for more about the security advantages of immutable containers.

Privileges for eBPF and Kernel Modules

Another thing that requires significant privileges is modifying the kernel, which you can do at runtime using Kernel modules or with eBPF. Both these approaches allow for code to be dynamically loaded into the Linux kernel at runtime to extend its functionality. As you know, all containers on a host share the same kernel, so adding, say, observability or security tools into the kernel means they can immediately see and affect all the containerized applications on that host.

The problem with kernel modules is that they don't have the same extensive community testing and field hardening as the rest of the Linux kernel. Users are, quite reasonably, wary of installing kernel modules because of the risk of a bug that causes a crash and brings down the whole machine.

In contrast, eBPF programs are guaranteed to be safe to run in the kernel thanks to the eBPF verification process, which analyzes every possible execution path through a program and refuses to load it into the kernel if there is any possibility that it might crash. This makes eBPF a very powerful platform for building infrastructure tooling, especially in containerized environments.

The term *eBPF* used to stand for extended Berkeley Packet Filter, but this was confusing since eBPF can do so much more than filtering packets, so the eBPF community decided that it's best to consider it a standalone term rather than an acronym.

To delve into eBPF, you might like to start with the slides from my talk on unleashing the kernel with eBPF. The eBPF documentary is an engaging watch that tells the story of how eBPF came to be. You'll find a lot more information and resources on eBPF and on Brendan Gregg's website, and if you are inspired to try writing eBPF programs yourself, you could check out my book *Learning eBPF*, published by O'Reilly.

As you would expect, it takes significant privileges to load kernel modules or eBPF programs into the kernel. Kernel modules need the `CAP_SYS_MODULE` capability, and

eBPF needs at least `CAP_BPF`[2] plus some other capabilities that might be necessary depending on exactly what the eBPF program(s) do, such as:

- `CAP_NET_ADMIN` to attach into the networking stack
- `CAP_PERFMON` for attaching to kernel hooks related to performance monitoring
- `CAP_SYS_RESOURCE` to raise limits required for eBPF map memory
- `CAP_SYS_ADMIN` for mounting filesystems
- `CAP_SYS_PTRACE` for tracing user processes

Kernel-based tools may very likely also need certain filesystems from the host mounted into the container (for example, `/lib/modules` for kernel modules or `/sys/fs/bpf` for eBPF).

The bottom line is that if you want to run eBPF-based (or kernel module) tools, you will have to allow them additional privileges. The benefits may well outweigh the risks as it opens up the option of eBPF-based runtime security tooling, which we'll discuss in Chapter 15.

For your own application code, use a non-root user whenever you can, or run with user namespaces (as seen in "User Namespace" on page 47), so that root inside the container is not the same as root on the host. One practical way to run with user namespaces, if your system supports it, is to use *rootless containers*.

Rootless Containers

If you worked through the examples in Chapter 4, you'll know that you need root privileges to perform some of the actions that go into creating a container. This is typically seen as a no-go in traditional shared machine environments, where multiple users can log in to the same machine. An example is a university system, where students and staff often have accounts on a shared machine or cluster of machines. System administrators quite rightly object to giving root privileges to a user so that they can create containers, as that would also allow them to do anything (deliberately or accidentally) to any other user's code or data.

The Rootless Containers initiative drove work including the kernel changes required to allow non-root users to run containers. There is now full support for rootless mode in Docker and containerd. The `podman` container implementation has supported rootless containers for a long time, and it doesn't use a privileged daemon process in the way that Docker does. This is why the examples at the start of this chapter behave differently if you have `docker` aliased to `podman`.

2 From Linux 5.8 onward. Before that, loading eBPF programs was covered by `CAP_SYS_ADMIN`.

In Kubernetes there is a differentiation between container workloads running as non-root and a non-root user being permitted to run Kubernetes node components. Running workloads as non-root is achieved using `pod SecurityContext`; at the time of writing, support for rootless Kubernetes components is still marked as Alpha, though it has been available in this state since Kubernetes 1.22 (with the latest version being 1.33!).

In a Docker system, even if you're not using rootless containers, you don't actually need to be root to run a container, but you need to be a member of the `docker` group that has permissions to send commands over the Docker socket to the Docker daemon. It's worth being aware that being able to do this *is equivalent to having root on the host*. Any member of that group can start a container, and as you are now aware, by default they will be running as root. If they were to mount the host's root directory with a command like `docker run -v /:/host <image>`, they would have full access to the host's root filesystem too.

Rootless containers make use of the user namespace feature that you saw in "User Namespace" on page 47. A normal non-root user ID on the host can be mapped to root inside the container. If a container escape occurs somehow, the attacker doesn't automatically have root privileges, so this is a significant security enhancement.

Read more about root inside and outside a `podman` container in Scott McCarty's blog post.

However, rootless containers aren't a panacea. Not every image that runs successfully as root in a normal container will behave the same in a rootless container, even though it appears to be running as root from the container's perspective.

As the documentation for user namespaces states, they isolate not just user and group IDs but also other attributes, including capabilities. In other words, you can add or drop capabilities for a process in a user namespace, and they apply only inside that namespace. So if you add a capability for a rootless container, it applies only in that container but not if the container is supposed to have access to other host resources.

Dan Walsh wrote a blog post with some good examples of this. One of them is about binding to low-numbered ports, which, as you saw in the discussion of Nginx earlier, requires `CAP_NET_BIND_SERVICE`. If you run a normal container with `CAP_NET_BIND_SERVICE` (which it would likely have by default if running as root) and sharing the host's network namespace, it could bind to any host port. A rootless container, also with `CAP_NET_BIND_SERVICE` and sharing the host's network, would not be able to bind to low-numbered ports because the capability doesn't apply outside the container's user namespace.

By and large, the namespacing of capabilities is a good thing, as it allows containerized processes to seemingly run as root but without the ability to do things that would require capabilities at the system level, like changing the time or rebooting the machine. The vast majority of applications that can run in a normal container will also run successfully in a rootless container.

When using rootless containers, although the process appears from the container's perspective to be running as root, from the host's perspective, it's a regular user. One interesting consequence of this is that the rootless container doesn't necessarily have the same file access permissions as it would have without the user remapping. To get around this, the filesystem needs to have support to remap file ownership and group ownership within the user namespace. (Not all filesystems have this support at the time of writing.)

Running as root inside a container isn't exactly a problem in and of itself, as the attacker still needs to find a way to escape the container. From time to time, container escape vulnerabilities have been found in container runtimes and in the kernel, and they probably will continue to be found. But a vulnerability isn't the only way that container escape can be made possible. Later in this chapter, you'll see ways in which risky container configurations can make it easy to escape the container, with no vulnerability required. Combine these bad configurations with containers running as root, and you have a recipe for disaster.

With user ID overrides and rootless containers, there are options for avoiding running containers as the root user. However you achieve it, you should try to avoid containers running as root.

The --privileged Flag and Capabilities

Docker and other container runtimes let you specify a `--privileged` option when you run a container. Andrew Martin has called it "the most dangerous flag in the history of computing," with good reason: it's incredibly powerful, and it's widely misunderstood.

It's often thought that `--privileged` equates to running a container as root, but you already know that containers run as root by default. So what other privileges could this flag be bestowing on the container?

The answer is that, although in Docker the process runs under the root user ID by default, a large group of root's normal Linux capabilities are not granted as a matter of course. (If you need a refresher on what capabilities are, skip back to "Linux Capabilities" on page 21.)

It's easy enough to see the capabilities that a container is granted by using the `capsh` utility. For simplicity, in these examples I am only showing the current set of capabilities and the bounding set and omitting some other output for clarity.

To set the scene, let's look at the output when all capabilities are dropped and then one is added:

```
$ docker run --rm -it --cap-drop=all --cap-add=CAP_NET_BIND_SERVICE
alpine sh -c 'apk add -U libcap; capsh --print'

Current: cap_net_bind_service=ep
Bounding set =cap_net_bind_service
```

These examples do something I advised against earlier: they install the `libcap` package at runtime. Don't do this in production containers!

This container has, and can only ever have, one capability. Now let's see the default set of capabilities granted by Docker:

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

Current: cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,
cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,
cap_mknod,cap_audit_write,cap_setfcap=ep
Bounding set =cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,
cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,
cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap
```

That's a substantial set of capabilities granted by default. The precise set of capabilities granted without the privileged flag is implementation dependent. The OCI defines a default set, granted by `runc`.

Finally, let's see what happens when we add the `--privileged` flag:

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

Current: =ep
Bounding set =cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,
cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,
cap_net_bind_service,cap_net_broadcast,cap_net_admin,cap_net_raw,cap_ipc_lock,
cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,
cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,
cap_sys_time,cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write,
cap_audit_control,cap_setfcap,cap_mac_override,cap_mac_admin,cap_syslog,
cap_wake_alarm,cap_block_suspend,cap_audit_read,cap_perfmon,cap_bpf,
cap_checkpoint_restore
```

As you might remember from Chapter 2, several Linux tools have changed to showing an empty set of capabilities if running as root with a full set of (implicitly granted) capabilities. That's why the `Current` set reported here is empty. The bounding set explicitly mentions `CAP_SYS_ADMIN`, and this single capability flag grants access to a huge range of privileged activities, including things like manipulating namespaces and mounting filesystems. Similarly, `CAP_NET_ADMIN` for manipulating the network stack and `CAP_BPF` for loading eBPF programs are included with this `--privileged` flag.

Eric Chiang wrote a blog post about the dangers of `--privileged` in which he shows an example of breaking out of a container onto the host filesystem by mounting a device from `/dev` into the container filesystem.

Docker introduced the `--privileged` flag to enable "Docker in Docker." This is used widely for build tools and CI/CD systems running as containers, which need access to the Docker daemon in order to use Docker to build container images. But as this blog post describes, you should use Docker in Docker, and the `--privileged` flag in general, with caution.

A more subtle reason why the `--privileged` flag is so dangerous is that because people often think that it's needed to give the container root privileges, they also believe the converse: that a container running without this flag is not a root process. Please refer to "Containers Run as Root by Default" on page 143 if you're not yet convinced about this.

Even if you have reasons to run containers with the `--privileged` flag, I would advise controls or at least an audit to ensure that only those containers that really need it are granted the flag. Consider specifying individual capabilities instead.

Let's use `capable`, an eBPF-based command-line tool installed as part of the `libbpf-tools` package, to trace out `cap_capable` events and show the capabilities that a given container requests from the kernel. Run `capable` in one terminal and start an `nginx` container in a second:

```
$ docker run -it --rm nginx
```

The output from `capable` will include output like this (some omitted for clarity):

```
TIME     UID  PID    COMM    CAP   NAME                AUDIT   VERDICT
18:06:24 0    23745  nginx   21    CAP_SYS_ADMIN       1       allow
18:06:24 0    23745  nginx   21    CAP_SYS_ADMIN       1       allow
18:06:24 0    23745  nginx   2     CAP_DAC_READ_SEARCH 1       allow
18:06:24 0    23745  nginx   2     CAP_DAC_READ_SEARCH 1       allow
18:06:24 0    23745  nginx   1     CAP_DAC_OVERRIDE    1       allow
18:06:24 0    23745  nginx   21    CAP_SYS_ADMIN       1       allow
```

```
18:06:24 0    23792  nginx   6     CAP_SETGID          1       allow
18:06:24 0    23792  nginx   7     CAP_SETUID          1       allow
```

Once you know which capabilities your container needs, you can follow the principle of least privilege and specify at runtime the precise set that should be granted. The recommended approach is to drop all capabilities and then add back the necessary ones as follows:

```
$ docker run --cap-drop=all --cap-add=<cap1> --cap-add=<cap2> <image> ...
```

Now you are warned of the dangers of the `--privileged` flag and the opportunity to shrink-wrap capabilities for a container. Let's look at another way that container isolation can be sidestepped: mounting sensitive directories from the host.

Mounting Sensitive Directories

Using the `-v` option, you can mount a host directory into a container so that it is available from the container. And there is nothing to stop you from mounting the host's root directory into a container, like this:

```
$ touch /ROOT_FOR_HOST
$ docker run -it -v /:/hostroot ubuntu bash
root@91083a4eca7d:/$ ls /
bin   dev  home      lib    media  opt   root  sbin  sys  usr
boot  etc  hostroot  lib64  mnt    proc  run   srv   tmp  var
root@91083a4eca7d:/$ ls /hostroot/
ROOT_FOR_HOST  etc             lib         media  root  srv  vagrant
bin            home            lib64       mnt    run   sys  var
...
```

Because this example uses the default `ubuntu` container image, it runs as root. An attacker who compromises this container is also root on the host, with full access to the entire host filesystem.

Mounting the entire filesystem is a pathological example, but there are plenty of other examples that range in their subtlety, such as the following:

- Mounting `/etc` would permit modifying the host's `/etc/shadow` file from within the container, or messing with `cron` jobs, `init`, or `systemd`.
- Mounting `/bin` or similar directories such as `/usr/bin` or `/usr/sbin` would allow the container to write executables into the host directory—including overwriting existing executables.
- Mounting host log directories into a container could enable an attacker to modify the logs to erase traces of their dastardly deeds on that host.
- In a Kubernetes environment, mounting `/var/log` can give access to the entire host filesystem to any user who has access to `kubectl logs`. This is because container log files are symlinks from `/var/log` to elsewhere in the filesystem, but

there is nothing to stop the container from pointing the symlink at any other file. See this blog post for more on this interesting escape. This escape requires running as root, so you can mitigate it by using `runAsNonRoot`. You should also never allow writable `hostPath` mounts to `/var/log`.

- In Kubernetes you can mark a volume mount as `readOnly`, but beware! If there are submounts within that directory on the host, they might in fact be writable. To ensure that any submounts are also read-only within the container, set the `recursiveReadOnly` setting to enabled. This is explained further by Akihiro Suda in a Kubernetes blog post. Docker attempts to do this by default, according to the documentation.

These file locations are vulnerable only if the container runs as root (or another privileged user). If you apply good practices such as running as non-root or using namespaces, then standard Linux file permissions will help to protect most, if not all, sensitive locations.

Mounting the Docker Socket

In a Docker environment, there is a Docker daemon process that essentially does all the work. When you run the `docker` command-line utility, this sends instructions to the daemon over the Docker socket that lives at `/var/run/docker.sock`. Any entity that can write to that socket can also send instructions to the Docker daemon. The daemon runs as root and will happily build and run any software of your choosing on your behalf, including—as you have seen—running a container as root on the host. Thus, access to the Docker socket is effectively the equivalent of root access on the host.

One common use of mounting the Docker socket is in CI tools like Jenkins, where the socket is needed specifically for sending instructions to Docker to run image builds as part of your pipeline. This is a legitimate thing to do, but it does create a potential soft underbelly that an attacker can pierce. A user who can modify a Jenkinsfile can get Docker to run commands, including those that could give the user root access to the underlying cluster. For this reason, it's exceptionally bad practice to run a CI/CD pipeline that mounts a Docker socket in a production cluster.

Sharing Namespaces Between a Container and Its Host

On occasion, there might be reasons to have a container use some of the same namespaces as its host. For example, suppose you want to run a process in a Docker container but give it access to the process information from the host. In Docker, you can request this with the `--pid=host` parameter.

Recall that containerized processes are all visible from the host; thus, sharing the process namespace to a container lets that container see the other containerized processes too. The following example starts by running a long-running `sleep` inside one container; that process can be observed from another container started with `--pid=host`:

```
$ docker run --name sleep --rm -d alpine sleep 1000
fa19f51fe07fca8d60454cf8ee32b7e8b7b60b73128e13f6a01751c601280110
$ docker run --pid=host --name alpine --rm -it alpine sh
$ ps | grep sleep
30575 root      0:00 sleep 1000
30738 root      0:00 grep sleep
```

What's even more exciting is that running `kill -9 <pid>` from the second container can kill the `sleep` process in the first!

You have seen several ways in which sharing namespaces or volumes between containers, or between a container and its host, can weaken the container's isolation and compromise security, but it's by no means *always* a bad idea to share information with containers. To conclude this chapter, let's look at sidecar containers and then debug containers, which are common patterns for sharing namespaces between containers for good reasons.

Sidecar Containers

A *sidecar container* is deliberately given access to one or more of an application container's namespaces so that it can off-load functionality from that application. In a microservice architecture, you might have functionality that you want to reuse in all your microservices, and a common pattern is to package that functionality into sidecar container images so that it can easily be reused.

Before sidecars, you could implement reusable code in a library. This is easy to import into applications written in the same language, but it is more complex if your teams write apps in more than one language.[3] With a sidecar, because it's containerized, the functionality can be written in any language, as illustrated in Figure 11-1, and it can also be used to instrument containers provided by a third party.

3 Libraries written in one language might also be reused from another, using language-specific bindings or Foreign Function Interfaces.

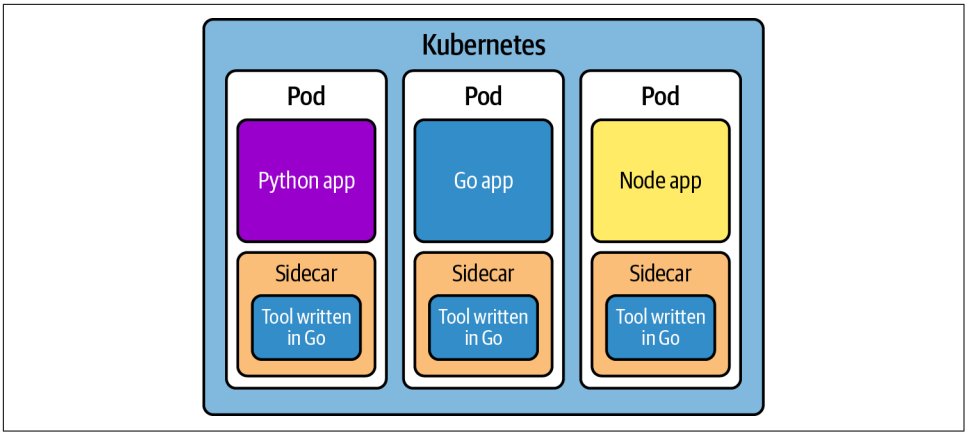


Figure 11-1. Sidecar containers allow applications to use infrastructure tooling written in a different language.

Here are a few common example use cases for sidecars:

- Service mesh sidecars take over the networking functionality on behalf of the application container. The service mesh can, for example, ensure that all network connections use mutual TLS. Off-loading this functionality to a sidecar means that so long as a container is deployed with the sidecar, it will set up secure TLS connections; there is no need for each application team to spend time reimplementing and testing this feature in every application. (Further discussion of service meshes is coming up in Chapter 12—see "Service Mesh" on page 179.)
- Observability sidecars can set up destinations and configurations for logging, tracing, and gathering metrics. For example, Prometheus and OpenTelemetry support sidecars for exporting observability data.
- Security sidecars can police the executables and network connections that are permitted within an application container. (For example, see my blog post from some years ago about securing AWS Fargate containers using Aqua's MicroEnforcer in sidecar containers.)

This is just a selection of applications for sidecar containers, which legitimately share namespaces with application containers.

