O'REILLY®

What Is eBPF?

An Introduction to a New Generation of Networking, Security, and Observability Tools

Liz Rice

Compliments of
ISOVALENT
now part of CISCO

REPORT

ISOVALENT
now part of CISCO

Get the power of eBPF for your cloud native infrastructure - with Isovalent.

Harness the power of eBPF for cloud native networking, security, and observability with the Isovalent Enterprise Platform. Brought to you by the creators of eBPF, Cilium, and Tetragon, the Isovalent Enterprise Platform is perfectly suited for your Kubernetes and hybrid-cloud environments, ensuring secure, high-performant, and scalable connectivity.

The Isovalent Enterprise Platform provides the fastest, most reliable path to enterprise-grade Kubernetes networking and security and is selected by leading enterprises across all verticals.

Isovalent Enterprise Platform offers:

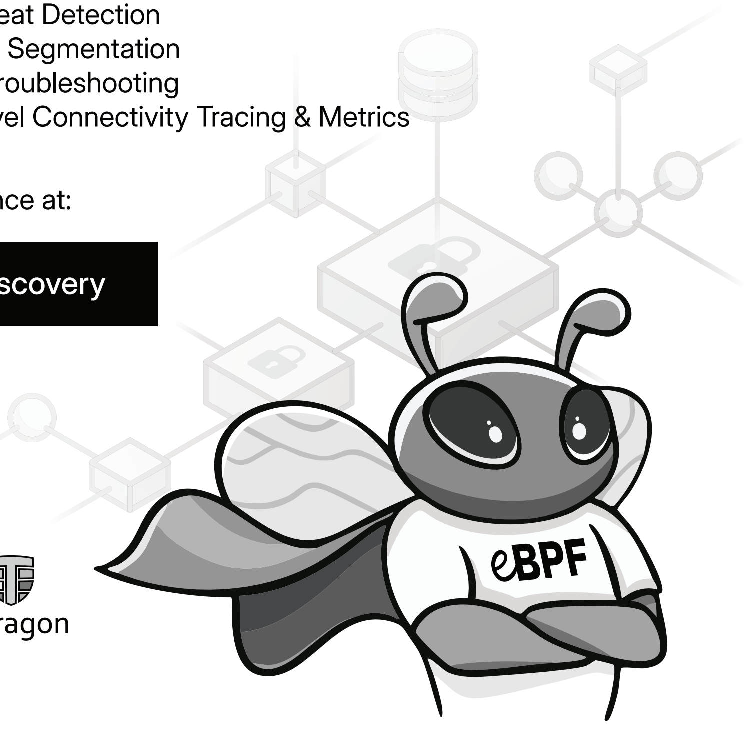
- Multi-Cloud, High Performance Networking
- Runtime Security & Observability
- SIEM Export & Threat Detection
- Multi-Tenant Micro Segmentation
- Fault Analytics & Troubleshooting
- Network & API-Level Connectivity Tracing & Metrics

Get hands-on experience at:

isovalent.com/discovery

Proud creators of

eBPF   cilium   tetragon



What Is eBPF?

An Introduction to a New Generation of Networking, Security, and Observability Tools

Liz Rice

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

What Is eBPF?

by Liz Rice

Copyright © 2022 O'Reilly Media. All rights reserved.

Printed in the United States of America.

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

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

Acquisitions Editor: John Devins
Development Editor: Rita Fernando
Production Editor: Kristen Brown
Copyeditor: nSight, Inc.

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

April 2022: First Edition

Revision History for the First Edition

2022-04-13: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *What Is eBPF?*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

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

978-1-492-09723-5

[LSI]

Table of Contents

1. Introduction. 1
Extended Berkeley Packet Filter 2
eBPF-Based Tools 2

2. Changing the Kernel Is Hard. 5
The Linux Kernel 5
Adding New Function to the Kernel 6
Kernel Modules 8
eBPF Verification and Security 9
Dynamic Loading of eBPF Programs 10

3. eBPF Programs. 13
Kernel and User Space Code 13
Custom Programs Attached to Events 14
eBPF Maps 16
Opensnoop Example 17

4. eBPF Complexity. 25
Portability Across Kernels 25
Linux Kernel Knowledge 27
Coordinating Multiple eBPF Programs 28

5. eBPF in Cloud Native Environments. 31
One Kernel per Host 31
eBPF Versus the Sidecar Model 32
eBPF and Process Isolation 33

6. **eBPF Tools** 35
    Networking 35
    Observability 39
    Security 41

7. **Conclusion** 47

CHAPTER 1

Introduction

In the last couple of years, eBPF has gone from relative obscurity to one of the hottest technology areas in modern infrastructure computing. Personally, I've been excited about the possibilities that eBPF enables ever since seeing Thomas Graf speak about it in a Black Belt session at DockerCon 17.[1] At the Cloud Native Computing Foundation (CNCF), my colleagues on the Technical Oversight Committee put eBPF forward as one of the areas to watch in our predictions of the technologies that would take off in 2021. Over 2,500 signed up for that year's eBPF Summit virtual conference, and several of the world's most advanced software engineering companies came together to create the eBPF Foundation. Clearly, there is a lot of interest in this technology.

In this short report, I hope to give you some insight into why people are so excited about eBPF and the capabilities it offers for tooling in modern compute environments. You'll get a mental model for what eBPF is and why it's so powerful. There are some code examples to help make it more concrete (but you can skip over these if you prefer). You'll get an understanding of what's involved when building eBPF-enabled tools, and why eBPF has become so seemingly ubiquitous in such a short period of time.

---

[1] Thomas Graf, "Cilium: Network and Application Security with BPF and XDP" (DockerCon 17, April 17–20).

Inevitably, in this short report there isn't room to go into all the details, but I'll leave you with some pointers for more information if you want to dive in more deeply.

Extended Berkeley Packet Filter

Let's get the acronym out of the way: eBPF stands for *Extended Berkeley Packet Filter*. From that name, you can see that its roots lay in filtering network packets, and the original paper[2] was written at the Berkeley Lab (Lawrence Berkeley National Laboratory). But (in my opinion) the name is not terribly helpful for conveying the true power of eBPF, as the "extended" versions enable so much more than packet filtering. These days, *eBPF* is used as a standalone name that encompasses more than its acronym suggests.

So, if it's not just about packet filtering, what is eBPF? eBPF is a framework that allows users to load and run custom programs within the kernel of the operating system. That means it can extend or even modify the way the kernel behaves.

As an eBPF program is loaded into the kernel, a verifier ensures that it is safe to run, and rejects it if not. Once loaded, an eBPF program needs to be attached to an event, so that whenever the event happens, the program is triggered.

eBPF was originally developed for Linux, and that is the operating system I'll focus on in this report; but it is notable that, as of this writing, Microsoft is developing an eBPF implementation for Windows.

Now that the Linux kernels in widespread use all have support for the "extended" parts, the terms *eBPF* and *BPF* are largely used interchangeably these days.

eBPF-Based Tools

As you'll see in this report, the ability to dynamically change the behavior of the kernel is tremendously useful. Traditionally, if we want to observe how our applications are behaving, we add code into those apps to generate logs and traces. eBPF allows us to collect customized information about how an app is behaving *without having to change the app in any way*, by observing it from within the kernel. We can build on this observability to create eBPF security tools that detect or even prevent malicious activity from within the kernel. And we can create powerful, high-performance networking capabilities with eBPF, handling network packets within the kernel and avoiding costly transitions to and from user space.

---

[2] Steven McCanne and Van Jacobson, "The BSD Packet Filter: A New Architecture for User-Level Packet Capture" (working paper, Lawrence Berkeley National Laboratory, Berkeley, December 19, 1992).

The concept of observing applications from the kernel's perspective isn't entirely new—it builds on older Linux features, such as perf,[3] which also collects behavior and performance information from within the kernel without having to modify the applications being measured. But these tools define a scope for the kinds of data that can be collected, and the formats in which the data is made available. With eBPF, we have far more flexibility because we can write entirely custom programs, allowing us to build a wide range of tools for different purposes.

eBPF programming is incredibly powerful, but it's also complex. For most of us, the utility of eBPF is going to come not from writing programs ourselves but from using tools created by others. There are an increasing number of projects and vendors building on the eBPF platform to create a new generation of tooling, covering observability, security, networking, and more.

I'll discuss some more of these higher-level tools later in this report, but if you're comfortable on the Linux command line and can't wait to see eBPF in action, a great place to start is the BCC project. It includes a huge collection of tracing tools; even just glancing at the list should give you some idea of the vast scope of operations we can instrument with eBPF, including file operations, memory usage, CPU stats, and even observing any bash command entered anywhere in the system.

In the next chapter, we'll look at why changing the kernel's behavior is useful, and why eBPF makes it vastly easier to do this than writing kernel code directly.

---

[3] *perf* is a Linux subsystem for collecting performance data.

CHAPTER 2

Changing the Kernel Is Hard

Since eBPF allows running custom code in the Linux kernel, let's make sure you're up to speed on what the kernel does. Then we can cover why eBPF changes the game when it comes to modifying how the kernel behaves.

The Linux Kernel

The Linux kernel is the software layer between your applications and the hardware they're running on. Applications run in an unprivileged layer called *user space*, which can't access hardware directly. Instead, an application makes requests using the system call (syscall) interface to request the kernel to act on its behalf. That hardware access can involve reading and writing to files, sending or receiving network traffic, or even just accessing memory. The kernel is also responsible for coordinating concurrent processes, enabling many applications to run at once.

As application developers, we typically don't use the system call interface directly, because programming languages give us high-level abstractions and *standard libraries* that are easier interfaces to program. As a result, a lot of people are blissfully unaware of how much the kernel is doing while our programs run. If you want to get a sense of how often the kernel is invoked, you can use the `strace` utility to show all the system calls an application makes. Here's an example, where using `cat` to read the word *hello* from a file and write it to the screen involves over 100 system calls:

```
liz@liz-ebpf-demo-1:~$ strace -c cat liz.txt
hello
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- -------------
  0.00    0.000000           0         5           read
  0.00    0.000000           0         1           write
  0.00    0.000000           0        21           close
  0.00    0.000000           0        20           fstat
  0.00    0.000000           0        23           mmap
  0.00    0.000000           0         4           mprotect
  0.00    0.000000           0         2           munmap
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         4           pread64
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         2         1 arch_prctl
  0.00    0.000000           0         1           fadvise64
  0.00    0.000000           0        19           openat
------ ----------- ----------- --------- --------- -------------
100.00    0.000000                   107         2 total
```

Because applications rely so heavily on the kernel, it means that we can learn a lot about how an application behaves if we can observe its interactions with the kernel. For example, if you are able to intercept the system call for opening files, you can see exactly which files any application accesses. But how could you do that interception? Let's consider what would be involved if we wanted to modify the kernel, adding new code to create some kind of output whenever that system call is invoked.

Adding New Function to the Kernel

The Linux kernel is complex, with around 30 million lines of code[1] at time of writing. Making a change to any codebase requires some familiarity with the existing code, so unless you're a kernel developer already, this is likely to present a challenge.

But you'll be facing a challenge that isn't purely technical. Linux is a general purpose operating system, used in different environments and circumstances. This means that if you want to make a change to the kernel, it's not simply a matter of writing code that works. It has to be accepted by the community (and more specifically by Linus Torvalds, creator and main developer of Linux) that your change will be for the greater good of all. This isn't a given—only one-third of submitted kernel patches are accepted.[2]

---

[1] "Linux 5.12 Coming in at Around 28.8 Million Lines,…" Phoronix (March 2021).

Let's suppose that you've figured out a good technical approach for intercepting the system call for opening files. After some months of discussion and some hard development work on your part, let's imagine that the change is accepted into the kernel. Great! But how long will it be until it arrives on everyone's machines?

There's a new release of the Linux kernel every two or three months, but even when a change has made it into one of these releases, it's still some time away from being available in most people's production environments. This is because most of us don't just use the Linux kernel directly—we use Linux distributions like Debian, Red Hat, Alpine, Ubuntu, etc., that package up a version of the Linux kernel with various other components. You may find that your favorite distribution is using a kernel release that's several years old.

For example, a lot of enterprise users employ Red Hat® Enterprise Linux® (RHEL). At the time of writing, the current release is RHEL 8.5, dated November 2021. This uses a kernel based on version 4.18. This kernel was released in August 2018.

As illustrated in the cartoon in Figure 2-1, it takes literally years to get new functionality from the idea stage into a production environment Linux kernel.[3]

---

[2] Yujuan Jiang et al., "Will My Patch Make It? And How Fast?," (paper, 2013). According to this research paper, 33% of patches are accepted, and most take 3–6 months.

[3] Thankfully, security patches to existing functionality get made available more quickly.