Deploying Sidecars

In Kubernetes, the sidecar container pattern is really a first-class citizen: a pod specification can include multiple containers, which share a common network namespace, can share volumes, and are scheduled and restarted together. Many sidecar-based tools, like Istio, Linkerd, and Vault, modify the application pod specification as it's deployed, using a MutatingAdmissionWebhook. This intercepts pod creation requests and adds the sidecar container definition into the pod specification.

AWS ECS and Fargate also support sidecars natively, as they are configured using Task Definitions that can include multiple containers. They will be created with a network namespace shared between them, and they can also share storage volumes.

Docker doesn't really have native support for sidecars, though you can achieve something similar using Docker Compose to define multiple containers that share a network namespace.

Sidecar Limitations

Sidecars have been a useful pattern for injecting functionality, but they come with their fair share of drawbacks. It's pretty clear that in this model, the number of containers being deployed is increased—one sidecar container per application container—and this consumes additional resources. This resource consumption is compounded by the fact that we're deliberately isolating containers or pods from each other, so it's intentionally harder to share information between them; sidecars will need their own copies of common configuration information, routing tables, and the like.

The life cycle of the sidecar is tightly coupled to the workload container, and if the sidecar needs an upgrade, then the workload container has to be restarted. There can be issues related to the order in which workload and sidecar containers start and initialize relative to each other, making sidecar tooling harder to operate in production. These limitations have pushed the industry toward *sidecarless* models for infrastructure tools, often using eBPF to instrument a whole (virtual) machine at the kernel level. I'll come back to eBPF-based tooling, particularly with regard to container security, in Chapter 15.

Debug Containers

In Kubernetes, *debug containers* are ephemeral containers that can be attached to a running pod using the `kubectl debug` command, and they are used for troubleshooting or diagnostic purposes.

This can be very useful, but as you've probably already guessed, they provide a route for executing arbitrary code, so they can easily be misused for malicious purposes. They should be treated as high-risk, privileged tools. If you permit them at all in production clusters, make sure that their use is restricted to trusted users using appropriate access controls, and ensure thorough audit logging. It's a good idea to enforce their ephemeral nature by cleaning them up automatically after a period of time.

Summary

This chapter covered several ways in which the isolation that's normally provided by containers can be compromised through bad configuration.

All the configuration options are provided for good reasons. For example, mounting host directories into a container can be extremely useful, and sometimes you do need the option to run a container as root or even with the additional capabilities provided by the `--privileged` flag. However, if you're concerned about security, you'll want to minimize the extent to which these potentially dangerous configurations are used and employ tools to spot when they are happening.

If you're running in any kind of multitenant environment, you should be even more attentive to containers with these potentially dangerous configurations. Any `--privileged` container will have full access to any other container on the same host, regardless of relatively superficial controls such as whether they are running in the same Kubernetes namespace.

In "Sidecar Containers" on page 158, I mentioned service meshes, which can off-load some networking functionality. Now seems like a good time to talk about container networking.

CHAPTER 12

Container Network Security

Every external attack reaches your deployment across a network, so it's important to understand something about networking in order to consider how to secure your applications and data. This isn't going to be a comprehensive treatment of everything to do with networking (that would make this book a lot longer!), but it should give you the essentials of a sensible mental model you can use to think about network security in your container deployment.

I'll start with an overview of container firewalling and microsegmentation, which can provide a much more granular approach to network security than traditional firewalling approaches.

Then there is a review of the seven-layer networking model, which is worth knowing about so that you can understand the level a network security feature acts at. With this in place, we will discuss how container network security is implemented using a couple of different approaches. We will discuss Kubernetes network policies at different layers of the network model and look at some best practices for network policy rules.

Container Firewalls and Microsegmentation

Containers often go hand in hand with microservice architectures, where an application is broken into small components that can be deployed independently of each other. This can offer real benefits from a security perspective, because it's much easier to define what normal behavior looks like in a small component. A given container probably has to communicate with only a limited set of other containers, and only a subset of containers need contact with the outside world.

For example, consider an ecommerce application broken into microservices. One of these microservices could handle product search requests; it receives search requests

from end users and looks up their search queries in a product database. The containers that make up this service don't have any reason to communicate with, say, the payment gateway. Figure 12-1 illustrates this example.

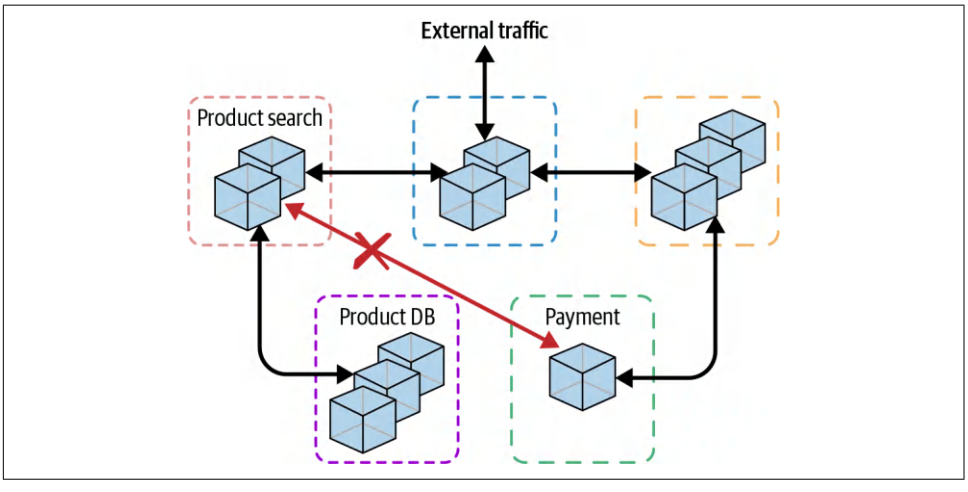


Figure 12-1. Microsegmentation

A container firewall can restrict the traffic that flows to and from a set of containers. In an orchestrator like Kubernetes, the term *container firewall* is rarely used; instead, you'll hear about *microsegmentation*, where network policies are defined in terms of workloads and services rather than between individual containers. In both cases, the principle is to restrict network traffic so that it can flow to and from approved destinations only.

The open source distribution of Kubernetes doesn't come with networking capabilities built in; instead, it has an interface called the Container Network Interface (CNI). There are several projects and products that can fulfill the networking functionality required on the other side of that interface. These are known as CNI plug-ins, or colloquially, they are referred to as simply CNIs. Kubernetes has a native concept of Network Policies for restricting traffic between workloads, though as you'll see later in this chapter, *CNIs are not actually required to enforce these policies*. If you want them to work, you need to pick an appropriate CNI!

Docker doesn't really have the concept of network policies built in, beyond defining separate (virtual) networks to isolate containers from each other, though there are commercial solutions that target container firewalling capabilities outside Kubernetes.

Container network security tools, like their traditional firewall counterparts, will typically report on attempted connections outside the rules, providing useful forensics for investigation into possible attacks.

They can be used in conjunction with other network security tools that you may have come across in traditional deployments as well. For example:

- It's common to deploy your container environment in a virtual private cloud (VPC), which isolates your hosts from the rest of the world.
- You can use a firewall around the entire cluster to control traffic in and out.
- You can use API firewalls (also known as web application firewalls [WAFs]) to restrict traffic at Layer 7.
- You can encrypt network traffic so that it can't be intercepted. This is discussed in Chapter 13.

None of these approaches is unique to containerized deployments—they all work in traditional deployments too. Combining them with container-aware security gives additional defense in depth.

Before we look at how container firewalling and network security policies are implemented, let's review the seven-layer networking model and follow the path of an IP packet through a network.

OSI Networking Model

The Open Systems Interconnection (OSI) networking model was published in 1984 and defines a layered model of networking that is still commonly referenced today, although as you can see from Figure 12-2, the seven layers don't all have an equivalent in IP-based networks.

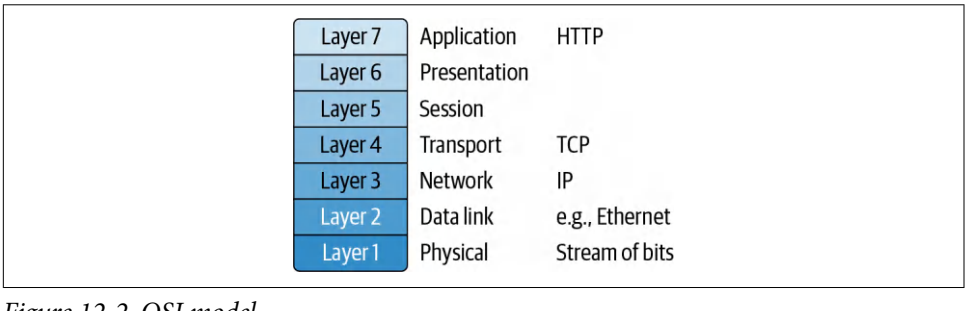


Figure 12-2. OSI model

These are the layers that matter in an IP-based network:

- Layer 7 is the application layer. If you think about an application making a web request or sending a RESTful API request, you are picturing something that happens at Layer 7. The request is typically addressed by a URL, and to get the request to its destination, the domain name gets mapped to an Internet Protocol

(IP) address using a protocol called Domain Name Resolution that is offered by a Domain Name Service (DNS).

- Layer 4 is the transport layer, typically TCP or UDP packets. This is the layer at which port numbers apply.
- Layer 3 is the layer at which IP packets travel and at which IP routers operate. An IP network has a set of IP addresses assigned to it, and when a container joins the network, it gets assigned one of those IP addresses. For the purposes of this chapter, it doesn't matter whether the IP network uses IP v4 or IP v6—you can consider that to be an implementation detail.
- At Layer 2, data packets are addressed to endpoints connected to a physical or virtual interface (which I'll discuss in a moment). There are several Layer 2 protocols, including Ethernet, Wi-Fi, and, if you cast your mind back into history, Token Ring. (Wi-Fi is slightly confusing here since it covers both Layer 2 and Layer 1.) I'll only cover Ethernet in this chapter since that is predominantly what's used for Layer 2 container networking. At Layer 2, interfaces are addressed using MAC[1] addresses.
- Layer 1 is called the physical layer, although to keep us all on our toes, interfaces at Layer 1 can be virtual. A physical machine will have a physical network device attached to some kind of cable or wireless transmitter. Cast your mind back to "Enter the VMM" on page 61, and you will recall that a VMM gives a guest kernel access to virtual devices that map to these physical devices. When you get a network interface on, say, an EC2 instance in AWS, you're getting access to one of these virtual interfaces. Container network interfaces are commonly virtual at Layer 1 as well. Whenever a container joins a network, it has a Layer 1 interface to that network.

For more depth or an alternative explanation, you might like to refer to Cloudflare's description of the OSI networking model.

Let's see what happens at these different layers when an application wants to send a message.

1 In the networking context, MAC stands for Media Access Control, though if you're like me you just think of this term as referring to a hardware address. Don't confuse this with the security acronym Mandatory Access Control.

Sending an IP Packet

Imagine an application that wants to send a request to a destination URL. Since this is the application, it stands to reason from the preceding definition that this is happening at Layer 7.

The first step is a DNS lookup to find the IP address that corresponds to the host name in that URL. DNS could be defined locally (as in the */etc/hosts* file on your laptop), or it could be resolved by making a DNS request to a remote service at a configured IP address. (If the application already knows the IP address it wants to send a message to, rather than using a URL, the DNS step is skipped.)

Once the networking stack knows which destination IP address it needs to send the packet to, the next step is a Layer 3 routing decision, which consists of two parts:

1. To reach a given destination, there might be multiple hops in the IP network. Given the destination IP address, what is the IP address of the next hop?
2. What interface corresponds to this next-hop IP address?

Next, the packet has to be converted to Ethernet frames, and the next-hop IP address has to be mapped to the corresponding MAC address. This relies on the Address Resolution Protocol (ARP), which maps IP addresses to MAC addresses. If the network stack doesn't already know the MAC address for the next-hop IP address (which could already be held in an ARP cache), then it uses ARP to find out.

Once the network stack has the next-hop MAC address, the message can be sent out over the interface. Depending on the network implementation, this could be a point-to-point connection, or the interface may be connected to a *bridge*.

The easiest way to understand a bridge is to imagine a physical device with a number of Ethernet cables plugged in. The other end of each cable connects to the network card on a device—a computer, say. Every physical network card has a unique MAC address hardcoded into it by the manufacturer. The bridge learns the MAC address at the far end of each of the cables plugged into its interface. All the devices connected to the bridge can send packets to each other through the bridge. In container networking, the bridge is implemented in software rather than being a separate physical device, and the Ethernet cables are replaced by virtual Ethernet interfaces. So the message arrives at the bridge, which uses the next-hop MAC address to decide which interface to forward it on.

When the message arrives at the other end of the Ethernet connection, the IP packet is extracted and passed back up to Layer 3. Data is encapsulated with headers at different layers in the networking stack, as shown in Figure 12-3.

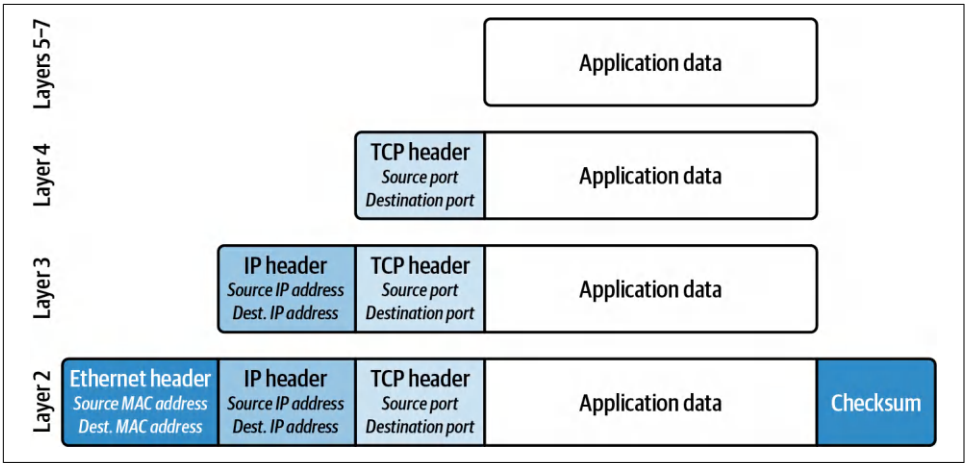


Figure 12-3. Networking headers

If this is the packet's final destination, then it gets passed up to the receiving application. However, this might just be the next hop for the packet, and in that case, the networking stack needs to make another routing decision about where to send the packet next.

This explanation glosses over a few details (such as how ARP works or how routing decides which is the next-hop IP address), but it should be sufficient for our purposes of thinking about container networking.

IP Addresses for Containers

"Sending an IP Packet" on page 167 talks about getting traffic to reach a destination based on its IP address. Containers can share the IP address of their host, or they can each have their own network stack running in their own network namespace. You saw how network namespaces are set up in Chapter 4. Since there's a good chance you are running containers under Kubernetes, let's explore how IP addresses are used in Kubernetes.

In Kubernetes, each pod has its own IP address. If the pod includes more than one container, you can infer that each container shares the same IP address. This is achieved by having all containers in a pod share the same network namespace. Every node is configured to use a range of addresses (a CIDR block), and when a pod is scheduled to a node, it gets assigned one of the addresses from that range.

It's not strictly true that nodes are always assigned a range of addresses up front. For example, on AWS, a pluggable IP address management module dynamically assigns pod IP addresses from the range associated with the underlying VPC.

Kubernetes requires that pods in a cluster can connect directly to each other without any network address translation (NAT) between them. In other circumstances, NAT allows IP addresses to be mapped so that one entity sees a destination as being at a certain IP address, even though that isn't the actual address of the destination device. (This is one reason why IPv4 has been in use for much longer than originally predicted. Although the number of IP-addressable devices has far outstripped available addresses in the IPv4 address space, NAT means we can reuse the vast majority of IP addresses within private networks.) In Kubernetes, network security policies and segmentation might prevent a pod from being able to communicate with another pod, but if the two pods can communicate, they see each other's IP addresses transparently without any NAT mapping. There can, however, still be NAT between pods and the outside world.

Kubernetes services are a form of NAT. A Kubernetes service is a resource in its own right, and it gets assigned an IP address of its own. It's just an IP address, though—a service doesn't have any interfaces, and it doesn't actually listen for or send traffic. The address is just used for routing purposes. The service is associated with some number of pods that actually do the work of the service, so packets sent to the service IP address need to be forwarded to one of these pods. We'll shortly see how this is done.

In traditional networking you might come across the term *PAT* (port address translation) as well as *NAT*. In the cloud native and container world, you're more likely to hear about *port mapping*, where a host port number is translated to a different port number inside a container.

Network Isolation

It's worth explicitly pointing out that two components can communicate with each other only if they are connected to the same network. In traditional host-based environments, you might have isolated different applications from each other by having separate VLANs for each one.

In the container world, Docker made it easy to set up multiple networks using the `docker network` command, but it's not something that fits naturally in the Kubernetes model where every pod can (modulo network policies and security tools) access every other pod by IP address.

It's also worth noting that in Kubernetes, the control components run in pods and are all connected to the same network as the application pods. If you come from a telecommunications background, this may surprise you, since a lot of effort was put into separating the control plane from the data plane in phone networks, primarily for security reasons.

Instead, Kubernetes container networking can be enforced using network policies that act at Layer 3/4 and, depending on your network plug-in, Layer 7. I'll describe the traditional approach to these first, and then describe why eBPF, which you have come across in Chapter 9, makes for a better implementation.

Layer 3/4 Routing and Rules

As you already know, routing at Layer 3 is concerned about deciding the next hop for an IP packet. This decision is based on a set of rules about which addresses are reached over which interface. But this is just a subset of things you can do with Layer 3 rules: there are also some fun things that can go on at this level to drop packets or manipulate IP addresses, for example, to implement load balancing, NAT, firewalls, and network security policies. Rules can also act at Layer 4 to take into account the port number.

Traditionally, these rules rely on a kernel feature called `netfilter`, though nowadays most deployments are using or moving toward eBPF-based implementations, for reasons you'll learn about later in this chapter.

`netfilter` is a packet-filtering framework that was first introduced into the Linux kernel in version 2.4. It uses a set of rules that define what to do with a packet based on its source and destination addresses. There are a few different ways that `netfilter` rules can be configured in user space, the most common of which is `iptables`.

iptables

The `iptables` tool is the traditional way of configuring IP packet–handling rules that are dealt with in the kernel using `netfilter`. There are several different table types. The two most interesting types in the context of container networking are `filter` and `nat: filter` is for deciding whether to drop or forward packets, and `nat` is for translating addresses.

As a root user, you can see the current set of rules of a particular type by running `iptables -t <table type> -L`.

The `netfilter` rules that you can set up with `iptables` can be useful for security purposes. Traditionally, container firewall solutions, as well as older Kubernetes network plug-ins, make use of `iptables` to set up network policy rules that are implemented

using `netfilter` rules. I'll come back to this in "Network Policies" on page 174. First, let's delve into the rules that get set up with `iptables`.

As mentioned earlier, in Kubernetes, a service is an abstraction that maps a service name to a set of pods. There is a component called *kube-proxy* on each node that manages the load-balancing of traffic across all the pods that provide a service, by default using `iptables` rules. A client might send a request to a service domain name, which gets resolved to an IP address using DNS. When a packet destined for that service IP address arrives, there is an `iptables` rule that matches the destination address and swaps the destination address for that of one of the corresponding pods.