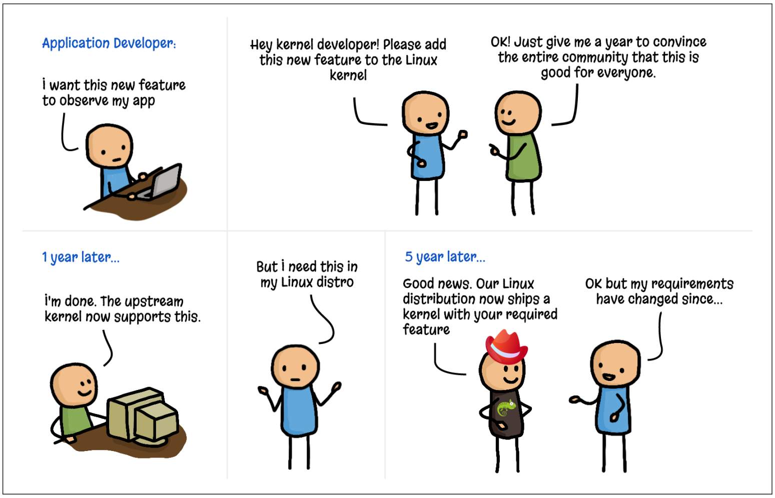

Application Developer:
i want this new feature to observe my app
Hey kernel developer! Please add this new feature to the Linux kernel
OK! Just give me a year to convince the entire community that this is good for everyone.
1 year later...
i'm done. The upstream kernel now supports this.
But i need this in my Linux distro
5 year later...
Good news. Our Linux distribution now ships a kernel with your required feature
OK but my requirements have changed since...

Figure 2-1. Adding features to the kernel (cartoon by Vadim Shchekoldin, Isovalent)

Kernel Modules

If you don't want to wait for years for your change to make it into the kernel, there is another option. The Linux kernel was designed to accept kernel modules, which can be loaded and unloaded on demand. If you want to change or extend kernel behavior, writing a module is certainly one way to do it. In our example of instrumenting the system call for opening files, you could write a kernel module to do this.

The biggest challenge here is that this is still full-on kernel programming. Users have historically been very cautious about using kernel modules for one simple reason: if kernel code crashes, it takes down the machine and everything running on it. How can a user be confident that a kernel module is safe to run?

Being "safe to run" doesn't just mean not crashing—the user wants to know that a kernel module is safe from a security perspective. Does it include vulnerabilities that an attacker could exploit? Do we trust the authors of the module not to put malicious code in it? Because the kernel is privileged code, it has access to everything on the machine, including all the data, so malicious code in the kernel would be a serious cause for concern. This applies to kernel modules too.

The safety of the kernel is one important reason why Linux distributions take so long to incorporate new releases. If other people have been running a kernel version in a variety of circumstances for months or years, this should have flushed out issues. The distribution maintainers can have some confidence that the kernel they ship to their users/customers is hardened—that is, safe to run.

eBPF offers a very different approach to safety: the *eBPF verifier*, which ensures that an eBPF program is only loaded if it's safe to run.

eBPF Verification and Security

Since eBPF allows us to run arbitrary code in the kernel, there needs to be a mechanism to make sure it's safe to run, won't crash users' machines, and won't compromise their data. This mechanism is the eBPF verifier.

The verifier analyzes an eBPF program to ensure that regardless of input, it will always terminate safely and within a bounded number of instructions. For example, if a program dereferences a pointer, the verifier requires that the program checks the pointer first to make sure that it is not null. Dereferencing a pointer means "looking up the value at this address" and the null or zero value is not a valid address to look at. If you dereference a null pointer in an application, that app crashes; whereas, dereferencing a null pointer in the kernel crashes the whole machine, so it's crucial to avoid it.

Verification also makes sure that eBPF programs can only access memory they are supposed to access. For example, imagine an eBPF program that's triggered in the networking stack, and passed the kernel's *socket buffer* that includes the data being transferred. There are special helper functions such as `bpf_skb_load_bytes()` that this eBPF program can call to read bytes of data from the socket buffer. Another eBPF program triggered by, say, a system call, where there's no socket buffer available, will not be permitted to use this helper function. The verifier also ensures that the program only reads bytes of data within that socket buffer—it's not allowed to access arbitrary memory. The intention here is to make sure that eBPF programs are safe from a security perspective.

Of course, it would still be possible to write a malicious eBPF program. If you can observe data for legitimate reasons, you can also observe it for illegitimate ones. Be careful to load only trusted eBPF programs from verifiable sources, and only grant permissions to manage eBPF tools to people that you would trust with root access.

Dynamic Loading of eBPF Programs

eBPF programs can be loaded into and removed from the kernel dynamically. Once they are attached to an event, they'll be triggered by that event regardless of what caused that event to occur. For example, if you attach a program to the syscall for opening files, it will be triggered whenever any process tries to open a file. It doesn't matter whether that process was already running when the program was loaded.

This leads to one of the great strengths of observability or security tooling that uses eBPF—it instantly gets visibility over everything that's happening on the machine.

Additionally, as illustrated in Figure 2-2, people can create new kernel functionality very quickly through eBPF without requiring every other Linux user to accept the same changes.

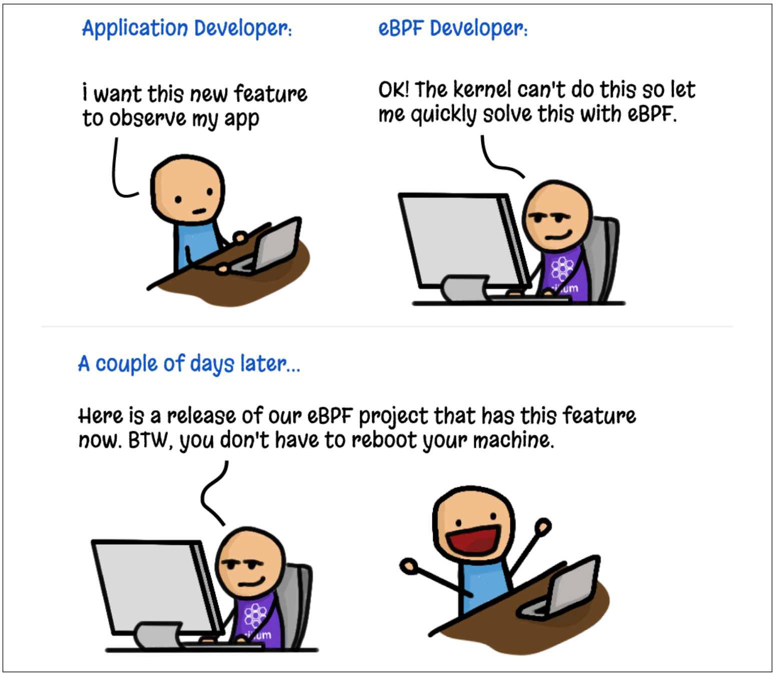

Application Developer:
i want this new feature to observe my app
eBPF Developer:
OK! The kernel can't do this so let me quickly solve this with eBPF.
A couple of days later...
Here is a release of our eBPF project that has this feature now. BTW, you don't have to reboot your machine.

Figure 2-2. Adding kernel features with eBPF (cartoon by Vadim Shchekoldin, Isovalent)

Now that you've seen how eBPF allows dynamic, custom changes to the kernel, let's examine what's involved if you want to write an eBPF program.

CHAPTER 3

eBPF Programs

In this chapter, let's turn to what's involved in writing eBPF code. We need to consider the eBPF program itself, that runs in the kernel, and also the user space code that will interact with it.

Kernel and User Space Code

First of all, what programming languages can you use to write eBPF programs?

The kernel accepts eBPF programs in bytecode form.[1] It's possible to write this bytecode by hand, in much the same way that it's possible to write application code in assembly language—but it's generally more practical for humans to use a higher-level language that can be compiled (that is, translated automatically) into bytecode.

eBPF programs can't be written in arbitrary high-level languages for a couple of reasons. First, the language compiler needs to have support for emitting the eBPF bytecode format that the kernel expects. Second, many compiled languages have runtime features—for example, Go's memory management and garbage collection—that make them unsuitable. At time of writing the only options for writing eBPF programs are C (compiled with `clang/llvm`) and, more recently, Rust. The vast majority of eBPF code published to date is in C, and this makes sense given that it's the language of the Linux kernel.

---

[1] See the BPF instruction set documentation.

At a minimum, something in user space needs to load the program into the kernel and attach it to the right event. There are utilities such as `bpftool` to help with this, but these are low-level tools that assume detailed knowledge of eBPF and are designed more for eBPF specialists than for the average user. In most eBPF-based tools, there is a user space application that takes care of loading the eBPF program into the kernel, passes in any configuration parameters, and displays information collected by the eBPF program in a user-friendly way.

The user space part of an eBPF tool can, at least in theory, be written in any language, though in practice there are libraries to support this in a fairly small set of languages: C, Go, Rust, and Python among them. This language choice is further complicated because not all languages have libraries that support `libbpf`, which has become a popular option for making eBPF programs portable across different versions of the kernel. (We'll discuss `libbpf` in Chapter 4.)

Custom Programs Attached to Events

The eBPF program itself is typically written in C or Rust and compiled into an object file.[2] This is a standard ELF (Executable and Linkable Format) file that can be inspected with tools like `readelf`, and it contains both the program bytecode and the definition of any maps (which we'll discuss shortly). As shown in Figure 3-1, user space program reads this file and loads it into the kernel, if allowed by the verifier that you met in the previous chapter.

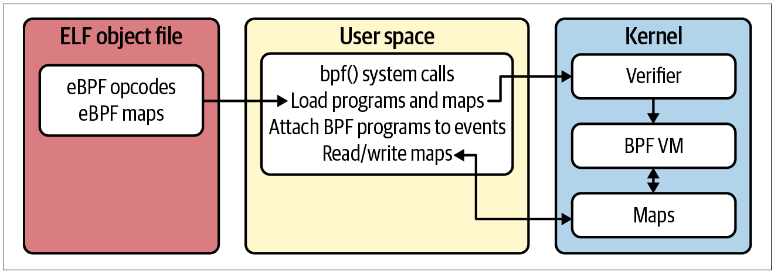


Figure 3-1. A user space application uses the `bpf()` system call to load eBPF programs from an ELF file into the kernel

---

[2] It's also possible to skip the object file and load bytecode directly into the kernel using the `bpf()` system call.

Once you have an eBPF program loaded into the kernel, it has to be attached to an event. Whenever the event happens, the associated eBPF program(s) are run. There's a very wide range of events that you can attach programs to; I won't cover them all, but the following are some of the more commonly used options.

Entry to/Exit from Functions

You can attach an eBPF program to be triggered whenever a kernel function is entered or exited. Many of today's eBPF examples use the mechanism of *kprobes* (attached to a kernel function entry point) and *kretprobes* (function exit). In more recent kernel versions, there is a more efficient alternative called *fentry/fexit*.[3]

Note that you can't guarantee that all functions defined in one kernel version will necessarily be available in future versions unless they are part of a stable API such as the syscall interface.

You can also attach eBPF programs to user space functions with *uprobes* and *uretprobes*.

Tracepoints

You can also attach eBPF programs to *tracepoints*[4] defined within the kernel. Find the events on your machine by looking under `/sys/kernel/debug/tracing/events`.

Perf Events

Perf[5] is a subsystem for collecting performance data. You can hook eBPF programs to all the places where perf data is collected, which can be determined by running perf list on your machine.

Linux Security Module Interface

The LSM interface allows for security policies to be checked before the kernel allows certain operations. You may have come across

---

[3] fentry/fexit is described in an article by Alexei Starovoitov: "Introduce BPF Trampoline" (LWN.net, November 14, 2019).

[4] *Oracle Linux Blog*, "Taming Tracepoints in the Linux Kernel," by Matt Keenan, posted March 9, 2020.

[5] Brendan Gregg's site is a good source of information about perf events.

AppArmor or SELinux that make use of this interface. With eBPF, you can attach custom programs to the same checkpoints, allowing for flexible, dynamic security policies and some new approaches to runtime security tooling.

Network Interfaces—eXpress Data Path

eXpress Data Path (XDP) allows attaching an eBPF program to a network interface, so that it is triggered whenever a packet is received. It can inspect or even modify the packet, and the program's exit code can tell the kernel what to do with that packet: pass it on, drop it, or redirect it. This can form the basis of some very efficient networking functionality.[6]

Sockets and Other Networking Hooks

You can attach eBPF programs to run when applications open or perform other operations on a network socket, as well as when messages are sent or received. There are also hooks called *traffic control* or *tc* within the kernel's network stack where eBPF programs can run after initial packet processing.