If the set of pods behind a service changes, the `iptables` rules get rewritten on each host accordingly. Unfortunately for `iptables` fans, pods come and go dynamically in a Kubernetes environment, and this rewriting of rules turned out to be a bottleneck at scale, ultimately leading to the adoption of newer technologies like eBPF. That said, seeing some `iptables` rules can help you to understand the logic behind both kube-proxy load-balancing and network policies, so let's take a look.

It's easy enough to see the `iptables` rules for a service. Let's take a Kubernetes cluster with a two-replica deployment of `nginx`, behind a service (I have removed some of the output fields for clarity):

```
$ kubectl get svc,pods -o wide
NAME                 TYPE        CLUSTER-IP      PORT(S)
service/kubernetes   ClusterIP   10.96.0.1       443/TCP
service/my-nginx     NodePort    10.100.132.10   8080:32144/TCP

NAME                              READY   STATUS    IP
pod/my-nginx-75897978cd-n5rdv     1/1     Running   10.32.0.4
pod/my-nginx-75897978cd-ncnfk     1/1     Running   10.32.0.3
```

You can list the current address translation rules with `iptables -t nat -L`. There will likely be a lot of output, but it's not too hard to find the interesting parts that correspond to this `nginx` service. First, here is the rule that corresponds to the `my-nginx` service running on IP address 10.100.132.10. You can see that it's part of a chain called "KUBE-SERVICES," which makes sense since it relates to a service:

```
Chain KUBE-SERVICES (2 references)
target                    prot opt source     destination
...
KUBE-SVC-SV7AMNAGZFKZEMQ4  tcp  --  anywhere   10.100.132.10   /* default/my-
nginx:http cluster IP */ tcp dpt:http-alt
...
```

The rule specifies a target chain, which appears later in the `iptables` rules:

```
Chain KUBE-SVC-SV7AMNAGZFKZEMQ4 (2 references)
target                    prot opt source     destination
KUBE-SEP-XZGVVMRRSKK6PWWN  all  --  anywhere   anywhere      statistic mode
```

```
random probability 0.50000000000
KUBE-SEP-PUXUHBP3DTPPX72C  all  --  anywhere   anywhere
```

It seems reasonable to infer from the "random probability 0.5" part of this that traffic is being split between these two targets with equal probability. This makes a lot of sense when you see that these targets have rules that correspond to the IP addresses of the pods (10.32.0.3 and 10.32.0.4):

```
Chain KUBE-SEP-XZGVVMRRSKK6PWWN (1 references)
target          prot opt source               destination
KUBE-MARK-MASQ  all  --  10.32.0.3            anywhere
DNAT            tcp  --  anywhere             anywhere        tcp to:10.32.0.3:80
...
Chain KUBE-SEP-PUXUHBP3DTPPX72C (1 references)
target          prot opt source               destination
KUBE-MARK-MASQ  all  --  10.32.0.4            anywhere
DNAT            tcp  --  anywhere             anywhere        tcp to:10.32.0.4:80
```

The problem with `iptables` is that, as mentioned previously, performance drops off as the system writes and rewrites complex sets of rules. In fact, kube-proxy's use of `iptables` was identified as a performance bottleneck when running Kubernetes at scale. This blog post points out that 2,000 services with 10 pods each results in an additional 20,000 `iptables` rules on every node.

One technology that might have helped address this was IPVS, which was more performant for kube-proxy's case (see Project Calico's performance comparison of `iptables` and IPVS). But IPVS never really took off for a number of reasons: it requires `ip_vs` kernel modules (which can be brittle), it wasn't adopted by public cloud providers, and the ecosystem of tooling that many operators grew familiar with around `iptables` didn't work with it.

`nftables` is another more modern approach, and there is a project to use it for a more performant version of kube-proxy. But the Kubernetes community has already moved on to eBPF-based solutions that offer far more than just filtering and load-balancing rules, with better performance, observability, and policy control.

Performance is out of scope for this book, but to see how eBPF enables host-level performance for container networking, I highly recommend Daniel Borkmann's talk called "Turning Up Performance to 11".

One thing that all these approaches have in common—IPVS, `iptables`, `nftables`, and eBPF—is that they all act within the kernel. Recalling that the kernel is shared across all the containers on a host, this tells you that when they are used to enforce security policies, this is happening at the host level and not within each container.

As you saw in Chapter 9, eBPF is being adopted by many infrastructure tools in not only networking but also observability, security, and more. I showed you the `iptables` rules for load-balancing an `nginx` service across two pods. Let's consider the same in an eBPF-based alternative.

eBPF

The Cilium networking plug-in (a graduated project in the CNCF) was designed using eBPF from the start, and Calico also has an eBPF option. I'll use Cilium for this example, but the general approach for Layer 3/4 policies is similar in Calico.

Using Cilium with kube-proxy replacement enabled, there are no `iptables` rules for service load-balancing. Instead, Cilium installs eBPF programs directly into the kernel that intercept network packets. It also maintains data structures in the kernel called *eBPF maps* to hold information, such as the mapping of service IP addresses and ports to backend pods.

When a pod sends a network packet to a service, a Cilium eBPF program does the following:

1. Intercepts the packet within the network stack and identifies the destination IP address and port
2. Looks up the service in the eBPF map to find the pods that back that service
3. Selects a backend pod using a load-balancing algorithm (e.g., random, round-robin, Maglev)
4. Rewrites the packet with the destination IP and port for the selected pod
5. Forwards the packet

All this happens entirely within the kernel, avoiding any costly transitions to user space.

Instead of inspecting `iptables` rules to see how service load-balancing behaves, you can inspect Cilium's eBPF map data using the `cilium-dbg` CLI. For example:

```
$ cilium-dbg bpf lb list

SERVICE ADDRESS      BACKEND ID  BACKEND ADDRESS
10.100.132.10:8080   1           10.32.0.3:80
                     2           10.32.0.4:80
```

Now that you have an idea how network packets are manipulated using `iptables` or eBPF programs, let's see how these technologies are used to implement networking policies for security purposes.

Network Policies

Network policies in Kubernetes define the traffic that can flow to and from different pods. When a message is being sent or received, if it's not approved by the policy, the network needs to either refuse to set up a connection or drop the message packets. In the ecommerce example from the start of this chapter, one policy might prevent traffic from the product search container that has the destination address of the payment service.

Policies enforced at Layers 3 and 4 can be defined in terms of ports, IP addresses, or—more commonly—the labels applied to pods. (We'll come to application layer policies soon.)

In a traditional environment, an application would typically be installed on a machine, and then the IP address for that machine could be used to identify the application. If multiple applications run on the same machine, they will use different ports at Layer 4. It made sense, in these environments, for firewall rules to be based on IP address and ports, but in a Kubernetes world, pods are ephemeral, and IP addresses assigned to them can be reused as pods are destroyed and created. It no longer makes sense to use IP addresses and ports to identify a containerized workload.

Labels in Kubernetes can be used for all sorts of purposes, one of which is to indicate which service a pod is part of and perhaps also some other abstraction such as an application name. We can more generally think of labels identifying the *workload* that is running within a given pod. Network policies define the rules that allow or deny traffic between workloads and perhaps to other entities such as destinations outside the cluster.

Here's a simple `NetworkPolicy` object that allows pods to access the `my-nginx` service only if they are labeled with `access=true`:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: access-nginx
spec:
  podSelector:
    matchLabels:
      app: my-nginx
  ingress:
  - from:
    - podSelector:
        matchLabels:
          access: "true"
```

It's the networking plug-in, rather than a core Kubernetes component, that enforces network policy, and the mechanism used could be `iptables`, eBPF, or maybe in the future something else!

Layer 3/4 Policy with iptables

Creating this network policy in a cluster using the Weave networking plug-in results in the following additional `iptables` rule in the filter table:

```
Chain WEAVE-NPC-INGRESS (1 references)
target     prot opt source       destination
ACCEPT     all  --  anywhere     anywhere             match-set weave-{U;]TI.l|Md
RzDhN7$NRn[t)d src match-set weave-vC070kAfB$if8}PFMX{V9Mv2m dst
/* pods: namespace: default, selector: access=true ->
pods: namespace: default, selector: app=my-nginx (ingress) */
```

The match-set rule isn't really human-readable, but the comment (between `/*` and `*/`) matches our expectation that the rule allows traffic from pods in the default namespace, with the label `access=true` going to pods in the default namespace with the label `app=my-nginx`.

Now that you have seen Kubernetes using `iptables` rules for network policy enforcement, let's try configuring an `iptables` rule of our own. I'm doing this on a fresh Ubuntu installation so that the rules are empty to start with:

```
$ sudo iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

I'll set up `netcat` to respond to requests on port 8000:

```
$ while true; do  echo "hello world" | nc -l 8000 -N; done
```

In another terminal, I can now send requests to this port:

```
$ curl localhost:8000
hello world
```

Now I'll create a rule that rejects traffic on port 8000:

```
$ sudo iptables -I INPUT -j REJECT -p tcp --dport=8000
$ sudo iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source       destination
REJECT     tcp  --  anywhere     anywhere         tcp dpt:8000 reject-with icmp-
port-unreachable
```

```
Chain FORWARD (policy ACCEPT)
target     prot opt source       destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source       destination
```

As you probably would have predicted, the `curl` command no longer succeeds in getting a response:

```
$ curl localhost:8000
curl: (7) Failed to connect to localhost port 8000: Connection refused
```

This demonstrates that `iptables` can be used to restrict traffic. You could imagine building up lots of rules like this to limit the traffic between containers, creating your own container network security policy, but I don't recommend doing it by hand unless your deployment is small-scale and relatively static. Even a small Kubernetes deployment involves more rules than I would want to write manually. To give you an idea, I have a single Kubernetes node running the Calico network plug-in, and with just a handful of application pods running and no network policies, `iptables -L` on this machine gives me more than 300 lines of filter table rules.

Layer 3/4 Policies with eBPF

You've just seen how a network plug-in could use `iptables` rules to enforce an example network policy that restricts access to the `my-nginx` service to pods labeled with `access=true`. If we create the same policy using Cilium as the network plug-in, no `iptables` rules are created. Instead, the policy is enforced using eBPF programs and maps.

To do this, Cilium creates an *identity* for a pod based on a hash of its labels. Here's an example of looking at the set of identities that Cilium is tracking:

```
$ cilium-dbg identity list

ID      LABELS
123     k8s:app=my-nginx
45      k8s:access=true
...
```

Policies are translated into allow/deny rules for ingress and egress to and from an endpoint. The example rule we have been considering translates into the following:

```
$ cilium-dbg bpf policy get 123
Ingress:
  From Identity 45: ALLOW
  All other: DROP
```

The policy is an ingress rule that applies to the identity `123`, which matches `app=my-nginx` and only allows traffic that comes from identity `45`, corresponding to `access=true`.

The policy rules are held in eBPF Maps in a hash table, making for really fast lookups within the kernel when the Cilium eBPF program in the network stack intercepts a packet destined for (in this case) an endpoint with identity `123`.

So far, the policies we have considered have rules in terms of network Layers 3 and 4. Some solutions can also enforce solutions at the application Layer 7.

Layer 7 Policies

Policies at Layer 3/4 might allow or deny traffic between a given pair of endpoints, but that policy will drop all matching traffic. Layer 7 policies can specify the traffic to be permitted or blocked based on application layer protocol characteristics. For HTTP traffic, this could be to allow only certain URLs, methods, or specific headers. DNS traffic can be filtered on, say, domain name or query type (A, AAAA, MX, etc.). GRPC traffic can be blocked based on the method, service name, or headers.

Cilium, Calico, and Antrea have support for Layer 7 network policies, implemented by integrating the Envoy proxy. Rule enforcement happens in user space within the proxy.

I first saw Cilium in action at a talk by Thomas Graf at DockerCon 2017, when he showed the "Star Wars demo". This includes Layer 3/4 policies to ensure that only Empire spacecraft can access the Death Star, and Layer 7 policies to prevent Empire spacecraft from doing something dangerous, like an HTTP PUT request sent to the Death Star's API at `/v1/exhaust-port`. You can try this for yourself in a Kubernetes cluster by following the documentation.

One of the benefits of Cilium's approach to network policy is that Layer 3/4, Layer 7, and even authentication policies can all be defined as a single resource:

```
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: mutual-auth-echo
spec:
  endpointSelector:
    matchLabels:
      app: echo                          ❶
  ingress:
  - fromEndpoints:
    - matchLabels:
        app: pod-worker                  ❷
```

```
  authentication:
    mode: "required"                          ❸
  toPorts:
  - ports:
    - port: "3000"
      protocol: TCP                           ❹
    rules:
      http:
      - method: "GET"
        path: "/headers"                      ❺
```

❶ This policy applies to any pod with the label `app=echo`.

❷ This is an ingress policy that applies to traffic from pods with the label `app=pod-worker`.

❸ The policy specifies that traffic between these endpoints must be mutually authenticated, which also requires that both pods are cryptographically identified before the traffic is permitted. Authentication is covered in Chapter 13.

❹ Traffic is permitted only if it is TCP traffic to port 3000.

❺ And it must also be a GET method HTTP request to the path `/headers`.

The simplicity of combining all policy requirements makes for easier operations and troubleshooting.

Network Policy Solutions

Kubernetes has `NetworkPolicy` objects, although as mentioned earlier, Kubernetes does not itself enforce them—they have to be enforced by a CNI plug-in, and that will happen only if you choose a CNI that supports this. Perhaps unfortunately, if the CNI plug-in doesn't enforce Kubernetes Network Policies, *they are silently ignored*. Check carefully to make sure that the CNI you're using actually does something useful with them. I have heard sad stories about operators spending ages configuring policies, without realizing that they had absolutely no effect.

Depending on the CNI plug-in, you may have options for upgrading to a commercial version that gives you more flexibility, additional visibility, or easier management. There are commercial container security platforms that include container firewalls to achieve essentially the same thing but are not installed directly as a Kubernetes network plug-in. Some commercial offerings include the ability to learn what normal traffic looks like for a particular container image or workload type so that policies can be created automatically.

If the CNI plugin you're using doesn't support Layer 7 policies, that might be a reason to consider using a service mesh.

Service Mesh

A *service mesh* can be applied in addition to the CNI plug-in in a Kubernetes deployment, to provide an additional set of controls and capabilities at the application layer. "Service mesh" can mean different things to different people, potentially encompassing service discovery, Layer 7 load balancing, canary roll-outs, and observability features, but for the purposes of this book, let's consider what is probably the most fundamental capability: providing secure network connectivity between containerized workloads. This consists of two parts: encrypted, authenticated connections, which are covered in Chapter 13, and restricting network traffic at the application layer by enforcing Layer 7 policies. In most service mesh implementations, these policies are separated from the Layer 3/4 policies enforced by a CNI.

This article from DigitalOcean gives a good overview of the other features that service meshes offer, such as canary deployments, that are unrelated to networking or security.

Examples of service mesh projects include Istio and Linkerd as well as managed options from the cloud providers such as AWS ECS Service Connect. The Cilium CNI provides a lot of features normally associated with service mesh but without an additional service mesh control plane to manage.

Here's an example of an Istio Authorization Policy for restricting HTTP traffic:

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: echo
spec:
  selector:
    matchLabels:
      app: echo                                    ❶
  action: ALLOW                                    ❷
  rules:
  - from:
    - source:                                      ❸
        principals: ["cluster.local/ns/default/sa/pod-worker"]
    to:
    - operation:
        methods: ["GET"]                           ❹
```

❶ Like the previous example for "Layer 7 Policies," the policy applies to pods with the label `app=echo`.

❷ It's a policy that allows traffic.

❸ Requests are permitted only if they originate from a workload with an X.509 certificate identifying it as using the `pod-worker` service account. We'll discuss certificates and identities further in Chapter 13.

❹ Only (HTTP) GET requests are permitted.

Until recently, most service meshes were implemented using sidecar containers, which have some limitations, as discussed in Chapter 9. There are now sidecarless options for service mesh, including Istio Ambient Mesh, and the service mesh capabilities of Cilium, that provide the same functionality but without requiring a sidecar in every pod.

If you are using or considering a sidecar-based implementation, they present a couple of additional security-related constraints to be aware of:

- A sidecar can only provide security support to pods into which it has actually been injected. If it's not present, it can't do anything. Carefully test and/or audit configurations to ensure that the mesh is applied to all the workloads where it is expected.
- A service mesh sidecar container lives alongside application containers within a pod. If an application container were to be compromised, it might attempt to bypass or modify the rules enforced by the sidecar. The sidecar and application containers share the same network namespace; thus, it is a good idea to make sure that the `CAP_NET_ADMIN` capability is withheld from application containers so that if one is compromised, it can't modify the shared networking stack.
- The sidecar life cycle is tightly coupled to the life cycle of the pod. This means restarting your pods if the service mesh needs an upgrade.

Service mesh policies are defined at the service level, so it would be a good idea to use the principle of defense in depth, with a complementary container network or runtime security solution to restrict traffic that flows directly to containers rather than via a service IP address.

Network Policy Best Practices

Whichever tooling you use to create, manage, and enforce network policies, there are some recommended best practices:

Default deny
: Following the principle of least privilege, set up a policy for each namespace that denies ingress traffic by default and then add policies to permit traffic only where you expect it.

Default deny egress
: Egress policies relate to traffic exiting your pod. If a container were to be compromised, an attacker could probe the surrounding environment across the network. Set up policies for each namespace to deny egress traffic by default and then add policies for expected egress traffic.

Restrict pod-to-pod traffic
: Pods are typically labeled to indicate their application. Use policies to limit traffic so that it can flow only between permitted applications, along with policies that allow traffic only from pods with the appropriate labels.

Restrict ports
: Restrict traffic so that it is accepted only to specific ports for each application.

Ahmet Alp Balkan provides a set of useful network policy recipes.

Summary

In this chapter, you have seen how containers enable very granular firewalling solutions within a deployment. This granularity helps maintain several security principles:

- Segregation of duties/least privilege by allowing containers only a limited ability to communicate
- Limiting the blast radius by preventing a compromised container from attacking all its neighbors
- Defense in depth by combining network policies at Layer 3/4, Layer 7, and cluster-wide traditional firewalling

I mentioned that you might want to consider encrypting network traffic between containers. In Chapter 13, I will explain some approaches to securing and encrypting traffic and attempt to demystify the role of keys and certificates in setting up these secure connections.

CHAPTER 13

Securely Connecting Components

In any distributed system, there are different components that need to communicate with each other, and in a cloud native world, those components may well be containers exchanging messages with each other or with other internal or external components. In this chapter, you'll see how secured network connections allow components to safely send encrypted messages to each other. You'll explore how components identify themselves to each other and set up secure connections between themselves so that malicious components can't get involved in these communications.

If you're familiar with how keys and certificates work, you can safely skip this chapter, as there is nothing particularly container-specific about it. I have included it in this book because in my experience, it's an area of confusion for many folks who may be coming across these concepts for the first time when they start exploring containers and cloud native tools.

If you are responsible for administering a cloud native system, you will likely need to configure certificates, keys, and certificate authorities (CAs) for Kubernetes, etcd, or other infrastructure components. These can be notoriously confusing, and installation instructions tend to explain what to do without covering the "why" or the "how." You may find this chapter useful for understanding the roles that these different pieces play.

Let's start by considering what we mean by "secure connections."

Secure Connections

In everyday life, we see secure connections being used in web browsers. If you visit, say, your online banking facility, most browsers will alert you if the connection isn't secure, so you shouldn't enter your login credentials. There are two parts to setting up a secure connection to a website:

- First, you need to know that the website you are browsing really is owned by your bank. Your browser checks the identity of the website by verifying its certificate. This part is known as *authentication*.
- The second part is *encryption*. When you are accessing your bank information, you don't want any third parties to be able to intercept (or worse, interfere with) that communication channel.

You may well be familiar with the fact that secure website connections use a protocol called *HTTPS*, which stands for HTTP-Secure. As its name suggests, this is a regular HTTP connection that has been made secure, and this security is added at the transport layer using a protocol imaginatively called *transport layer security* (TLS).

If you're thinking, "But I thought the *S* stood for SSL (Secure Sockets Layer)?" don't worry—you're really not wrong. The transport layer is the layer that communicates between a pair of network sockets, and TLS is the modern name for the protocol that used to be called SSL. The first SSL spec was published by Netscape in 1995 (as version 2, the initial version 1 having been recognized as so seriously flawed that it was never released). By 1999, the Internet Engineering Task Force (IETF) created the TLS v1.0 standard, largely based on Netscape's SSL v3.0, and the industry is now primarily using TLS v1.3.

Whether you call it SSL or TLS, the protocol relies on certificates to set up secure connections. Confusingly, we still tend to call these "SSL certificates" 20 years after the move to TLS. If you really want to be correct, you should call them "X.509 certificates."

TLS isn't the only option for secure connections, and as you'll see later in this chapter, protocols like IPSec and WireGuard are also commonly used to encrypt containerized traffic. These protocols all involve an exchange of identity information and the encryption of traffic using a cryptographic cipher.

Both identity and encryption key information are commonly exchanged using X.509 certificates. Let's delve into what these certificates are and how they work.

There are several tools for generating keys, certificates, and CAs including `ssh-keygen`, `openssl keygen`, and `minica`. I demonstrated using `minica` in a talk called "A Go Programmer's Guide to Secure Connections" in which I also show what's happening step-by-step as a client sets up a TLS connection with a server.

X.509 Certificates

The term "X.509" is the name of the International Telecommunications Union (ITU) standard that defines these certificates. The certificate is a piece of structured data that includes information about the identity of its owner and also includes the public encryption key for communicating with the owner. This public key is half of a public/private key pair (see Figure 13-1).

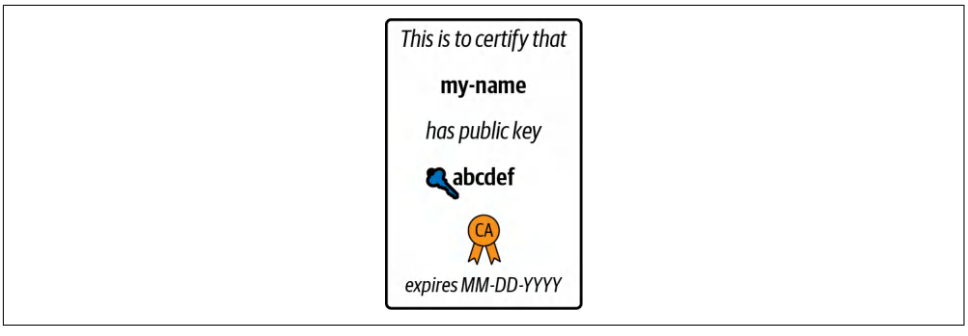


Figure 13-1. Certificate

As illustrated in Figure 13-1, the vital pieces of information in a certificate are:

- The name of the entity that this certificate identifies. This entity is called the *subject*, and the subject name is typically in the form of a domain name. In practice, certificates should use a field called "Subject Alternative Names," which allows the certificate to identify the subject by more than one name.
- The subject's public key.
- The name of the CA that issued the certificate. I'll come back to this later in "Certificate Authorities" on page 187.
- The validity of the certificate—that is, the date and time at which the certificate expires.

Public/Private Key Pairs

Public/private keys are an example of *asymmetric encryption*, where data is encrypted using the public key, but the encrypted data can only be decrypted using the private key. As its name suggests, a public key can be shared with anyone. The public key has a corresponding private key that the owner should never disclose.

The math behind the encryption and decryption is beyond the scope of this book, but I collected some recommended resources about it in a post on Medium.

The private key is generated first, and from that, a corresponding public key can be calculated. The public/private key pair can be used for two useful purposes:

- As illustrated in Figure 13-2, a public key can be used to encrypt a message that can be decrypted only by the holder of the corresponding private key.

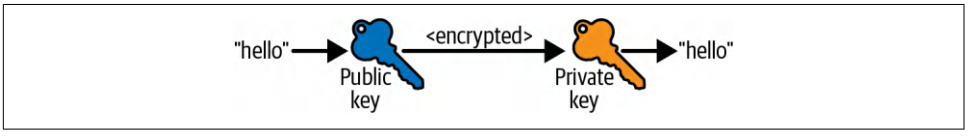


Figure 13-2. Encryption

- A private key can be used to sign a message that any holder of the corresponding public key can check to verify that it came from the private key owner. This is shown in Figure 13-3.

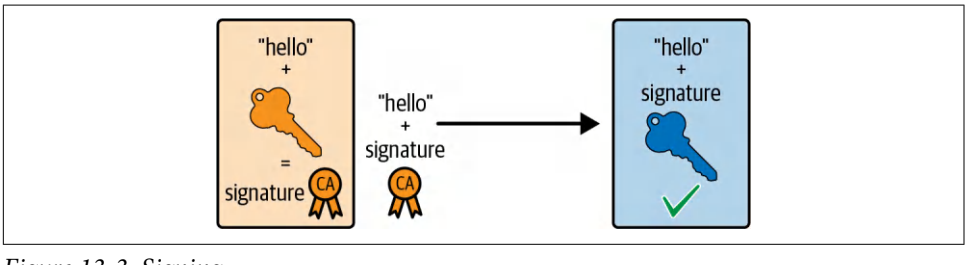


Figure 13-3. Signing

Both the encryption and the signing capabilities of public/private key pairs are used to set up secure connections.

Let's suppose that you and I want to exchange encrypted messages. Once I have generated a key pair, I can give you the public key so that you can send me encrypted messages. But if I send you that public key, how do you know that it really came from me and not from an imposter? To establish that I am who I say I am, we will need to involve a third party that you trust and that will vouch for my identity. This is the function of a *certificate authority* (CA).

Certificate Authorities

A CA is a trusted entity that signs a certificate, thus verifying that the identity contained in that certificate is correct. You should only trust a certificate that has been signed by an authority you trust.

A client that initiates a TLS connection receives a certificate from the destination, which it can check to make sure that it is talking to the entity that it intended to reach. For example, when you open a web connection to your bank, your browser checks that the certificate matches the URL of your bank, and it also checks what CA signed the certificate.

Other components need to be able to safely identify the CA, so it is represented by a certificate. But that certificate needs to be signed by a CA, and to verify the signer's identity, there needs to be another certificate, and so on and so forth. It seems that we could build a never-ending chain of certificates! Eventually, there has to be a certificate that we can trust.

In practice, the chain ends with what's called a *self-signed certificate*: an X.509 certificate that the CA signed for itself. In other words, the identity represented by the certificate is the same as the identity whose private key is used to sign the certificate. If you can trust that identity, you can trust the certificate. Figure 13-4 shows a certificate chain, where Ann's certificate is signed by Bob, and Bob's is signed by Carol. The chain ends with Carol's self-signed certificate.

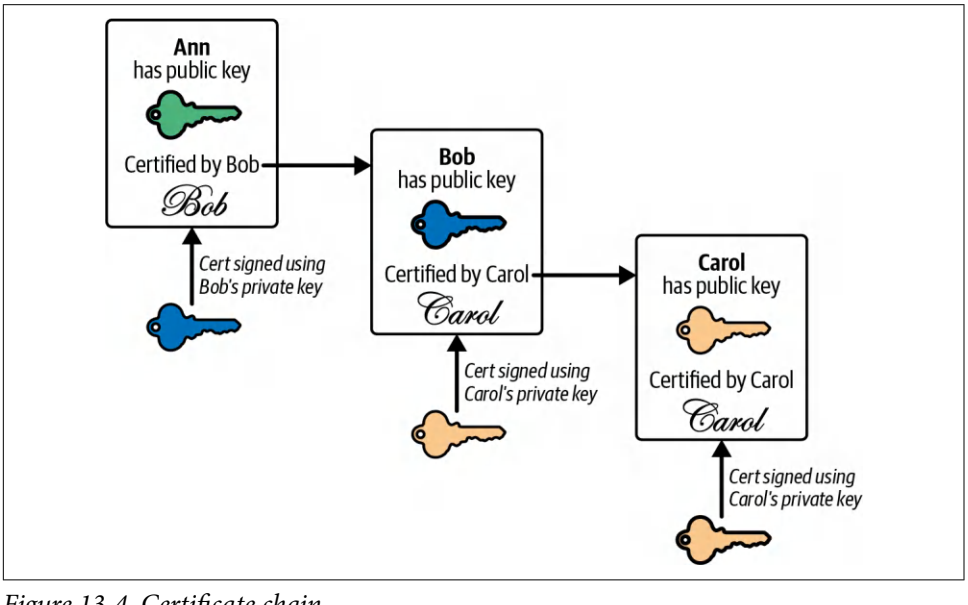


Figure 13-4. Certificate chain

Web browsers use a set of certificates from well-known, trusted CAs known as *root CAs*. These might be installed along with the browser or owned by the operating system. Your web browser will trust any certificate (or certificate chain) signed by one of these root CAs. If the certificate isn't signed by one of the trusted CAs, it will show the site as insecure (and in browsers today, you will almost certainly see a warning or error message).

If you're setting up a website that people will connect to over the internet, you will need a certificate for that website signed by a trusted, public CA. There are several vendors who act as these CAs and will generate a certificate for a fee, or you can get one for free from Let's Encrypt.

When you're setting up distributed system components such as, say, Kubernetes or etcd, you get to specify a set of CAs that are used to validate certificates. Assuming that your system is under your private control, it doesn't matter to you whether members of the public at large (or their browsers) trust these components—the important thing is that the components can trust each other. Because this is a private system, you don't need to use publicly trusted CAs, and you can simply set up your own CAs with self-signed certificates.

In an enterprise environment, a Public Key Infrastructure (PKI) often includes CA capabilities and typically also manages certificate policies, renewal, and revocation.

Whether you're using your own CA or a public one, you'll need to tell the CA about the certificate(s) you want generated. This is done with a *Certificate Signing Request*.

Certificate Signing Requests

A Certificate Signing Request (CSR) is a file that includes the following information:

- The public key that the certificate will incorporate
- The domain name(s) that this certificate should work with
- Information about the identity that this certificate should represent (for example, the name of your company or organization)

You already know that an X.509 certificate includes this information, plus the signature from the CA, so it makes complete sense that this is what you send in a CSR to request an X.509 certificate from a CA.

Tools like `openssl` can create a new key pair and CSR in one step. Perhaps confusingly, `openssl` can take a private key as input for generating a CSR. This makes sense when you recall that the public key is derived from the private key. The component running as this identity (represented by the certificate) will use the private key for decrypting and signing messages (as you'll see shortly), but it never uses the public key itself. It's the other components that it communicates with who will need the

public key, and they get that from the certificate. This component needs the private key, and it needs the certificate that it will send to other components.

Now that you have a good understanding of what a certificate is, let's discuss how certificates are used for TLS connections.

TLS Connections

Transport layer connections have to be initiated by a component, and that component is called the *client*. The entity it is communicating with is called the server. It may well be the case that this client-server relationship is true only at the transport layer, and at higher layers the components could be peers.

A client opens a socket and requests a network connection to the server. For a secure connection, it requests that the server should send back a certificate. As you know from earlier in this chapter, the certificate conveys two important pieces of information: the identity of the server and its public key.

The point of this is that the client can check that the server can be trusted. The client checks that the server's certificate was signed by a trusted CA, and if so, that is confirmation that the server can be trusted. The client can go on to use the server's public key to encrypt messages that it sends to the server. The client and server agree on a symmetric key used to both encrypt and decrypt the remainder of the messages transferred on this connection—this is more performant than using the asymmetric public/private key pair. This message flow is shown in Figure 13-5.

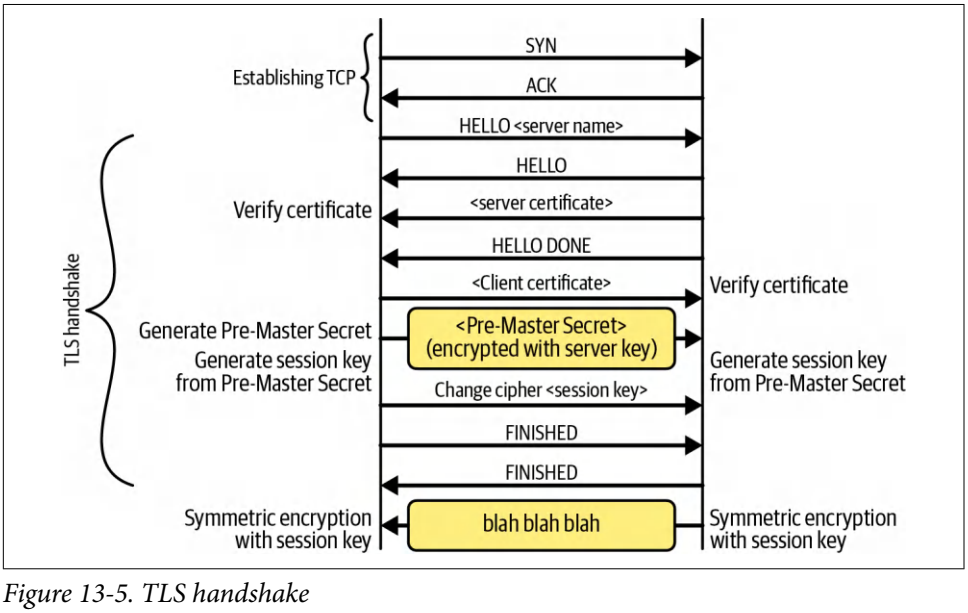


Figure 13-5. TLS handshake

You may have come across the term *skip verify*. This refers to an option at the transport layer that allows a client to skip the step where it verifies that the certificate was signed by a known CA. The client simply assumes that the identity claimed by the certificate is correct. This can be handy in nonproduction environments because it means you don't have to bother configuring the client with information about CAs, and you can simply use self-signed certificates. You still have encrypted communications between components, but you can't have full confidence that components aren't imposters, so please don't use skip-verify options in production!

Once the client has verified the server, it can trust it. But how can the server trust the client?

If we were discussing a website where you have an account, like a bank, it's important that the server verifies your identity before it gives out details of your bank balance, or worse. For a true client/server relationship such as logging into your bank, this is typically dealt with through Layer 7 authentication. You supply a username and password, perhaps supplementing this with multifactor authentication through a code sent in a text message, through a one-time password generated by a Yubikey, or by a mobile app like Authy, 1Password, or Google Auth.

Another way to validate the client's identity is through another X.509 certificate. The message flow in Figure 13-5 shows both the server and client certificates being exchanged—this is an option that can be configured at the server side. The server used one to confirm its identity to the client, so why not do the same thing in reverse? When this happens, it's called *mutual TLS* or mTLS.

TLS and mTLS connections are widely used between clients and servers, and containers (or Kubernetes pods) can be either (or both). But these containers run on hosts, and an alternative method for encrypting the traffic between containers is to encrypt all the network traffic between their hosts using a technology such as WireGuard or IPSec.

WireGuard and IPSec

While they differ in implementation, these technologies encrypt IP (Layer 3) traffic by setting up a secure tunnel between endpoints, encapsulating the traffic that flows through the tunnel. They are both commonly used for virtual private networks (VPNs), that you have likely encountered for securely connecting computers to your employer's network, or to make a device appear to be located in a different country.

In WireGuard, a network interface is created on a device, typically called `wg0` (and `wg1`, `wg2`, etc., if there are multiple). You'll be familiar with the idea of creating a network interface from Chapter 4, where you saw containers being connected to their host across a virtual Ethernet connection. WireGuard uses a UDP tunnel to

encapsulate and encrypt IP packets. Traffic can be routed to this interface (just like any other).

The WireGuard interface is configured with its private key and a list of peers, public keys for each peer, and the IP addresses that are allowed to send traffic from that peer. Here's an example of this configuration, taken from the WireGuard website:

```
[Interface]
PrivateKey = yAnz5TF+lXXJte14tji3zlMNq+hd2rYUIgJBgB3fBmk=
ListenPort = 51820

[Peer]
PublicKey = xTIBA5rboUvnH4htodjb6e697QjLERt1NAB4mZqp8Dg=
AllowedIPs = 10.192.122.3/32, 10.192.124.1/24

[Peer]
PublicKey = TrMvSoP4jYQlY6RIzBgbssQqY3vxI2Pi+y71lOWWXX0=
AllowedIPs = 10.192.122.4/32, 192.168.0.0/16

[Peer]
PublicKey = gN65BkIKy1eCE9pP1wdc8ROUtkHLF2PfAqYdyYBz6EA=
AllowedIPs = 10.10.10.230/32
```

Packets are encrypted using the public key for the destination peer and encapsulated in a UDP packet. The recipient decrypts the packet using its private key and checks that the source IP is permitted, dropping the packet if not.

WireGuard is generally considered to be highly secure (at least until we reach the era of post-quantum cryptography) and simple to configure, so it's a great choice for transparently encrypting traffic between nodes. But there is a downside to WireGuard, which is that it's not FIPS (Federal Information Processing Standards) compliant—not because it's technically not secure enough but because the developers quite reasonably don't want to go through the bureaucratic process of getting their algorithms verified by FIPS. In regulated environments, it's important to be able to tick the compliance box, so you might want to consider IPSec.[1]

In IPSec, there is no dedicated network interface; instead, IPSec security policies are configured on existing interfaces. This configuration includes the host's own security settings, its private key and the public keys of its peers, as well as the IP addresses and networks that are allowed to send traffic through the IPSec tunnel. IPSec also allows the configuration of a selection of encryption and authentication algorithms, so the first step between peers is to establish a "security association" that establishes the security parameters to be used between these peers, including the encryption and authentication algorithms.

1 WolfSSL looks to replace the cryptography suite in WireGuard with FIPS-compliant algorithms.

When IPSec is enabled on a network interface, it intercepts IP packets being sent or received on that interface and applies the IPSec security policies to them, encrypting and authenticating the packets, as well as checking the source IP address and other security parameters.

IPSec can operate in different modes: transport mode or tunnel mode, which affect how packets are processed and encrypted. Tunnel mode is considered more secure because it encrypts the IP header for the original packet, which includes the source and destination IP addresses, and it can be used where the packets will traverse NAT devices. Transport mode generally has better performance, and it's perfectly suitable in a private network where the IP addresses are not considered sensitive.

WireGuard and IPSec are widely used in container deployments to achieve zero-trust networking.

Zero-Trust Networking

Zero-trust networking is a security approach that assumes that all network traffic, whether it's coming from within or outside the network, is potentially malicious and that malicious actors may have access to the network. To mitigate the risks that this assumption suggests, all traffic should be encrypted and authenticated.

Algorithms like WireGuard and IPSec can play a crucial role in zero-trust networking by providing secure and encrypted communication between devices. Where containers are running on hosts connected to a zero-trust network, the traffic is automatically encrypted as it travels from one host to another, using the hosts' identities for authentication. For many deployments, including regulated environments, this is sufficient to meet compliance requirements, and it is operationally very straightforward.

In a Kubernetes network, as you saw in Chapter 12, connectivity between pods is provided by a Container Network Interface (CNI) plug-in. Some CNIs, including Cilium and Calico, can be configured to use IPSec or WireGuard, ensuring that all traffic between hosts is encrypted.

If you don't own the whole network that containers are connected to (for example, the VPCs they are attached to and any VPN connections between clusters) or you don't trust all the containers deployed in it, encrypting the network traffic may not be sufficient. In this case, you can authenticate and encrypt the traffic between individual containers (or pods, in Kubernetes).

Secure Connections Between Containers

Nothing in this chapter so far is specific to containers, but now let's review some of the circumstances in which you might need to understand keys, certificates, and CAs:

- If you are installing or administering Kubernetes or other distributed system components, it's likely that you'll come across options for using secure connections. Installation tools like `kubeadm` make it easy to use TLS between control plane components, automatically configuring certificates as appropriate. This doesn't do anything to secure the traffic between containers and the outside world.
- As a developer, you might write application code that sets up mTLS connections with other components (whether it's running in a container or not). In that case, your app code needs access to certificates that you'll need to create.
- Rather than writing your own code to set up authenticated and encrypted connections, you can use a service mesh to do it for you. Service meshes were discussed in Chapter 12, and I'll cover how they can provide encrypted connections between containers very soon in this chapter, but first let's talk about revoking certificates.

Certificate Revocation

Imagine that an attacker somehow obtains a private key. They can now impersonate the identity associated with that key, because they can successfully decrypt messages that were encrypted using the public key embedded in any corresponding certificates. To prevent this, you need a way of invalidating the certificate immediately rather than waiting for its expiry date.

This invalidation is called *certificate revocation* and can be achieved by maintaining a Certificate Revocation List (CRL) of certificates that should no longer be accepted.

Try not to share identities (and their certificates) across multiple components or users. It may seem like a management burden to set up individual identities and certificates for each component, but it means you can revoke the certificate for one identity without having to reissue a new one to all the (legitimate) users. It also allows for a separation of concerns whereby each identity can be granted a separate set of permissions.

In Kubernetes, certificates are used by the kubelet component on each node to authenticate to the API server and confirm that it really is an authorized kubelet. These certificates are issued by the cluster's CA and can be rotated automatically.

Certificates are also one of the mechanisms that clients (whether human or software) can use to authenticate themselves with the Kubernetes API Server.

At the time of writing, Kubernetes does not support certificate revocation. Once a certificate is issued, there is no way to invalidate it until it expires. Instead, best practice is to use short-lived certificates and automatic rotation. You can also use RBAC configuration to prevent API access for the client associated with a certificate (but this won't stop a certificate being used to establish a TLS connection).

Service Meshes for Encrypted Traffic

You already met service meshes and saw how they can be used to enforce application-level network policies, but I glossed over how they provide authenticated and encrypted traffic between workloads. Let's dig into that now.

As mentioned previously, for a long time, service meshes typically used the sidecar model, injecting a proxy component into every pod (or at least every pod that's being connected to the service mesh). This proxy intercepts network connections going to and from a container. When pod A initiates a connection to pod B, proxy A acts as the endpoint for that connection, sets up an mTLS connection to proxy B, and forwards the payload data over that connection.

However, this sidecar model has several shortcomings, especially at scale. If you run 1,000 pods, you have 1,000 additional containers for the sidecar proxies, each consuming resources such as the memory for its routing table and increasing CPU load. Traffic has to travel extra hops between proxies and application containers, affecting network latency. Perhaps even more importantly, many users found that operating sidecar-based service meshes introduced significant complexity. The sidecars need to be injected correctly, and as mentioned before, they are tightly bound to the life cycle of the pod—so upgrading the service mesh involves restarting application pods.

As a result, many teams have turned to sidecarless approaches such as Cilium or Istio Ambient Mode, which offer improved efficiency, faster deployment times, and simpler operations while still preserving core mesh features like workload authentication, encrypted connections, Layer 7 policies, and observability.

Thomas Graf laid out the evolution from sidecar-based to sidecarless service mesh, while the Istio documentation gives a good comparison of its sidecar and ambient modes.

For a service mesh to authenticate on behalf of a container or pod, it needs to access an X.509 certificate representing that individual workload. As you've seen, there are several steps to creating X.509 certificates, so issuing certificates and managing identities for thousands of workloads is challenging. One initiative to ease this challenge is *SPIFFE*.

SPIFFE

SPIFFE stands for Secure Production Identity Framework For Everyone. It defines a standard way for issuing and managing identities, known as SVIDs (SPIFFE Verifiable Identity Documents), which in practice are usually x.509 certificates, though Json Web Token (JWT) is also supported. The key idea is to decouple workload identity from machine identity or from traditional approaches like IP addresses, even in dynamic environments like Kubernetes, where workloads are short-lived and frequently rescheduled to different host machines.

The SPIFFE Runtime Environment (SPIRE) is an open source implementation of the SPIFFE specification. It acts as a CA and workload attestor, automatically issuing SVIDs to workloads based on characteristics such as container image digests or Kubernetes pod labels. SPIRE ensures that only authorized workloads receive valid identities and that those identities are rotated automatically and securely.

As you know from earlier in this chapter, setting up an mTLS connection requires a private key and an X.509 certificate. Let's consider in a little more detail how keys and certificates are managed with SPIFFE/SPIRE:

- Each node runs a SPIRE agent, exposing the SPIRE Workload API through a socket, so that only local processes can connect to it.
- A workload (a pod in Kubernetes, but it could be any process in other environments) starts up and makes a request to the SPIRE agent through the Workload API socket.
- The SPIRE agent uses *workload attestation plug-ins* to determine what the workload is and its SPIFFE ID.
- Let's assume that it's a valid workload. The SPIRE agent generates a public/private key pair and sends a Certificate Signing Request to the SPIRE server.

- The server acts as a CA and responds with an SVID containing an X.509 certificate for the workload.
- The agent passes the certificate and the private key to the workload, which can now use this for setting up TLS or mTLS connections.

This allows the workload to be authenticated without keys and certificates having to be manually provisioned.

There are different options for the plug-in that provides workload attestation for SPIRE. Typically, the plug-in identifies the workload process using its process ID and collects information about that process to identify what workload it is. Armed with knowledge from earlier in this book, you can imagine a simple implementation using `/proc/<pid>/exe` to see what binary the process is running. There is a Kubernetes workload attestor that inspects the cgroups associated with the process to determine which pod it belongs to and then queries the Kubernetes API to retrieve metadata about the pod such as namespace, service account, and labels. Configuration on the SPIRE server tells it how to map this metadata to a SPIFFE ID.

Individual applications can use the Workload API directly, or you can use a service mesh to retrieve SVIDs safely on behalf of workloads, so that the whole operation is completely transparent to application code.

External Traffic

So far this chapter has discussed encrypting traffic between containerized workloads, but many deployments also have to consider traffic that flows between a container and some external endpoint:

- There could be ingress requests coming from, say, browsers on the public internet or from clients within your private network but outside the container cluster.
- There could be egress requests made by containers to services outside the container deployment. Examples might include third-party payment gateway services, external storage such as AWS S3, or internal database services.

You need to consider how this external traffic is secured.

Ingress Traffic

Externally, traffic typically goes through some kind of entry point before it reaches a containerized application. This could be a load balancer that sits between the outside world and your deployment, or it could be an Ingress or Gateway API within a Kubernetes cluster or a reverse proxy in a Docker compose setup. These all perform similar functions: they decide which container to send the external request to, and they might optionally terminate TLS connections.

There are a few options for TLS termination:

- Terminate all incoming TLS connections at the ingress layer, and forward requests as plain HTTP traffic to the container. If you're using transparent encryption techniques like WireGuard or IPSec, the traffic within the container network is still secure.
- Use TLS Passthrough, where the TLS connection reaches all the way to the container endpoint (or its service mesh proxy). This may be necessary if there are untrusted workloads in the deployment—for example, in a multitenancy environment.
- Use termination and re-encryption, where the original connection terminates at the ingress layer, but mTLS is used between the ingress and containers.

You might also have restrictions on which sources are permitted for external traffic. This could be a conventional firewall, configuration on the ingress component, or even ingress rules in a container network policy.

Egress Traffic

By default, containers can typically make an outbound connection to any reachable address. You can limit this with egress network policy rules and/or, in the public cloud, Security Group rules. To make it harder for an attacker to exfiltrate data, it's good practice to disable outbound traffic from workloads unless they explicitly need it.

You may also have security such as firewalling in place to protect the external services that your containers need to connect to. An *egress gateway* can make traffic from a containerized workload appear to come from a predictable IP address, making it easier to write the firewall rules. Cilium and Istio both implement egress gateways that run in a Kubernetes cluster. In the public cloud, you can do this at the infrastructure level with NAT gateways and routing table rules.

Network Observability and Logging

Just as you would in a traditional deployment, you should monitor and log inbound and outbound connections, in case of unexpected destinations, or a spike in traffic, that could indicate malicious activity. We'll consider some tools to help with this in Chapter 15.

Many CNIs and service meshes provide network observability tools, like Cilium's Hubble or Istio's Kiali, which can help to debug connectivity issues and provide information about potential compromises.

Summary

To avoid man-in-the-middle attacks, you'll need to make sure you can trust the network connections between your various software components. A good approach to this is to adopt zero-trust networking across your deployment, such that traffic is always encrypted. Transparent encryption between nodes using WireGuard or IPSec is generally easy to configure and requires no changes to your containers or applications.

In some cases, you may need to go further and authenticate individual workloads. Application code running in containers can initiate and/or terminate TLS connections themselves, or a service mesh can authenticate identities on behalf of workloads. Consider also how you will secure ingress and egress traffic to and from your containerized deployment.

Any component that can act as an endpoint for a TLS connection will need three things:

- A private key that should never be shared and should be treated as a secret
- A certificate that it can freely distribute and that other components can use to validate its identity
- Certificates from one or more trusted CAs that it can use to validate the certificates received from other components

Containerized workloads can act as TLS endpoints themselves, or they can offload this functionality to other components such as a service mesh.

You should now have a good understanding of the role that keys, certificates, and CAs each play. This knowledge will be helpful when you're configuring components to use them or debugging connection problems.

If you can trust the connections between containers and identify the component at the far end of a connection, you are in a good place to start passing *secrets* between containers. But you'll need to be able to pass secret values into containers safely—and that's what's coming up in Chapter 14.

CHAPTER 14

Passing Secrets to Containers

Application code often needs certain credentials to do its job. For example, it may need a password to access a database or a token giving it permission to access a particular API. Credentials, or secrets, exist specifically to restrict access to resources—the database or the API, in these examples. It's important to make sure that the secrets themselves stay "secret" and, in compliance with the principle of least privilege, are accessible only to people or components who really need them.

This chapter starts by considering the desirable properties of secrets and then explores the options for getting secret information into containers. It ends with a discussion of native support for secrets in Kubernetes.

Secret Properties

The most obvious property of a secret is that it needs to be secret—that is, it must be accessible only to the people (or things) that are supposed to have access. Typically you ensure this secrecy by encrypting the secret data and sharing the decryption key only with those entities that should have permission to see the secret.

The secret should be stored in encrypted form so that it's not accessible to every user or entity that can access the data store. When the secret moves from storage to wherever it's used, it should also be encrypted so that it can't be sniffed from the network. Ideally, the secret should never be written to disk unencrypted. When the application needs the unencrypted version, it's best if this is held only in memory.

It is perhaps tempting to imagine that once you have encrypted a secret, that is the end of the matter, because you can pass it safely to another component. However, the receiver would need to know how to decrypt the information it received, and that entails a decryption key. This key is in itself a secret, and the receiver would need to

get hold of that somehow, leading us back to the original question of how we can pass this next-level secret safely.

You need to be able to *revoke* secrets—that is, make them invalid in the event that the secret should no longer be trusted. This could happen if you know or suspect that an unauthorized person has been able to access the secret. You might also want to revoke secrets for straightforward operational reasons, such as someone leaving the team.

You also want the ability to *rotate* or change secrets. You won't necessarily know if one of your secrets has been compromised, so by changing them every so often, you ensure that any attacker who has been able to access some credentials will find that they stop working. It's now well-recognized that forcing humans to change passwords regularly is a bad idea, but software components can cope fine with frequently changing credentials.

The life cycle of a secret should ideally be independent of the life cycle of the component that uses it. This is desirable because it means you don't have to rebuild and redistribute the component when the secret changes.

The set of people who should have access to a secret is often much smaller than the set of people who need access to the application source code that will use that secret or who can perform deployments or administration on (parts of) the deployment. For example, in a bank, it's unlikely that developers should have access to production secrets that would grant access to account information. It's quite common for secret access to be write-only for humans: once a secret is generated (often automatically and at random), there may never be a reason for a person to legitimately read the secret out again.

It's not just people who should be restricted from having access to secrets. Ideally, the only software components that can read the secret should be those that need access to it. Since we are concerned with containers, this means exposing a secret only to those containers that actually need it to function correctly.

Now that we have considered the preferred qualities of a secret, let's turn to the possible mechanisms that could be used to get a secret into the application code running in a container.

Getting Information into a Container

Bearing in mind that a container is deliberately intended to be an isolated entity, it should be no surprise that there is a limited set of possibilities for getting information—including secret data—into a running container:

- Data can be included in the container image, as a file in the image root filesystem.
- Environment variables can be defined as part of the configuration that goes along with the image (see Chapter 6 for a reminder of how the root filesystem and config information make up an image).
- The container can receive information over a network interface.
- Environment variables can be defined or overridden at the point where the container is run (for example, including `-e` parameters on a `docker run` command).
- The container can mount a volume from the host and read information out of volumes on that host.

Let's take each of these options in turn.

Storing the Secret in the Container Image

The first two of these options are unsuitable for secret data because they require you to hardcode the secret into the image at build time. While this is certainly possible, it is generally considered a bad idea:

- The secret is viewable by anyone who has access to the source code for the image. You might be thinking that the secret could be encrypted rather than in plain text in the source code—but then you'll need to pass in another secret somehow so that the container can decrypt it. What mechanism will you use to pass in this second secret?
- The secret can't change unless you rebuild the container image, but it would be better to decouple these two activities. Furthermore, a centralized, automated system for managing secrets (like CyberArk or HashiCorp Vault) can't control the life cycle of a secret that is hardcoded in the source.

Unfortunately, it is surprisingly common to find secrets baked into source code. One reason is simply that developers don't all know that it's a bad idea; another is that it's all too easy to put the secrets directly into the code as a shortcut during development or testing, with the intention of removing them later—and then simply forget to come back and take them out.

Several image-scanning tools (discussed in Chapter 8) can help you spot when secrets have been hard-coded into a container image, so you can remove them and use a better mechanism instead!

If passing the secret at build time is off the table, the other options all pass the secret when the container starts or is running.

Passing the Secret Over the Network

The third option is to pass the secret over a network interface. It is commonplace to use managed services like AWS Secrets Manager or HashiCorp Vault to manage secrets, with the application code making API requests over the network to obtain secrets from the service.

But to keep the secret safe, the network communication between the container and the secret service has to be encrypted, which can present something of a bootstrapping problem: as you saw in Chapter 13, the container needs secret credentials to set up the secure network connection.

Transparent encryption might take care of this for you, or you can offload this part of the problem to a service mesh, but the service mesh itself will still need to access credentials passed in through some other mechanism.

Passing Secrets in Environment Variables

The fourth option, passing secrets via environment variables, is generally frowned upon for a couple of reasons:

- In many languages and frameworks, a crash will result in the system dumping debug information that may well include all the environment settings. If this information gets passed to a logging system, anyone who has access to the logs can see secrets passed in as environment variables.
- If you can run `docker inspect` (or an equivalent) on a container, you get to see any environment variables defined for the container, whether at build or at runtime. Administrators who have good reasons for inspecting properties of a container don't necessarily need access to the secrets.
- Anyone with access to the host can simply run `cat /proc/<process id>/environ` to see all the environment variables for that process. You'll see this in more detail later in "Secrets Are Accessible by Root" on page 207.

Here's an example of extracting the environment variables from a container image:

```
$ docker image inspect --format '{{.Config.Env}}' nginx
[PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
NGINX_VERSION=1.27.5 NJS_VERSION=0.8.10 NJS_RELEASE=1~bookworm
PKG_RELEASE=1~bookworm DYNPKG_RELEASE=1~bookworm]
```

You can also easily inspect environment variables at runtime. This example shows how the results include any definitions passed in on the run command (`EXTRA_ENV` here):