Some features can be implemented with an eBPF program alone, but in many cases we want the eBPF code to receive information from, or pass data to, a user space application. The mechanism that allows data to pass between eBPF programs and user space, or between different eBPF programs, is called *maps*.

eBPF Maps

The development of maps is one of the significant differences that justify the *e* for *extended*, in the eBPF acronym.

Maps are data structures that are defined alongside eBPF programs. There are a variety of different types of maps, but they are all essentially key–value stores. eBPF programs can read and write to them, as can user space code. Common uses for maps include:

---

[6] If you're interested in seeing a concrete example of this, you might like to watch my talk at eBPF Summit 2021 where I implement a very basic load balancer in a few minutes, as an illustration of how we can use eBPF to change the way the kernel handles network packets.

- An eBPF program writing metrics and other data about an event, for user space code to later retrieve
- User space code writing configuration information, for an eBPF program to read and behave accordingly
- An eBPF program writing data into a map, for later retrieval by another eBPF program, allowing the coordination of information across multiple kernel events

If both the kernel and user space code will access the same map, they will need a common understanding of the data structures stored in that map. This can be done by including header files that define those data structures in both the user space and kernel code, but if these aren't written in the same language, the author(s) will need to carefully create structure definitions that are byte-for-byte compatible.

We've discussed the main constituents of an eBPF tool: eBPF programs that run in the kernel, user space code to load and interact with those programs, and maps that allow programs to share data. To make things concrete, let's look at an example.

Opensnoop Example

For this example of an eBPF program, I've chosen `opensnoop`, a utility that shows you what files any process opens. The original version of this utility was one of many BPF tools that Brendan Gregg originally wrote in the BCC project which you can find on GitHub. It was later rewritten for `libbpf` (which you'll meet in the next chapter), and in this example I'm using the newer version under the `libbpf-tools` directory.

When you run `opensnoop`, the output you'll see depends a lot on what's happening on the virtual machine at the time, but it should look something like this:

```
PID    COMM          FD ERR PATH
93965  cat            3   0 /etc/ld.so.cache
93965  cat            3   0 /lib/x86_64-linux-gnu/libc.so.6
93965  cat            3   0 /usr/lib/locale/locale-archive
93965  cat            3   0 /usr/share/locale/locale.alias
...
```

Each line of output indicates that a process opened (or attempted to open) a file. The columns show the process ID, the command being run, the file descriptor, an indication of any error code, and the path of the file being opened.

Opensnoop works by attaching eBPF programs to the `open()` and `openat()` system calls that any application has to make to ask the kernel to open a file. Let's dig in to see how this is implemented. For brevity, we won't look at every line of the code, but I hope it's sufficient to give you an idea of how it works. (Feel free to skip to the next chapter if you're not interested in diving this deep!)

Opensnoop eBPF Code

The eBPF code is written in C, in the file `opensnoop.bpf.c`. Near the beginning of this file you can see the definitions of two eBPF maps—`start` and `events`:

```
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 10240);
    __type(key, u32);
    __type(value, struct args_t);
} start SEC(".maps");
struct {
    __uint(type, BPF_MAP_TYPE_PERF_EVENT_ARRAY);
    __uint(key_size, sizeof(u32));
    __uint(value_size, sizeof(u32));
} events SEC(".maps");
```

When the ELF object file is created, it contains a section for each map and each program to be loaded into the kernel, and the `SEC()` macro defines these sections.

As you'll see when we look into the program, the `start` map is used to temporarily store the arguments to the syscall—including the name of the file being opened—while the syscall is being processed. The `events` map[7] is used for passing event information from the eBPF code in the kernel to the user space executable. This is illustrated in Figure 3-2.

---

[7] At the time of writing, this code uses a perf buffer for the events map. If you were writing this code today for recent kernels, you would get better performance from a ring buffer, which is a newer alternative.

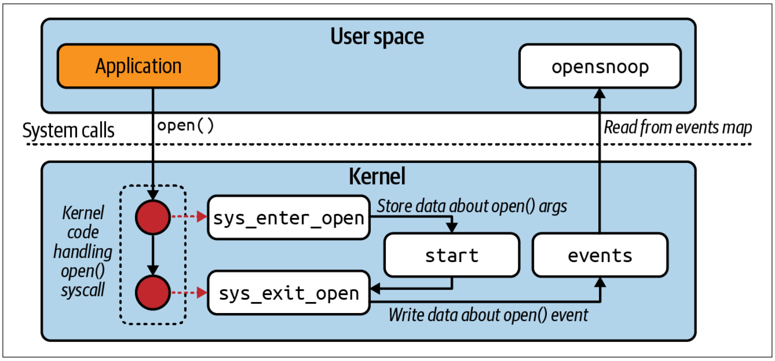

User space
Application
opensnoop
System calls
open()
Read from events map
Kernel
Kernel code handling open() syscall
sys_enter_open
Store data about open() args
start
events
sys_exit_open
Write data about open() event

Figure 3-2. Calling `open()` triggers eBPF programs that store data in opensnoop's eBPF maps

Later in the `opensnoop.bpf.c` file, you'll find two extremely similar functions:

```
SEC("tracepoint/syscalls/sys_enter_open")
int tracepoint__syscalls__sys_enter_open(struct
    trace_event_raw_sys_enter* ctx)
```

and

```
SEC("tracepoint/syscalls/sys_enter_openat")
int tracepoint__syscalls__sys_enter_openat(struct
    trace_event_raw_sys_enter* ctx)
```

There are two different system calls for opening files:[8] `openat()` and `open()`. They are identical except that `openat()` has an extra argument for a directory file descriptor, and the path name for the file to be opened is taken relative to that directory. Likewise, the two functions in `opensnoop` are identical except for handling this difference in the arguments.

As you can see, they both take a parameter that is a pointer to a structure called `trace_event_raw_sys_enter`. You'd find the definition for this structure in the `vmlinux` header file generated for the particular kernel you're running on. The art of writing eBPF programs includes working out what structure each program receives as its context, and how to access the information within it.

---

[8] In some kernels you'll also find `openat2()`, but this isn't handled in this version of `opensnoop`, at least at time of writing.