```
$ docker run -e EXTRA_ENV=HELLO --rm -d nginx
13bcf3c571268f697f1e562a49e8d545d78aae65b0a102d2da78596b655e2f9a
$ docker container inspect --format '{{.Config.Env}}' 13bcf
[EXTRA_ENV=HELLO PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:
/bin NGINX_VERSION=1.27.5 NJS_VERSION=0.8.10 NJS_RELEASE=1~bookworm
PKG_RELEASE=1~bookworm DYNPKG_RELEASE=1~bookworm]
```

"The Twelve-Factor App" manifesto encouraged developers to pass configuration through environment variables, so in practice you may find yourself running third-party containers that expect to be configured this way, including some secret values. You can mitigate the risk of environment variable secrets in a few ways (which may or may not be worthwhile, depending on your risk profile):

- You could process output logs to remove or obscure the secret values.
- You can modify the app container (or use a sidecar container) to retrieve the secrets from a secure store (like HashiCorp Vault, CyberArk Conjur, or cloud provider secret/key management systems). Some commercial security solutions will provide this integration for you.

AWS Fargate is an example of a managed container service that supports passing secrets using environment variables. Instead of including the secret value directly in the configuration for the Fargate task, the task definition can reference secrets held safely and in encrypted form in AWS Secrets Manager. This means the task definition itself doesn't include sensitive data (which would be similar to holding sensitive data in the source code for a container image). Still, by the time the containerized application running in Fargate sees the value retrieved from the Secrets Manager service, it will be an unencrypted environment variable.

One last thing to note about secrets configured through environment variables is that the environment for a process is configured only once, and that's at the point where the process is created. If you want to rotate a secret, you can't reconfigure the environment for the container from the outside.

Passing Secrets Through Files

A better option for passing secrets is to write them into files that the container can access through a mounted volume. Ideally, this mounted volume should be a temporary directory that is held in memory rather than written to disk. As an example, both Docker Swarm secrets and Kubernetes secrets can be mounted into containers using an in-memory filesystem. Combining this with a secure secrets store ensures that secrets are never stored "at rest" unencrypted.

Because the file is mounted from the host into the container, it can be updated from the host side at any time without having to restart the container. Provided the application knows to retrieve a new secret from the file if the old secret stops working, this means you can rotate secrets without having to restart containers. The requirement for applications to be aware of updated secrets has been made easier through Linux's `inotify` mechanism, where the filesystem can send an event to let a process know when a file has changed. (This `inotify` system is also very useful for runtime security tools that can now subscribe to events related to sensitive file access.)

Kubernetes Secrets

If you're using Kubernetes, the good news is that it has native secrets support that meets many of the criteria I described at the start of this chapter:

- Kubernetes Secrets are created as independent resources, so they are not tied to the life cycle of the application code that needs them.
- Kubernetes secrets are stored (along with other resource data) as base64-encoded values in etcd. Data at rest in etcd is not encrypted by default, but Kubernetes has built-in support that you can enable for encrypting Secrets (and any other resources of your choosing that you might consider sensitive). If you're using a managed Kubernetes service, you'll very likely find that this encryption is either on by default or easily configurable. (It's also possible to encrypt the entire etcd data store, but this is rarely done since Kubernetes started offering resource encryption at the API server level, which is usually easier to manage.)
- Secrets are encrypted in transit between components. This requires that you have secure connections between Kubernetes components (for example, a TLS connection between the API Server and etcd data store), though this is generally the case by default in most distributions.
- Kubernetes Secrets support the file mechanism as well as the environment variable method, mounting secrets as files in a temporary filesystem that is held in-memory and never written to disk.
- You can set up Kubernetes role-based access control (RBAC) so that users can configure Secrets resources but can't access them again, giving them write-only permissions.

In addition to the native Secrets support, Kubernetes now has an optional Secrets Store Container Storage Interface (CSI) Driver that eliminates the need to use native Kubernetes Secrets.

Secrets Store CSI Driver

This extension allows secrets to be pulled directly from a secure secret management service (like Vault or a cloud provider Key Management Service) at runtime and mounted into pods as files. These secrets are never stored in etcd and never exposed as environment variables.

To start using this approach you might need to update your applications to read secrets from the right files, and they need to be restarted on key rotation (unless they can watch for updates using `inotify`).

There is the option to sync these resources to native Kubernetes Secrets, though this would seem to defeat the whole point of using the Secrets Store driver! However, the ability to sync can help during a migration in which certain apps need the legacy Secrets approach—for example, because they read from environment variables. Additionally, some resources might refer to Secrets (for example, the Ingress resource can look for TLS certificates by reference to a Secret resource).

External Secrets Operator

Another approach is to use the External Secrets Operator, an open source Kubernetes controller that pulls secret data from external managers, like AWS Secrets Manager, HashiCorp Vault, or Google Secret Manager, and syncs it into native Kubernetes Secrets resources. This has the advantage that applications don't need to be modified, since they can consume secrets in the usual Kubernetes-native ways as environment variables or mounted files. The trade-off is that as native Kubernetes Secrets, they are stored in etcd, unlike with the Secrets Store CSI Driver, where they never touch the cluster's backing store.

Rotating Secrets in Kubernetes

When it comes to rotating secrets, there are two aspects to consider:

- Rotating the value of a Kubernetes Secret being passed to an application
- Rotating the keys in the EncryptionConfig resource used to encrypt Secret resources in etcd

If you're using plain Secret objects, you can update their values with `kubectl`, and then you will generally need to restart pods that use those secrets to get the application to use the new values. Secret managers (like Vault or AWS Secrets Manager) can make this process easier. If you're using the External Secrets Operator, secret values are updated automatically from the external manager, but applications still usually have to be restarted to pick up these new values (unless they are written to watch for changes).

Rotating the keys used for encrypting Secret resources is a multistep process that involves modifying the EncryptionConfig resource and restarting the API Server(s) at least twice:

1. Add the new key as a second entry in the EncryptionConfig resource.
2. Restart the API Servers to read the new EncryptionConfig. They have access to the new key and can use it for decryption if they encounter a resource that they can't decrypt with the old key.
3. Swap the keys so that the new key is the first entry in the EncryptionConfig resource.
4. Restart the API Servers to reread the EncryptionConfig, so they start using the new key for encryption.
5. Replace all the existing Secret resources so they are encrypted with the new key.
6. It's a good idea to update the EncryptionConfig to remove the old key.

In my experience, most enterprises choose a third-party commercial solution for secret storage, either from their cloud provider (such as the AWS Key Management System or its Azure or GCP equivalents) or from a vendor such as HashiCorp or CyberArk. These offer several benefits:

- A dedicated secrets management system can be shared with multiple clusters. Secret values can be rotated, irrespective of the life cycle of the application cluster(s).
- These solutions can make it easier for organizations to standardize on one way of handling secrets, with common best practices for management and consistent logs and auditing of secrets.

The public cloud providers all document their recommendations for encrypting Kubernetes secrets:

- AWS documentation for using Key Management Service with EKS
- Microsoft documentation for using Key Management Service with AKS
- Google documentation for using Cloud Key Management with GKE

Secrets Are Accessible by Root

Whether a secret is passed into a container as a mounted file or as an environment variable, it is going to be possible for the root user on the host to access it.

If the secret is held in a file, that file lives on the host's filesystem somewhere. Even if it's in a temporary directory, the root user will be able to access it. As a demonstration of this, you can list the temporary filesystems mounted on a Kubernetes node, and you'll find something like this:

```
$ sudo mount -t tmpfs
...
tmpfs on /var/lib/kubelet/pods/691c...5f8a/volumes/kubernetes.io~projected/
kube-api-access-lbgtt type tmpfs (rw,relatime,size=8024812k,noswap)
...
```

Using the directory names included in this output (which I'll refer to here as `$DIR`), the root user has no difficulty accessing the secret files held within them:

```
$ sudo ls -l $DIR
total 0
lrwxrwxrwx 1 root root 13 Jul 30 12:05 ca.crt -> ..data/ca.crt
lrwxrwxrwx 1 root root 16 Jul 30 12:05 namespace -> ..data/namespace
lrwxrwxrwx 1 root root 12 Jul 30 12:05 token -> ..data/token

$ sudo cat $DIR/ca.crt
-----BEGIN CERTIFICATE-----
MIIDBTCCAe2gAwIBAgIIeRWAgMJOFy4wDQYJKoZIhvcNAQELBQAwFTETMBEGA1UE
...
aRW9Jb5JqekJ
-----END CERTIFICATE-----
```

Extracting the secrets held in environment variables is almost as simple for the root user. Let's demonstrate this by starting a container with Docker on the command line, passing in an environment variable:

```
$ docker run --rm -it -e SECRET=mysecret ubuntu sh
$ env
...
SECRET=mysecret
...
```

This container is running `sh`, and from another terminal you can find the process ID for that executable:

```
$ ps -C sh
  PID TTY          TIME CMD
17322 pts/0    00:00:00 sh
```

In Chapter 4 you saw that lots of interesting information about a process is held in the `/proc` directory. That includes all its environment variables, held in `/proc/<process ID>/environ`:

```
$ sudo cat /proc/17322/environ
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=2cc99c98ba5aTERM=xtermSECRET=mysecretHOME=/root
```

As you can see, any secret passed into a container through the environment can be read in this way. Are you wondering whether it wouldn't be better to encrypt the secret first? Think about how you would get the decryption key—which also needs to be kept secret—into the container.

I can't overemphasize that anyone who has root access to a host machine has carte blanche over everything on that machine, including all its containers and their secrets. This is why it's so important to prevent unauthorized root access within your deployment and why running as root inside a container is so dangerous: since root inside the container is root on the host, it is just one step away from compromising everything on that host.

Summary

If you have worked through the book to this point, you should have a good understanding of how containers work, and you know how to send secret information safely between them. You have seen numerous ways in which containers can be exploited and many ways in which they can be protected.

The last group of protection mechanisms we shall consider relates to *runtime protection*, coming up in Chapter 15.

CHAPTER 15
Container Runtime Protection

In Chapter 10 you saw some approaches to strengthening container isolation using security profiles like seccomp, AppArmor, and SELinux. These all act at runtime, so they contribute to container runtime protection and limit the set of actions that containers can perform. In this chapter you will learn about more sophisticated security tools that provide more dynamic capabilities for detecting suspicious runtime activity, and use policies that can be tuned for individual workloads.

The term *security observability* refers to generating logs and metrics, along with tooling that helps teams understand security-relevant events that are happening in a deployment. Modern runtime security tools generate security observability data that includes container or Kubernetes identity information, making it much easier to correlate suspicious events to a specific containerized workload. Some tools can go further and selectively allow for policy *enforcement* that prevents suspicious activities from taking place.

This chapter starts by considering how we can take advantage of containerization to build workload-specific policies, and it lays out some of the types of behavior that tools could observe and restrict. Then we'll look at technology options for enforcing runtime behavior, and we'll consider some of the tools available.

One of the characteristics of containers is that they lend themselves to *microservice* architectures. Application developers can break down a complex software application into small, self-contained pieces of code that can be scaled independently, each delivered as a container image.

Breaking a large system into smaller components with well-defined interfaces makes it easier to design, code, and test the individual components. It also turns out to make them easier to secure.

Container Image Runtime Policies

If a given container image holds the code for an application microservice and that microservice does one small function, it's relatively easy to reason about what that microservice should do. The code for the microservice is built into a container image, and it's possible to construct a runtime profile or policy corresponding to that container image, defining what it should be able to do.

Every container instantiated from a given image should behave the same way, so it makes sense that a profile of expected behavior can be defined for an image and then used to police the behavior of all the containers based on that image.

In a Kubernetes deployment, runtime security might be policed on a pod-by-pod basis. A pod is essentially a collection of containers that share a network namespace, so the underlying mechanisms for runtime security are the same.

I'll use the same ecommerce platform example from "Container Firewalls and Microsegmentation" on page 163, with a product search microservice that accepts web requests specifying a search term (or the first few characters of a search term) as entered by a customer browsing the ecommerce site. The job of the product search microservice is to look in a product database for items that match the search term and return a response. Let's start by thinking about the expected network traffic for this microservice.

Network Traffic

From the description of the product search microservice, we can infer that its containers need to accept and respond to web requests coming from a particular ingress or load balancer, and they should initiate a database connection to the product database. Aside from common platform functions like logging or health checks, there is really no reason for this service to handle or initiate any other network traffic.

It would not be terribly onerous to draw up a network security policy defining the traffic that is permitted for this service and then use it to define rules that are enforced at the networking level, as you saw in Chapter 12. These rules could define that requests are only permitted inbound from the ingress/load balancer, and that the only egress traffic permitted is to the product database service. A Layer 7 policy might additionally define that these requests have to be HTTP GET requests.

Some security tools can act in a recording mode in which they monitor messages to and from a service over a period of time to automatically build up a profile of what normal traffic flow looks like. This profile can be converted into network policies.

Network traffic isn't the only behavior that you can observe and profile. Let's consider the executables.

Executables

How many executable programs should run in this product search microservice? For the sake of illustration, let's imagine that the code is written as a single Go executable called `productsearch`. If you were to monitor the executables running inside these product search containers, you should only ever see `productsearch`. Anything else would be an anomaly and possibly a sign of attack.

Even if the service is written in a scripted language like Python or Ruby, you can make inferences about what is or isn't acceptable. Does the service need to "shell out" to run other commands? If not, then were you ever to observe `bash`, `sh`, or `zsh` executables running in a product search container, you should be alarmed.

This relies on you treating the containers as immutable and assumes that you are not opening shells directly into containers on your production system. From a security perspective, there is very little difference between an attacker opening a reverse shell through an application vulnerability and an administrator opening a shell to perform some kind of "maintenance." As discussed in "Immutable Containers" on page 111, this is considered bad practice!

Any executables or dependencies that the application code needs should be included in that image. We discussed this earlier through the lens of vulnerability detection: you can't scan for vulnerabilities in code that isn't included in the image, so you should make sure that everything you want to scan is included. Treating containers as immutable gives us another powerful option for detecting code injection at runtime: *drift prevention*. Whenever a container starts to run a new executable process, this should only be permitted if the file being executed existed in the image when it was scanned. By using file fingerprints from the scanning step, rather than a list of filenames, this approach can prevent an attacker from trying to disguise an injected executable as a legitimate one.

Most of the time, we run programs by executing a program written in a file, but there is also something called *fileless execution*. Program instructions are held in memory and never written to disk. This technique is often used in malware attacks to evade detection by anti-malware tools that scan files looking for malware signatures. Modern runtime security tools can spot this execution from memory as well as from files.

File Access

Our hypothetical product search microservice probably does very little or no file access, other than writing logs and perhaps reading secrets so that it can authenticate itself to the product database service.

But this only takes into account the files that the microservice developer wrote code to access explicitly. Depending on the language that the service is written in, there could be several shared libraries and files accessed at runtime. For example, take a simple executable like `cat` and use the `opensnoop` tool to see how many files it tries to access:

```
PID    COMM               FD ERR PATH
45883  cat                 3   0 /etc/ld.so.cache
45883  cat                 3   0 /lib/aarch64-linux-gnu/libc.so.6
45883  cat                 3   0 /usr/lib/locale/locale-archive
... over 20 more file access attempts ...
45883  cat                -1   2 /usr/lib/locale/C.UTF-8/LC_CTYPE
45883  cat                 3   0 /usr/lib/locale/C.utf8/LC_CTYPE
45883  cat                 3   0 myfile
```

A policy that limits the set of files that a container can access will need to take into account all these ancillary files. This list is sufficiently long that even an experienced programmer might omit a few of these files if they tried to draw up a policy by hand, but some security tools offer the ability to profile running containers automatically and then alert on or prevent opening files outside the expected profile.

Another more generic approach is to define a set of files that are considered "sensitive" and shouldn't be accessed (or at least written to) by any workload. It makes sense to log all access to certain files, such as the `/etc/shadow` file, which holds password information—you might even prefer to block attempts to write to this file. As another example, in a Kubernetes environment, you should protect the `/etc/kubernetes/manifests` directory from unexpected write access to prevent unwanted static pods being created.

User and Group IDs

As discussed in Chapter 6, you can define the user ID under which processes run within a container, so this is another aspect that can be policed by security tools at runtime. (I hope you're using non-root users in your application profiles—see "Containers Run as Root by Default" on page 143.)

As a general rule, if the container is doing one job, it probably needs to operate under only one user identity. If you were to observe the container using a different identity, this would be another red flag. If a process were to be unexpectedly running as root, this privilege escalation would be an even greater cause for concern.

You can use scanning and admission control to control the user ID that a container is supposed to use. For defense in depth, use a runtime security tool that can also spot when a process attempts to change the user or group that it is running under.

AI for Generating Runtime Policies

When I wrote the first edition of this book, the idea of asking an AI to analyze a piece of software and write a runtime security profile for it would have been science fiction. As I am writing the second edition of this book, the use of LLM-based coding tools is exploding. At this moment in time, the results of using AI are still very mixed but promising and improving fast. It seems highly likely to me that AI-driven tools will soon be able to work out what executables, file access, privileges, and network connections are required for a container image to run successfully—I would not be surprised if those tools already exist by the time you're reading this page!

There are already companies promising autonomous segmentation and distributed exploit protection, with policies generated automatically based on observed behaviors and on security research into vulnerabilities.

Whether profiles and policies are generated by hand or automatically, you'll need a tool that can spot when a containerized process does something that is outside of its expected behaviors. Let's now think about the technology options that might be able to detect, and even prevent, out-of-policy activity.

Technology Options for Runtime Security

There are several different technology options that have been used over the years for building runtime security tooling.

LD_PRELOAD

In "File Access" on page 212, you saw that running `cat` causes several files to be opened. If you turn back to examine the output I showed, the first file is `/etc/ld.so.cache`. This file is a cache of *dynamic libraries*—dependencies that get pulled in for use by an executable when it starts running. `LD_PRELOAD` is a long-standing technique where a tool injects custom code that gets used in place of the normal shared dynamic libraries.

A runtime security tool might use `LD_PRELOAD` to add in instrumentation that gets called first, before the standard library code is called by an application. A typical approach is to override functions in the standard C library, including system calls.

Unfortunately there are several drawbacks to this approach:

- It only works on dynamically linked applications, not static ones. A malicious actor can compile their payload as a standalone executable and walk straight past any preload-based applications. Your own executables might be standalone binaries, too, especially if they are written in Rust or Go, so preload-based observability tooling will be completely blind to their activities.
- A malicious application can modify `LD_PRELOAD` settings to bypass tooling that uses it.
- Especially in a containerized environment, different applications might be using different standard C library implementations (`glibc` or `musl`, for example) and different or custom versions of those libraries. It's not easy to write `LD_PRELOAD` wrappers that are robust to all these variants.

The `LD_PRELOAD` technique can perform well, and it formed the basis for early commercial container security tools. But as better options like eBPF emerged, and with the proliferation of Go and Rust applications, `LD_PRELOAD` no longer seriously stands up as a robust technique for modern security tooling.

Ptrace

`ptrace` is a Linux syscall that allows one process to observe and even control another. It's commonly used as the basis for debugging and tracing tools, and it could easily observe, say, all syscalls that each user application makes. That sounds like a great foundation for security tooling, right? Unfortunately, there are problems with this approach too:

- Every breakpoint that `ptrace` catches and acts on triggers a context switch from the traced to the tracing process. This adds a painful amount of overhead if you want to trace any significant amount of activity.
- A traced process can easily detect whether it is being traced, and malicious actors can take advantage of that knowledge.
- It's not difficult to use `ptrace` to extract application data from memory or to inject malicious code, so there is a dedicated `CAP_SYS_PTRACE` capability used to restrict its use, which is not usually granted by default to containers because of the security risk. If you allow this capability, you would be wise to have other defenses in place to prevent malicious processes from abusing its power. (By default, `ptrace` won't work in rootless containers because `CAP_SYS_PTRACE` isn't granted.)
- Many Linux distributions limit `ptrace` so that it can attach only to direct child processes. You can tell if this limitation is in place by looking for a value of 1 in the file `/proc/sys/kernel/yama/ptrace_scope`.

Though `ptrace` has many legitimate uses for debugging today, it's not a serious contender for container security tooling.

Several years ago I did a talk called "Debuggers from Scratch" that demonstrates how `ptrace` can be used to place break points for debugging purposes.

Seccomp, AppArmor, and SELinux

You met these technologies in Chapter 10, as approaches for strengthening container isolation, and they have their place in a defense-in-depth approach to runtime security. But they fall short in comparison to modern runtime security tooling:

- These approaches all rely on predefined, static policies with limited filtering capabilities: seccomp can filter syscalls by number and with very basic argument matching, while AppArmor and SELinux apply access control based on filesystem paths or type labels. Because they weren't designed with containers in mind, it's hard to express policies like "allow this action in container A, but not container B."
- They don't offer security observability—at best, a log will tell you that an action was blocked. There's no correlation to container or Kubernetes identities, and the logs generated provide no context, so they are of limited use for incident response.
- A process running in a container can stay within its static profile and still be malicious. Privilege escalation, fileless execution, or lateral movement are practically impossible to spot with these tools.

One strength that these techniques all share is that they all run in the kernel, from whence they have access to all containers running on a machine. Let's consider the advantage of using the kernel as a vantage point to observe containers.

Kernel-Based Runtime Security

In "Container Processes from the Host Perspective" on page 54, you saw that all the containers on a host machine share a single kernel. Applications run in unprivileged user space, and make syscalls requesting assistance from the kernel whenever they need to access a file, send a network message, or allocate memory. The kernel is also responsible for coordinating processes, creating new ones, and both checking and giving them privileges.

All of these actions are interesting from a security perspective. Runtime security tooling might have policies about whether individual containers can access certain files,

open network connections to certain locations, or request additional Linux capabilities. Since all these actions involve the kernel, the kernel is an ideal place for runtime security tooling to live (see Figure 15-1).

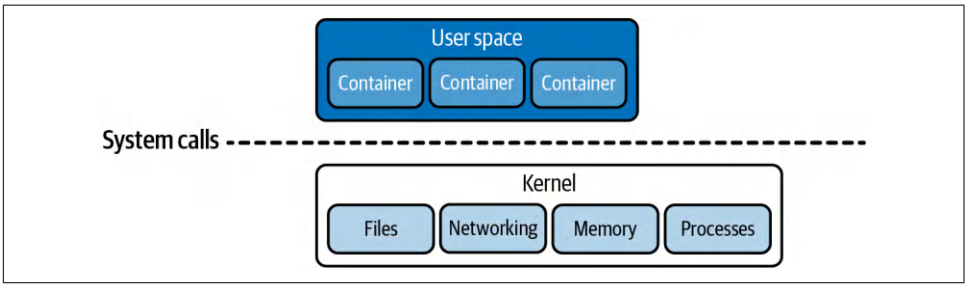


Figure 15-1. Containerized applications share a single kernel

As illustrated in Figure 15-1, containers share a single kernel that coordinates processes, checks permissions, and handles operations including file, network, and memory access. The kernel offers an ideal vantage point for security tools, with visibility over all the containerized applications that are running on that host. What's even better is that kernel-based tooling can have a life cycle that is entirely independent of the containers it controls. This is a massive advantage over the sidecar model that you met in Chapter 12 in relation to service mesh.

Historically, the only way to extend the kernel was to use kernel modules. The main concern about kernel modules is that they can be brittle, and a kernel module crash will bring down the whole machine it's running on. That's true of the kernel itself, but typically the kernel will have been through a *lot* of hardening and testing before it reaches a Linux distribution that's being deployed in many production environments. To give you an idea, Red Hat Enterprise Linux version 9 is widely in use as I write this text in 2025, and it's based on a 5.14 kernel released in 2021. Any given kernel module would be used by a tiny fraction of all Linux users, so there's a significantly greater chance that it still has bugs that could cause it to crash. Many organizations are understandably reluctant to use kernel modules or ban their use altogether.

However, in recent years, eBPF has enabled the development of tools that run in the kernel but that can't cause the kernel to crash because, as you learned in Chapter 11, eBPF programs are verified as they are loaded. Let's consider its use in runtime security tools.

eBPF for Runtime Security

eBPF allows a developer to create programs that can be loaded into the kernel dynamically and attached to kernel events. Those eBPF programs can be used to report events back to user space, and in recent years, eBPF has evolved to allow security tools that can block events or terminate malicious processes.

Perhaps the most basic example of runtime security observability would be to observe programs as they are executed. Let's consider an `nginx` container. Under normal circumstances, the only processes you expect to see inside such a container are `nginx` processes. There are several tools that can show processes as they are launched using eBPF technology, which allows custom programs to run within the kernel.

For this example, we'll use `execsnoop` from the `libbpf-tools` package. (You met `capable` from the same package back in Chapter 11.) You'll need to run this as root to have the required Linux capabilities that allow inserting eBPF code into the kernel:

```
$ sudo execsnoop
```

After starting `execsnoop` in one terminal, I run an `nginx` container from another:

```
$ docker run --rm -d --name nginx nginx
```

In the first terminal, you'll see the output from `execsnoop` showing lots of new processes that start as a result of running the `nginx` container. It might be a fun diversion to consider what some of these processes are, especially knowing what you know from Chapter 4 about how containers are created. Here's the output (edited a little for clarity):

```
PCOMM            PID    PPID   ARGS
docker           43495  38722  /usr/bin/docker run --rm -d --name nginx nginx
containerd-shim  43505  744    /usr/local/bin/containerd-shim-runc-v2          ❶
    -namespace moby -address /run/containerd/containerd.sock -publish-binary
    /usr/local/bin/containerd -id 27bfa...e2357 start
containerd-shim  43513  43505  /usr/local/bin/containerd-shim-runc-v2
    -namespace moby -id 27bfa...e2357 -address /run/containerd/containerd.sock
runc             43524  43513  /usr/local/bin/runc ... create                  ❷
    --bundle /run/containerd/io.containerd.runtime.v2.task/moby/27bfa...e2357
    --pid-file /run/containerd/io…task/moby/27bfa...e2357/init.pid 27bfa...e2357
...
iptables         43543  3127   /usr/sbin/iptables --wait -t raw -C PREROUTING  ❸
    -d 172.17.0.3 ! -i docker0 -j DROP
iptables         43545  3127   /usr/sbin/iptables --wait -t raw -A PREROUTING
    -d 172.17.0.3 ! -i docker0 -j DROP
...
runc             43555  43513  /usr/local/bin/runc ... start 27bfa...e2357     ❹
docker-entrypoi  43537  43513  /docker-entrypoint.sh nginx -g daemon off;      ❺
find             43561  43537  /usr/bin/find /docker-entrypoint.d/ ...         ❻
...
nginx            43537  43513  /usr/sbin/nginx -g daemon off;                  ❼
```

Here's the `docker` command I just ran in the second terminal:

❶ This `containerd-shim` command is a child of whatever process has the ID 744 (on my system when I ran this example—if you try this for yourself, your process IDs will vary!). By running `ps 744` I learn that this parent process is `containerd`. The `containerd-shim` process acts as an intermediary between the `containerd`

daemon and `runc`, allowing the container process to keep running even if `containerd` crashes or is restarted.

❷ Here is the `runc create` command that creates the container from the specified bundle. Looking inside that `/run/containerd/io.containerd.runtime.v2.task/moby/27bfa...e2357` directory, I can see an OCI container bundle, with a root filesystem and a *config.json* file. Running `cat` to see the contents of the *init.pid* file parameter showed me the process ID 43537, which we'll see again in a moment.

❸ Process 3127 is the Docker daemon on this system, and we can see that it is setting up `iptables` rules for the new container. It won't surprise you to learn that `docker inspect nginx` showed me that the IP address for this container is 172.17.0.3.

❹ Now `runc start` makes the container start running…

❺ …launching the `docker-entrypoint.sh nginx` command. I can see that this command corresponds exactly to what is specified in `process.args` inside the container's *config.json* file, and you can also see that this has the process ID 43537 that I found in the *init.pid* file.

❻ The `docker-entrypoint.sh` script starts with a `find` command that locates all the nginx configuration scripts it should run. For me this caused 16 more processes to execute…

❼ …before finally running the `nginx` daemon.

There are two key takeaways from this example. The first is to note that `execsnoop` running on the host machine is capable of seeing processes running inside containers. Even though the container is isolated using namespaces and cgroups, it is still using the host's kernel, so eBPF-based tooling that instruments the kernel has access to the container too. This is one of the reasons eBPF is such a compelling technology for building infrastructure tools for containerized environments.

The second takeaway is that there is an initial period where initialization processes run inside the container, but after a while, we would only expect to see `nginx` running. This is a common pattern and worth bearing in mind when thinking about what events a runtime security policy should report. In this case, `find` is needed during initialization, but the same `find` executable running at a later point might be an indication that a bad actor is looking for something within the container's filesystem.

Now if I execute another command inside the container—for example, `docker exec -it nginx ls`—the executable shows up in `execsnoop`'s output:

```
ls               43889  43877    0 /usr/bin/ls
```

Imagine that, as an attacker, I insert a cryptocurrency miner inside one of your containers. When the miner executable starts, you'll want that to be observed and reported (and possibly even prevented), but you don't want to be notified when `nginx` is going through its initialization or running its usual `nginx` executable. Runtime security tools can carry out this kind of observation, spotting when executables are launched and comparing the executable against a policy for the workload that is supposed to be running.

So far this chapter has talked about the kind of policies that might define whether a container is behaving normally or suspiciously. Now let's consider some of the runtime security tools that can report on or prevent suspicious behavior in a container.

Container Runtime Security Tools

For the reasons described earlier in this chapter, today's container-aware runtime security tools are based on eBPF. In the open source world, Falco is probably still the most widely adopted at the time of writing, but the Tetragon security tool from the Cilium project is rapidly gaining ground. Tracee also warrants a mention, and Inspektor Gadget can also be used for Kubernetes-aware security observability. There are also several commercial runtime security options available.

Falco

Falco is a graduated project from the CNCF, and as you can see from Figure 15-2, it has options to collect events using eBPF or with a kernel module.

Falco has a large set of community-driven policies, known as *rules*—for example, to detect events that breach NIST or PCI/DSS guidelines. In a Kubernetes deployment, events are enriched with information like pod and container names, labels, annotations, and Kubernetes namespaces, allowing the events to be correlated to the workload responsible.

One notable point to draw from the Falco diagram is that although events are collected in the kernel, they are parsed and compared against rules in user space.

There is no enforcement capability in Falco other than to have a user space tool react to a rule violation—for example, to kill the offending process.

The company that originally created the Falco project, Sysdig, provides an enterprise-ready version in its Sysdig Secure product, including a UI and policy management capabilities.

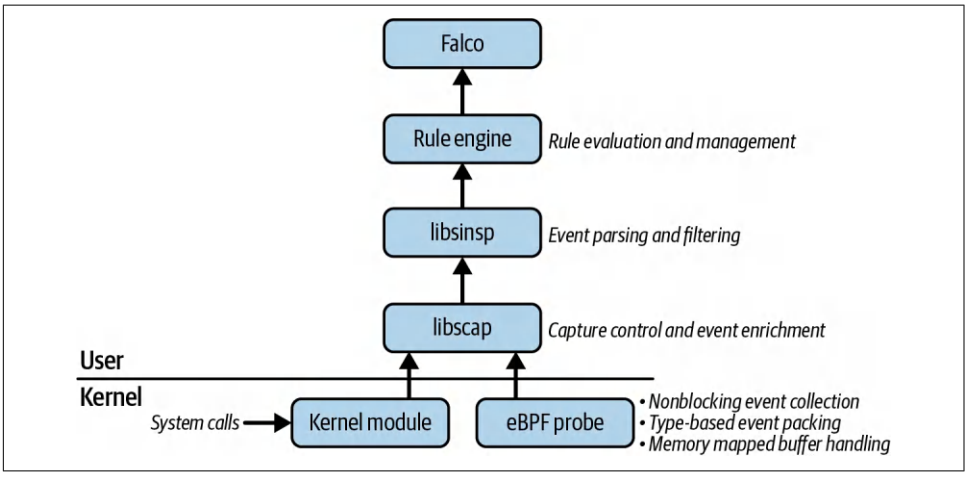


Figure 15-2. Falco components[1]

Cilium Tetragon

Tetragon is a runtime security tool that emerged from the Cilium CNCF project, which shares a long history of codevelopment with eBPF itself. Similar to Falco, Tetragon generates event information enriched with Kubernetes identity information but also includes timing details and information about process ancestry, which is invaluable for forensic investigations to determine how a security event came to take place.

In addition to providing runtime security observability, Tetragon takes advantage of some advanced eBPF capabilities.

In-kernel filtering

Tetragon has a userspace agent for management and to collect event data and forward it to an event database or security information and event management (SIEM). But in contrast to Falco, comparison with policies all takes place within the kernel. For example, you might want a policy that lets you know when certain files are accessed. With Tetragon, the comparison against the file names that you're interested in happens within the kernel so that only notable events make it as far as user space. This makes for significantly better performance.

1 Source: adapted from an image by Falco.

LSM API

Tetragon rules can take advantage of the Linux Security Module (LSM) API. This is the same API that tools like AppArmor use to enforce static security policies, but when it's used with eBPF, it can be used to apply more complex, dynamic policies (for example, so that they apply to new containers as they are created).

Enforcement

Tetragon policies can define whether to report on an event, override a return value to provide enforcement, or even kill the process responsible.

Tetragon policies are written as Kubernetes custom resources in YAML files. This makes them easy to manage in a Kubernetes environment, but Tetragon also works on bare-metal or virtual machines or with non-Kubernetes containers. Here's an example policy that uses SIGKILL for enforcement:

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: "example"
spec:
  kprobes:
  - call: "security_file_permission"                    ❶
    syscall: false
    args:                                               ❷
    - index: 0
      type: "file"
    - index: 1
      type: "int" # 0x04 is MAY_READ, 0x02 is MAY_WRITE
    selectors:
    - matchArgs:                                        ❸
      - index: 0
        operator: "Equal"
        values:
        - "/tmp/liz"
      matchActions:                                     ❹
      - action: Sigkill
```

❶ This policy attaches to a kernel function called `security_file_permission()`, which is part of the LSM API and is called by the kernel every time a process wants to access a file for reading or writing. It's used to control whether the operation should be permitted or not.

❷ There are two arguments to this function. The first is a pointer to a kernel file data structure, and the second is a value indicating whether the file is looking for read or write access.

❸ The policy defines some *selectors* that are used to filter events. In this case, Tetragon will only act on the event if the file indicated by the first parameter has the name `/tmp/liz`.

❹ If the *selectors* match, Tetragon sends a SIGKILL to terminate the responsible process (and also reports the event to user space).

When an application, possibly compromised, attempts to access this file, a Tetragon eBPF program in the kernel is triggered. Without any transition to userspace, eBPF code compares the event to the policy, and if all the conditions are met, it sends the SIGKILL signal. This ensures that the out-of-policy action is never able to complete. Figure 15-3 illustrates this sequence of events.

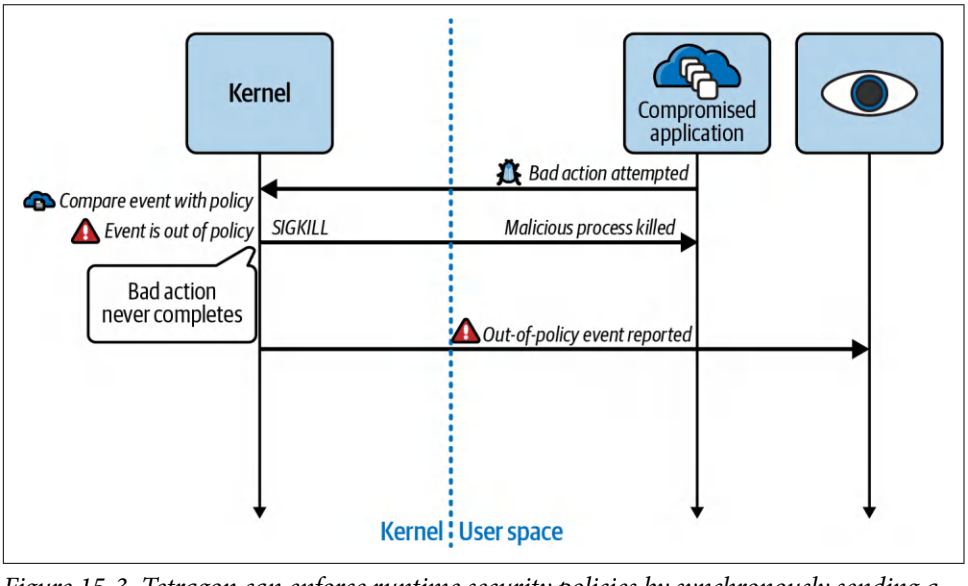


Figure 15-3. Tetragon can enforce runtime security policies by synchronously sending a SIGKILL signal from the kernel

Tetragon also generates Prometheus-format metrics that can be visualized with a tool like Grafana or Splunk to easily spot whether and when issues have taken place or to identify which workloads are affected.

My employer, Isovalent (now part of Cisco), provides an enterprise distribution of Tetragon.

Tracee

Tracee is another eBPF-based open source project from my former employer, Aqua Security. Like Falco, it provides detection but not enforcement, and like Tetragon, it provides some in-kernel filtering, though with less sophisticated filtering capabilities.

Inspektor Gadget

Inspektor Gadget is another CNCF project, providing a collection of eBPF-based "gadgets" for observing and debugging Kubernetes workloads. It doesn't provide detection or enforcement capabilities like Tetragon or Falco, but it does have some useful gadgets that can be useful for security purposes, including:

- Execution monitoring
- Network tracing
- Syscall tracing
- Capabilities monitoring
- File access auditing

You can attach gadgets to a running pod to observe what files are being accessed or which binaries are being executed, and this could be useful for investigating a live incident. You could also use gadgets to build a baseline of expected behavior from a workload and use this to help build a Tetragon policy, Falco rule, or seccomp policy.

Prevention or Alerting

Whichever tool you use for runtime protection, there is one last aspect to consider: what action do you want the tool to take when it finds potentially anomalous behavior? Do you want to prevent that action, terminate the process, or just alert on the event and let a human (or an AI) decide what to do? This can be a complicated question, and the answer may well depend on the workloads involved:

- If you automatically stop a process or delete the container when it triggers an alert, will this affect the service for users? Are there multiple instances that can take over? What if it's a false positive?
- If you're relying on an orchestrator to bring up a new instance, what if the new instance is subject to the same attack? You can end up in a vicious cycle in which a container comes up, bad behavior is detected, and the security tool kills the container, only for it to be re-created by the orchestrator (for example, think about how Kubernetes will create or destroy pods to ensure that the number matches the desired replica count).
- If this is a new version of a container, can you roll back to the previous version?

There is no single correct answer when it comes to figuring out how to handle a security alert automatically. However long it takes *might* be long enough for an attacker to cause harm. Suppose you have an eBPF-generated event detecting that a sensitive file is being accessed. Tetragon's synchronous enforcement can stop that access before it happens, but with other tools, you have to wait for a user space component to kill the responsible process, and that might be long enough for some sensitive data to be exfiltrated. That risk is even greater if a human is consulted before action is taken. Prevention is much better than a cure in this regard.

If your security tools can actually prevent bad behavior within a container before it happens, there is a possibility that the container can carry on operating as before. For example, suppose an attacker has compromised a product search container and is attempting to run a cryptocurrency miner. The executable is not part of the profile, so the runtime security tool prevents it from being run at all. The "good" processes carry on as normal, but the cryptomining attack is prevented.

It's quite likely that you want different responses for different types of potentially malicious activity, because the consequences of preventing the activity might vary. For example, dropping a network packet as part of enforcing a network policy simply means that the receiving piece of software never sees it, so it can't[2] trigger a bug in that code. On the other hand, let's consider a runtime security tool that can prevent file access, returning an error code when access is attempted. Ideally all applications would be able to gracefully handle the error, but it's possible that the developers didn't anticipate it, and so the application crashes. You might quickly spot this during testing, but it might be missed if the file access is in some rarely exercised code path. With this in mind, you might rely on logging/alerting for most file access, and only block access to the most sensitive of files where a possible application crash would have lesser consequences than malicious access to the file.

When suspicious or malicious events occur, you might get peace of mind by knowing they have been prevented, but it's even better if you can determine whether there was a deliberate attack, and if so, how it took place. You'll need forensic evidence leading up to the suspicious activity. The eBPF tools you have met in this chapter can be invaluable for recording relevant data (for example, Tetragon's execution and file event audit trail, or Tracee in logging mode) and sending it to persistent storage. You might also want to consider quarantining, rather than terminating, the offending container.

2 "Can't" is a strong term, and theoretically there could be poorly written code that falls over if it doesn't receive a message within a certain time frame, but I hope that a bug like that would be fixed before it gets anywhere near production.

Quarantining a Container

Rather than killing a suspicious container altogether, you can pause it to preserve state information that could be useful for forensics. Using Docker, you might do this as follows:

- `docker pause` freezes the container, stopping all its processes without killing them.
- Optionally, you might use `docker commit` to capture a copy of the container's filesystem for later analysis.
- If there are volumes mounted into the container, you probably want to take a snapshot of those too.

Kubernetes doesn't provide a native mechanism for pausing pods, but some techniques you can consider include:

- Using network policy to isolate the pod
- Using `kubectl debug` to attach to the pod for investigations
- Forensic container checkpointing using the Kubelet Checkpoint API (currently in beta at time of writing)
- Tainting the node where the suspicious pod is running, preventing new pods from being scheduled to it

It's generally a good idea to at least pause and/or isolate the container if you think it is compromised so that the attacker has less opportunity to exfiltrate data or move laterally to other workloads.

Vulnerability Mitigation

New software vulnerabilities get published all the time, and even if a fix is available straightaway, it might take some time to get container images rebuilt, tested, and deployed. In the meantime, it is really helpful if your runtime security tool can monitor whether the vulnerability has been exploited in your deployment or even prevent the exploit. eBPF-based runtime security tools like Tetragon make this possible.

The key to this is eBPF's ability to load programs and configuration into the kernel dynamically. This means you can roll out a new policy across your deployment, and it starts protecting all workloads (containerized or otherwise) immediately and transparently.

What should be in the policy depends on the vulnerability. As an example, let's consider the 2024 xzutils backdoor, a vulnerability in a library called `liblzma` that can be triggered to allow remote execution via SSH. For the SSH daemon to run the (vulnerable) library code, it needs to map that library code into memory. The following Tetragon policy will generate output if it detects a vulnerable version of the library being accessed by the SSH daemon:

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: "cve-2024-3094-xz-ssh"
spec:
  kprobes:
  - call: "security_mmap_file"                    ❶
...
    args:
    - index: 0                                    ❷
      type: "file"
...
    selectors:
    - matchBinaries:                              ❸
      - operator: "In"
        values:
        - "/usr/sbin/sshd"
      matchArgs:
      - index: 0                                  ❹
        values:
        - "liblzma.so.5.6.0"
        - "liblzma.so.5.6.1"
      matchActions:
        - action: Post                            ❺
          rateLimit: "1m"
```

❶ The eBPF program that manages this policy is attached to the function `security_mmap_file()`, which is part of the Linux Security Module (LSM) API. You met some LSMs in Chapter 10, and using eBPF programs attached to this API allows for dynamic policies. This function gives an LSM—or an eBPF program—the chance to approve or deny the memory mapping of a file.

❷ The first argument to this function is the file being memory mapped.

❸ This `matchBinaries` selector in the policy tells Tetragon that the event is only of interest if the process that is doing the memory mapping is `/usr/sbin/sshd`, the SSH daemon.

❹ The event is also only of interest if the first argument—the file—matches one of the vulnerable versions of the library.

❺ If the event matches all the selector conditions, Tetragon will generate a report about it, rate-limited to one event per minute so as not to overwhelm the security team with data. (Tetragon could also block access rather than just reporting it, if preferred.)

Because the filtering (matching on the binary and filenames) is all performed by eBPF programs in the kernel, this policy has negligible impact on performance.

You can see this demonstrated during a talk I gave at ContainerDays.

Summary

Runtime security observability and enforcement has evolved tremendously since the first edition of this book, thanks largely to the evolution and adoption of eBPF. The ability to detect runtime events at a very granular level, as described in this chapter, makes specialist container security tooling a compelling prospect, especially since it can be done so transparently to the applications and with very little performance overhead.

You're closing in on the end of the book now. The final chapter reviews the top 10 security risks collated by OWASP and relates these risks to mitigations that are specific to containerized deployments.

CHAPTER 16

Containers and the OWASP Top 10

If you're in the security field, there's a good chance you have come across OWASP, the Open Web Application Security Project; perhaps you're even a member of a local chapter. This organization periodically publishes a list of the top 10 web application security risks.

While not all applications, containerized or otherwise, are web applications, this is a great resource for considering which attacks to be most concerned about. You'll also find great explanations of these attacks and advice on how to prevent them on the OWASP website. In this chapter, I'll relate the current top 10 risks[1] to container-specific security approaches.

Broken Access Control

This category relates to the abuse of privileges that may be granted unnecessarily to users or components. There are some container-specific approaches to applying least privilege to containers, as discussed in Chapter 11:

- Don't run containers as root.
- Limit the capabilities granted to each container.
- In Kubernetes, use RBAC to limit permissions.
- Use rootless containers, if possible.

You can also use runtime security tools, as discussed in Chapter 15, to limit the actions that can be taken by a container.

1 At the time of writing, the OWASP Top 10 was most recently published in 2021. An update is expected in late 2025, so look out for an updated version of this chapter in the online edition of this book.

These approaches can limit the blast radius of an attack, but none of these controls relates to user privileges *at the application level*, so you should still apply all the same advice as you would in a traditional deployment.

Cryptographic Failures

It is particularly important to protect any personal, financial, or other sensitive data that your application has access to. Whether containerized or not, sensitive information should always be encrypted at rest and in transit, using a strong cryptographic algorithm. Over time, as processing power increases, it becomes feasible to brute-force encryption, which means that older algorithms can start to be considered no longer safe to use. Additionally, as quantum computing becomes more effective, there is a concern that some widely used algorithms like RSA and elliptic curve cryptography will be rendered useless. Make sure you're keeping abreast of developments and using up-to-date algorithms as advised by authoritative organizations like NIST or the UK's National Cyber Security Centre (NCSC).

Because the sensitive data is encrypted, your applications will need credentials to access it. Following the principles of least privilege and segregation of duties, limit credentials to only those containers that really need access. See Chapter 14 for coverage of safely passing secrets to containers.

Depending on your use case, you may want to ensure that network traffic between containers is encrypted, as discussed in Chapter 13.

Consider scanning container images for embedded keys, passwords, and other sensitive data.

Injection

If your code has an injection flaw, an attacker can get it to execute commands masquerading as data. This is perhaps best illustrated through the immortal *xkcd* character Little Bobby Tables.

There is nothing container-specific about this, though container image scanning can reveal known injection vulnerabilities in dependencies. You should review and test your own application code, following the OWASP advice.

Insecure Design

If a system design is flawed from a security perspective, even a perfect implementation of that design will have insecurities. Containerized systems built from microservices arguably have an advantage over monoliths because they break the system into constituent components, and it is likely going to be easier to reason about the threat

models for each of these individual building blocks one at a time. If you're using commonly adopted tools like Kubernetes, you can also take advantage of the latest advice on best security practices (for example, implementing zero-trust networking between containers and ensuring least privilege access).

Security Misconfiguration

Many attacks take advantage of systems that are poorly configured. Examples highlighted in the OWASP Top 10 include insecure or incomplete configurations, open cloud storage, and verbose error messages containing sensitive information, all of which have mitigations specific to containers and cloud native deployments:

- Use guidelines like the Center for Internet Security (CIS) Benchmarks to assess whether your system is configured according to best practices. There are benchmarks for Docker and Kubernetes, as well as for the underlying Linux host. It may not be appropriate in your environment to follow every recommendation, but they are good starting points for assessing your installation. Best security practices are often baked into managed Kubernetes services by default.
- If you are using a public cloud service, you should check your configuration settings for things like publicly accessible storage buckets or poor password policies. Gartner refers to these checks as Cloud Security Posture Management (CSPM), and an internet search for this term will reveal numerous vendors for these tools to automate these checks. There are open source tools such as Cloud Custodian, Prowler, and CloudSploit.
- As discussed in Chapter 14, using environment variables to convey secrets can easily result in them being exposed via logs, so I encourage you to use environment variables only for information that isn't sensitive.
- If you're using a public container registry like Docker or Quay, ensure that access to individual images is controlled properly. A private container registry may be a more secure alternative, though access should of course be secured.
- Use network policies to restrict network traffic appropriately, as discussed in Chapter 12.

You might also want to consider the configuration information that forms part of each container image under this OWASP category. This was covered in Chapter 6, along with best practices for building images securely.

Vulnerable and Outdated Components

I hope that by this stage in the book you can anticipate my advice on vulnerable and outdated components: use an image scanner to identify known vulnerabilities in your container images. Where you are building images yourself, use minimal base images to reduce the attack surface.

You also need a process or tooling in place to:

- Rebuild container images to use up-to-date, fixed packages.
- Identify and replace running containers based on vulnerable images.

Consider runtime security tooling that can mitigate vulnerabilities before you can apply a fix, as discussed in Chapter 15.

Identification and Authentication Failure

This category covers broken authentication and compromised credentials. At the application level, all the same advice applies for containerized apps as for monoliths in traditional deployments, but there are some additional container-specific considerations:

- The credentials required by each container should be treated as secrets. These secrets need to be stored with care and passed into containers at runtime, as discussed in Chapter 14.
- Breaking an application into multiple containerized components means that they need to identify each other and communicate using secure connections. This can be handled directly by containerized application code, or you can use a service mesh or transparent encryption to offload this responsibility. See Chapter 13.

Software and Data Integrity Failures

This category covers supply chain security failures. As discussed in Chapter 7, best practices include using signed container images, verifying SBOMs, and securing your CI/CD build pipelines.

This category also covers failing to validate data integrity. One example is insecure deserialization, where a malicious user provides a crafted object that the application interprets to grant the user additional privileges or to change the application behavior in some way. (I witnessed an example of this myself back in 2011 as a Citibank customer, when Citi had a vulnerability allowing a logged-in user to access other people's accounts simply by modifying the URL.)

An application's handling of data is generally not something that is affected by whether it is running in containers or not.

Security Logging and Monitoring Failures

IBM's Cost of a Data Breach Report 2025 shares the terrifying statistic that, on average, breaches take 181 days to be identified and another 60 to be contained. It should be possible to dramatically reduce that with sufficient observation combined with alerting on unexpected behavior.

Use a centralized logging system such as Fluentd or ELK Stack, a SIEM, or your cloud provider's logging capabilities to ensure that logs from containerized applications are stored and not lost when a container is removed.

In any production deployment, you should be logging container events, including:

- Container start/stop events, including the identity of the image and the invoking user
- Access to secrets
- Any modification of privileges
- Modification of container payload, which could indicate code injection (see "Immutable Containers" on page 111)
- Inbound and outbound network connections
- Volume mounts (for analysis of mounts that might subsequently turn out to be sensitive, as described in "Mounting Sensitive Directories" on page 156)
- Failed actions such as attempts to open network connections, write to files, or change user permissions, as these could indicate an attacker performing reconnaissance on the system (see Chapter 15 on runtime security tooling)

Most serious commercial container security tools integrate with enterprise SIEM to provide container security insights and alerts through one centralized system. Even better than observing attacks and reporting on them after the event, these tools can provide the protection of not just reporting on unexpected behaviors but preventing them from happening based on runtime profiles, as discussed in Chapter 15.

Server-Side Request Forgery

In a container-specific OWASP Top 10, I think this category might get a higher ranking, because of the risks of obtaining control of the system by persuading an application to make requests to an internal metadata or control plane API. For example, an attacker might leverage an application SSRF vulnerability to make requests to the Kubernetes API, the cloud provider's metadata services, or a database service

accessible within the deployment. These services should be available only to trusted resources. Reduce the likelihood of this with network policies (see Chapter 12) and runtime monitoring (see Chapter 15).

Summary

The OWASP Top 10 is a useful resource for making any internet-connected application more secure against the most common types of attack.

Security needs to be baked into application code and into the infrastructure in which those applications run. I can't promise you perfect security, but by applying the advice in this book, you will be in a more secure position.

Conclusions

Congratulations on reaching the end of this book!

My first hope for you at this point is that you now have a solid mental model of what containers are. This will serve you well in discussions about how to secure your container deployments. You should also be armed with knowledge about different isolation options, should regular containers not give you enough isolation between workloads for your environment.

I also hope that you now have a good understanding of how containers communicate with each other and the outside world. Networking is a vast topic in its own right, but the most important takeaway here is that containers give you a unit not just of deployment but also of security. There are lots of options for restricting traffic so that only what is expected can flow between containers and to/from the outside world.

You've read about best practices for building container images and detecting known vulnerabilities or image tampering. You know why it's a good idea to treat containers as immutable, and you've seen how runtime security tools can spot, and even prevent, suspicious or malicious activities.

I'd imagine that you see how layered defenses will serve you well in the event of a breach. If an attacker takes advantage of a vulnerability in your deployment, there are still other walls they may not be able to breach. The more layers of defense, the less likely an attack is to succeed.

As you saw in Chapter 16, there are some preventative measures unique to containers that you can apply against OWASP's list of the most commonly exploited attacks against web applications. That top 10 list doesn't cover all the possible weaknesses in your deployment. Now that you have almost reached the end of the book, you might want to review the list of attack vectors specific to containers in "Container Threat Model" on page 3. You will also find a list of questions in the Appendix to help you assess where your deployment might be most vulnerable and where you should beef up your defenses.

I hope that the information in this book helps you to defend your deployment, come what may. If you are subject to an attack—whether you are breached or you succeed in keeping your application and data safe—I would love to hear about it. Feedback, comments, and stories about attacks are always welcome, and you can raise issues at *containersecurity.tech*. You can find me on LinkedIn, on GitHub, and on social media.

APPENDIX
Security Checklist

This appendix covers some important items you should at least think about when considering how best to secure your container deployments. In your environment, it might well not make sense to apply *every* item, but if you have thought about them, you will be off to a good start. No doubt this list is not absolutely comprehensive!

- Are you running all containers as a non-root user? See "Containers Run as Root by Default" on page 143.
- Are you running any containers with the `--privileged` flag? Are you dropping capabilities that aren't needed for each image? See "The --privileged Flag and Capabilities" on page 153.
- Are you running containers as read-only where possible? See "Immutable Containers" on page 111.
- Are you checking for sensitive directories mounted from the host? Are they read-only where possible? How about the Docker socket? See "Mounting Sensitive Directories" on page 156 and "Mounting the Docker Socket" on page 157.
- Are you running your CI/CD pipeline in your production cluster? Does it have privileged access or use the Docker socket? See "The Dangers of Docker Build" on page 74.
- Are you generating signed images, with SBOMs and build attestations? Are you verifying signatures before you deploy containers? See Chapter 7 on supply chain security.
- Are you scanning your container images for vulnerabilities? Do you have a process or tooling in place for rebuilding and redeploying containers where the image is found to include vulnerabilities? See Chapter 8 on software vulnerabilities in images.

- Are you using a seccomp, AppArmor, or SELinux profile? The default Docker profiles are a good starting point; even better would be to shrink-wrap a profile for each application. See Chapter 10 on strengthening container isolation, and "Container Image Runtime Policies" on page 210.
- Do you have network policies restricting traffic between components? See Chapter 12 on container network security.
- Are you encrypting traffic between components? This could be implemented transparently by the network or by using a service mesh. See "Zero-Trust Networking" on page 192 and "Secure Connections Between Containers" on page 193.
- Are you using a modern runtime security tool to protect against malicious file access, executables, network traffic, or privilege escalation and to mitigate known vulnerabilities? See Chapter 15 on runtime security tools.
- What base image are you using? Can you use an option such as a scratch or distroless image? Can you minimize the contents of your images to reduce the attack surface? See "Dockerfile Best Practices for Security" on page 91.
- Are you enforcing the use of immutable containers? That is to say, are you making sure that all executable code is added to a container image at build time and not installed at runtime? See "Immutable Containers" on page 111.
- Are you setting resource limits on your containers? See "Cgroups for Containers" on page 28.
- Do you have admission control to make sure that only approved images can be instantiated in production? See "Admission Control" on page 103.
- Are you passing secrets into containers using a temporary filesystem? Are your secrets encrypted at rest and in transit? Are you using a secure secrets management system for storage and rotation? See Chapter 14 on passing secrets to containers.
- Are you using hosts exclusively for running containers, separate from other applications? Are you keeping your hosts systems up to date with the latest security releases? Consider running an OS specifically designed for container hosts. See "Container Host Machines" on page 56.
- Are your hosts and container configured according to security best practices such as the CIS Benchmarks for Linux, Docker, and Kubernetes? See "Security Misconfiguration" on page 231.
- Are you using GitOps? If so, are you following best practices to secure the Git repositories that hold your configuration files? See Chapter 9.

- Are you collecting logs remotely for later audit or forensic analysis in the event of a breach? Do those logs give details about how and when processes were executed? See Chapter 15.

All these questions are relevant in a Kubernetes deployment, but there are additional attack surfaces that are outside the scope of this book. Cloud provider–managed Kubernetes services also involve the use of an identity and access management infrastructure, which should be configured carefully. The Kubernetes documentation discusses security at length, Microsoft Kubernetes Threat Matrix provides a good overview, and AWS provides a security best-practice guide for EKS, a lot of which applies wherever your Kubernetes is hosted. You can also use a CSPM tool to run regular checks on the security settings on the underlying cloud infrastructure.

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

About the Author

Liz Rice is chief open source officer at Isovalent, the eBPF and network security specialists behind the Cilium project, and now part of Cisco. She has held many roles with the Cloud Native Computing Foundation (CNCF), including governing board member, chair of the technical oversight committee, and cochair and keynote speaker at KubeCon + CloudNativeCon. She is also the author of O'Reilly's *Learning eBPF* and the first edition of *Container Security*. When not writing or talking about code, Liz loves riding bikes in places with better weather than her native London and making music under the pseudonym "Insider Nine."

Colophon

The animal on the cover of *Container Security* is an armored catfish (family *Loricariidae*), also called loricariids, suckermouth catfish, or plecos after the species *Hypostomus plecostomus*. These fish are native to Costa Rica, Panama, and South America, where they inhabit freshwater streams and rivers. Armored catfish are highly adaptable and can flourish in a number of different environments: slow- and fast-moving currents, canals, ponds, lakes, estuaries, and even home aquariums.

There are more than 680 species of loricariids, all of which vary in color, shape, and size. Common traits include the flexible bony plates that distinguish them from other catfish, a flattened body, and a ventral suckermouth that allows these fish to feed, breathe, and attach themselves to various surfaces. They can grow anywhere from three inches to over three feet, depending on conditions. Armored catfish will eat any number of things, including algae, invertebrates, small bivalves, water fleas, worms, insect larvae, and detritus—one genus, *Panaque*, is known for eating wood. Parental care is common in loricariids, and many species will create long burrows along a shoreline where the female will deposit her eggs. Males guard the eggs until they hatch.

Armored catfish are nocturnal and nonmigratory, but they do have a tendency to disperse and potentially displace native fish populations when introduced to a new environment. In addition to their armored bodies and overall hardiness, armored catfish have also evolved modified digestive systems that can function as additional respiratory organs. If necessary, these fish can breathe air and survive out of water for more than 20 hours!

The cover illustration is by Karen Montgomery, based on a black-and-white engraving from *Shaw's Zoology*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.