These two functions use a BPF helper function to retrieve the ID of the process that's calling this syscall:

```
u64 id = bpf_get_current_pid_tgid();
```

The code gets the filename and any flags that were passed to the syscall, and puts them in a structure called `args`:

```
args.fname = (const char *)ctx->args[0];
         args.flags = (int)ctx->args[1];
```

This structure is written into the `start` map using the current process ID as the key:

```
bpf_map_update_elem(&start, &pid, &args, 0);
```

And that's all that the eBPF programs do on entry to the syscall. But there's another pair of eBPF programs defined in `opensnoop.bpf.c` that get triggered when the syscalls exit:

```
SEC("tracepoint/syscalls/sys_exit_open")
int tracepoint__syscalls__sys_exit_open
```

This program and its `openat()` twin share common code in the function `trace_exit()`. Have you noticed that all the functions called by eBPF programs are prefixed by `static __always_inline`? That forces the compiler to put the instructions for these functions inline, because in older kernels a BPF program is not allowed to jump to a separate function. Newer kernels and versions of LLVM can support noninlined function calls, but this is a safe way to ensure the BPF verifier stays happy. (Nowadays there is also the concept of a BPF tail call, where execution jumps from one BPF program to another. You can read more about BPF function calls and tail calls in the eBPF documentation.)

The `trace_exit()` function creates an empty event structure:

```
struct event event = {};
```

This will get populated with information about the `open/openat` syscall that's coming to a conclusion and sent to user space via the `events` map.

There should be an entry in the start hash map that corresponds to the current process ID:

```
ap = bpf_map_lookup_elem(&start, &pid);
```

This has the information about the filename and flags that was written earlier during the `sys_enter_open(at)` call. The flags field is an integer stored directly in the structure, so it's OK to read it directly from the structure:

```
event.flags = ap->flags;
```

In contrast, the filename is written into some number of bytes in user space memory, and the verifier needs to be sure that it's safe for this eBPF program to read that number of bytes from that location in memory. This is done using another helper function, `bpf_probe_read_user_str()`:

```
bpf_probe_read_user_str(&event.fname, sizeof(event.fname),
                        ap->fname);
```

The current command name (that is, the name of the executable that made the `open(at)` syscall) is also copied into the event structure, using another BPF helper function:

```
bpf_get_current_comm(&event.comm, sizeof(event.comm));
```

The event structure gets written into the `events` perf buffer map:

```
bpf_perf_event_output(ctx, &events, BPF_F_CURRENT_CPU,
                      &event, sizeof(event));
```

The user space code reads event information out of this map. Before we get to that, let's look briefly at the Makefile.

libbpf-tools Makefile

When you build eBPF code, you get an object file containing the binary definitions of the eBPF programs and maps. You also need an additional user space executable that will load those programs and maps into the kernel, and act as the interface for the user.[9] Let's look at the Makefile that builds `opensnoop` to see how it creates both the eBPF object file and the executable.

Makefiles comprise a set of rules, and the syntax for these can be a bit opaque, so if you're not familiar with Makefiles and don't particularly care about the details, please do feel free to skip over this section!

---

[9] You could use a general-purpose tool like `bpftool`, which can read BPF object files and perform operations on them, but that requires the user to know details about what to load and what events to attach programs to. For most applications, it makes sense to write a specific tool that simplifies this for the end user.

The `opensnoop` example that we're looking at is one of a large set of example tools that are all built using one Makefile that you'll find in the `libbpf-tools` directory. Not everything in this file is particularly of interest, but there are a few rules I'd like to highlight. The first is a rule that takes a `bpf.c` file and uses the `clang` compiler to create a BPF target object file:

```
$(OUTPUT)/%.bpf.o: %.bpf.c $(LIBBPF_OBJ) $(wildcard %.h) $(AR..
    $(call msg,BPF,$@)
    $(Q)$(CLANG) $(CFLAGS) -target bpf -D__TARGET_ARCH_$(ARCH) \
         -I$(ARCH)/ $(INCLUDES) -c $(filter %.c,$^) -o $@ && \
    $(LLVM_STRIP) -g $@
```

So, `opensnoop.bpf.c` gets compiled into `$(OUTPUT)/opensnoop.bpf.o`. This object file contains the eBPF programs and maps that will get loaded into the kernel.

Another rule uses `bpftool gen skeleton` to create a skeleton header file from the map and program definitions contained in that `bpf.o` object file:

```
$(OUTPUT)/%.skel.h: $(OUTPUT)/%.bpf.o | $(OUTPUT)
   $(call msg,GEN-SKEL,$@)
   $(Q)$(BPFTOOL) gen skeleton $< > $@
```

The `opensnoop.c` user space code includes this `opensnoop.skel.h` header file to get the definitions of the maps that it shares with the eBPF programs in the kernel. This allows the user space and kernel code to know about the layout of the data structures that get stored in these maps.

The following rule compiles the user space code from `opensnoop.c` into a binary object called `$(OUTPUT)/opensnoop.o`:

```
$(OUTPUT)/%.o: %.c $(wildcard %.h) $(LIBBPF_OBJ) | $(OUTPUT)
   $(call msg,CC,$@)
   $(Q)$(CC) $(CFLAGS) $(INCLUDES) -c $(filter %.c,$^) -o $@
```

Finally, there is a rule that uses `cc` to link the user space application objects (in our case, `opensnoop.o`) into a set of executables:

```
$(APPS): %: $(OUTPUT)/%.o $(LIBBPF_OBJ) $(COMMON_OBJ) | $(OUT...
   $(call msg,BINARY,$@)
   $(Q)$(CC) $(CFLAGS) $^ $(LDFLAGS) -lelf -lz -o $@
```

Now that you have seen how the eBPF and user space programs are generated separately, let's look at the user space code.

Opensnoop User Space Code

As I mentioned, the user space code that interacts with eBPF code could be written in pretty much any programming language. The example that we'll discuss in this section is written in C, but if you're interested, you could compare it with the original BCC version written in Python, that you'll find in bcc/tools.

The user space code is in the `opensnoop.c` file. The first half of the file has `#include` directives (one of them being the autogenerated `opensnoop.skel.h` file), various definitions, and the code to handle different command line options, which we won't dwell on here. Let's also gloss over functions like `print_event()` which writes the information about an event to the screen. From an eBPF perspective, all the interesting code is in the `main()` function.

You will see functions like `opensnoop_bpf__open()`, `opensnoop_bpf__load()`, and `opensnoop_bpf__attach()`. These are all defined in the autogenerated code created by `bpftool gen skeleton`.[10] This autogenerated code handles all the individual eBPF programs, maps, and attachment points defined in the eBPF object file.

Once `opensnoop` is up and running, its job is to listen on the `events` perf buffer and write the information contained in each event to the screen. First, it opens the file descriptor associated with the perf buffer and sets `handle_event()` as the function to be called when a new event arrives:

```
pb = perf_buffer__new(bpf_map__fd(obj->maps.events),
    PERF_BUFFER_PAGES, handle_event, handle_lost_events,
    NULL, NULL);
```

Then it polls on buffer events until either a time limit is reached, or the user interrupts the program:

```
while (!exiting) {
        err = perf_buffer__poll(pb, PERF_POLL_TIMEOUT_MS);
...
}
```

The data parameter passed to `handle_event()` points to the event structure that the eBPF program wrote into the map for this event.

---

[10] See Andrii Nakryiko's post describing BPF skeleton code generation.

The user space code can retrieve this information, format it and write it out for the user to see.

As you've seen, `opensnoop` registers eBPF programs that are called every time any application calls the `open()` or `openat()` system call. These eBPF programs running in the kernel collect information about the context of that system call—the executable name and process ID—and about the file being opened. This information is written into a map, from which user space can read it and display it to the user.

You'll find dozens more examples of eBPF tools like this in the `libbpf-tools` directory, each of which typically instruments one syscall, or a family of related syscalls like `open()` and `openat()`.

System calls are a stable kernel interface, and they offer a very powerful way to observe what's happening on a (virtual) machine. But don't be fooled into thinking that eBPF programming begins and ends at intercepting system calls. There are plenty of other stable interfaces, including LSM and various points in the networking stack, to which eBPF can be attached. If you're willing to risk or work around changes between kernel versions, the range of places where you can attach eBPF programs is absolutely vast.

CHAPTER 4

eBPF Complexity

You've now seen an example of eBPF programming to give you a flavor of how it works. While basic examples can make eBPF seem relatively straightforward, there are some complexities that make it challenging.

One area that has historically made it relatively difficult to write and distribute eBPF programs is kernel compatibility.

Portability Across Kernels

eBPF programs can access kernel data structures, and these may change across different kernel versions. The structures themselves are defined in header files that form part of the Linux source code. Back in the day, you had to compile your eBPF programs against the correct set of header files compatible with the kernel where you want to run those programs.

BCC Approach to Portability

To address portability across kernels, the BCC[1] (BPF Compiler Collection) project took the approach of compiling eBPF code at runtime, in situ on the destination machine. This means the compilation toolchain needs to be installed onto every destination machine where you want the code to run,[2] and you have to wait for the compilation to complete before the tool starts. You also have to hope that the kernel headers are present on the filesystem (and that's not always the case). Enter BPF CO-RE.

---

[1] You'll find BCC at this GitHub page.

CO-RE

The CO-RE—compile once, run everywhere—approach consists of a few elements:

BTF (BPF Type Format)
: This is a format for expressing the layout of data structures and function signatures. Modern Linux kernels support BTF, so that you can generate a header file called `vmlinux.h` from a running system, containing all the data structure information about a kernel that a BPF program might need.

`libbpf`, *the BPF library*
: On the one hand, `libbpf` provides functions for loading eBPF programs and maps into the kernel. But it also plays an important role in portability: it leans on BTF information to adjust the eBPF code to compensate for any differences between the data structures present when it was compiled, and what's on the destination machine.

Compiler support
: The `clang` compiler was enhanced so that when it compiles eBPF programs, it includes what are known as *BTF relocations*, which are what `libbpf` uses to know what to adjust as it loads BPF programs and maps into the kernel.

Optionally, a BPF skeleton
: A skeleton can be autogenerated from a compiled BPF object file using `bpftool gen skeleton`, containing handy functions that user space code can call to manage the lifecycle of BPF programs—loading them into the kernel, attaching them to events and so on. These functions are higher-level abstractions that can be more convenient for the developer than using `libbpf` directly.

---

[2] Some projects take the approach of packaging the eBPF source plus the required toolchain into a container image. This avoids the complexity of installing that toolchain and any concomitant dependency management, but it still means that the compilation step runs on the destination machine.

For a more detailed explanation of CO-RE, read Andrii Nakryiko's excellent description.

BTF information in the form of a `vmlinux` file has been included in the Linux kernel since version 5.4,[3] but raw BTF data that `libbpf` can make use of can also be generated for older kernels. There's information on how to generate BTF files, and an archive of files for a variety of Linux distributions, on the BTF Hub.

The BPF CO-RE approach makes it far easier than it used to be for an eBPF programmer to get their code to run on any Linux distribution—or at least, on any Linux distribution new enough to have support for whatever set of eBPF capabilities their program uses. But this doesn't make eBPF programming a walk in the park: it's still essentially kernel programming.

Linux Kernel Knowledge

It quite quickly becomes clear that you need some domain knowledge about the Linux kernel in order to write more advanced tools. You'll need to understand the data structures you have access to, which depend on the context in which your eBPF code is called. Not every application developer has experience in parsing network packets, accessing socket buffers, or handling the arguments to a system call.

How will the kernel react to your eBPF code's behavior? As you learned in Chapter 2, the kernel consists of millions of lines of code. Its documentation can be sparse, so you might find yourself having to read kernel source code to figure out how something works.

You'll also need to figure out what events your eBPF code should attach to. With the option to attach a kprobe to any function entry point in the entire kernel, it might not be an easy decision. In some cases, it's straightforward—for example, if you want to access an incoming network packet, then the XDP hook on the appropriate network interface is an obvious choice. If you want to provide observability into a particular kernel event, it may not be terribly hard to find the appropriate point within the kernel code.

---

[3] See Andrii Nakryiko's IO Visor post for more information.

But in other cases, the choice may be less obvious. As an example, tools that simply use kprobes to hook into the functions that make up the kernel's syscall interface may be subject to a security exploit known as a *time-of-check to time-of-use* (TOCTTOU). An attacker has a small window of opportunity where they can change a syscall's arguments after the eBPF code has read them, but before they have been copied into kernel memory. There was an excellent presentation on this at DEF CON 29[4] by Rex Guo and Junyuan Zeng. Some of the most widely used eBPF tooling was written in quite a naive way and is subject to this kind of attack. It's not an easy exploit, and there are ways to mitigate these attacks, but if you're protecting highly sensitive data against sophisticated, motivated adversaries, please dig in to understand whether the tools you use might be affected.

You've already seen how BPF CO-RE enables eBPF programs to work on different kernel versions, but it only takes into account the changes in data structure layout and not broader changes to kernel behavior. For example, if you want to attach an eBPF program to a particular function or tracepoint in the kernel, you may need a Plan B for what to do if that function or tracepoint doesn't exist in a different kernel version.

Coordinating Multiple eBPF Programs

A lot of eBPF-based tools available today offer a suite of observability capabilities, enabled by hooking eBPF programs into a set of kernel events. Much of this was pioneered by the work that Brendan Gregg and others did in BCC and `bpftrace` tools. Today's generation of (often commercial) tools may offer much prettier graphics and UIs, but the eBPF programs they leverage are based highly on those originals.

Things get considerably more complicated when you want to write code that coordinates interactions between different types of events. As an example, Cilium sees network packets at a variety of points through the kernel's networking stack,[5] and manipulates traffic based on information from the Kubernetes CNI (container network interface) about Kubernetes pods. Building this system requires Cilium developers to have an in-depth understanding of how the kernel handles network traffic, and how the user space concepts of "pods" and "containers" map to kernel concepts like cgroups and namespaces. In practice, several Cilium maintainers are also kernel developers working on enhancements to eBPF and to networking support; hence, they have this knowledge.

---

[4] Rex Guo and Junyuan Zeng, "Phantom Attack: Evading System Call Monitoring," (DEF CON, August 5–8, 2021).

[5] The Cilium documentation describes how eBPF programs attached to different networking hooks are combined to achieve complex networking capabilities.

The bottom line is that although eBPF offers an extremely efficient and powerful platform for hooking into the kernel, it's nontrivial for the average developer without significant kernel experience. If you're interested in getting your hands dirty with eBPF programming, I highly encourage it as a learning exercise; building up experience in this area could be highly valuable since it's bound to continue to be a sought-after specialist skill for years to come. But realistically, most organizations are unlikely to build much bespoke eBPF tooling in-house, but instead will leverage projects and products from the specialist eBPF community.

Let's move on to considering why these eBPF-based projects and products are particularly powerful in a cloud native environment.

CHAPTER 5

eBPF in Cloud Native Environments

The cloud native approach to computing has taken off exponentially in recent years. In this chapter, I'll discuss why eBPF is so well-suited to tooling for cloud native environments. To keep things concrete, I'll refer to Kubernetes, but the same concepts apply to any platform that uses containers.

One Kernel per Host

To understand why eBPF is so powerful in the cloud native world, you'll need to be very clear on one concept: there is only one kernel per machine (or virtual machine), and all the containers running on that machine share the same kernel,[1] as shown in Figure 5-1. The same kernel is involved with and aware of all the application code running on any given host machine.

By instrumenting the kernel, as we do when using eBPF, we can simultaneously instrument all the application code running on that machine. When we load an eBPF program into the kernel and attach it to an event, it gets triggered irrespective of which process is involved with the event.

---

[1] This is nearly always true, unless you are using a virtualization approach like Kata containers, Firecracker or unikernels, where each "container" runs in its own virtual machine.

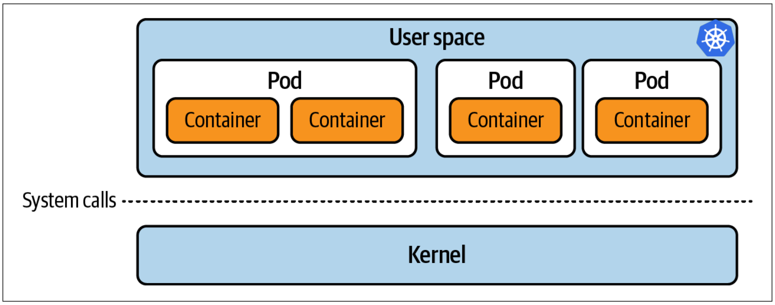


Figure 5-1. All the containers on the same host share a single kernel

eBPF Versus the Sidecar Model

Prior to eBPF, most observability and security tooling for Kubernetes used the *sidecar* model. This model allows you to deploy the instrumentation in a separate container but within the same pod as the application. When this approach was invented, it was a step forward because it meant you no longer had to write instrumentation code directly in the app. Simply by deploying the sidecar, it would have visibility over the other containers in the same pod. The process of injecting sidecars is usually automated, so this provides a mechanism for ensuring all your apps are instrumented.

Each sidecar container consumes resources, and this is multiplied by the number of pods with the sidecar injected. This can be very significant—for example, if each sidecar needs its own copy of routing information, or policy rules, this is wasteful. (For more on this, Thomas Graf wrote a comparison of sidecars with eBPF for service mesh.)

Another issue with sidecars is that you can't guarantee that every application on the machine has been instrumented correctly. Imagine that an attacker manages to compromise one of your hosts and starts a separate pod to run, say, a cryptocurrency miner. They are unlikely to do you the courtesy of instrumenting their mining pod with your sidecar observability or security tools. You'll need a separate system to be aware of this activity.

But that same cryptocurrency miner shares the kernel with the legitimate pods running on that host. If you're using eBPF-based instrumentation, as illustrated in Figure 5-2, the miner is automatically subject to it.

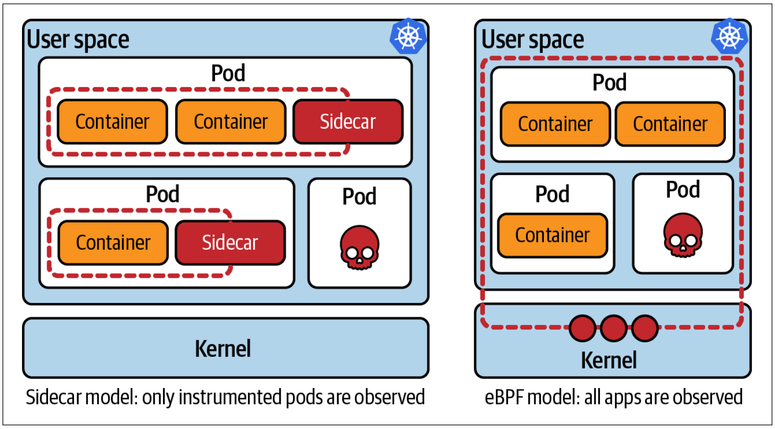

User space
Pod
Container
Container
Sidecar
Pod
Container
Sidecar
Pod
Kernel
Sidecar model: only instrumented pods are observed
User space
Pod
Container
Container
Pod
Container
Pod
Kernel
eBPF model: all apps are observed

Figure 5-2. Sidecars can only observe activity in their own pods, but eBPF programs can observe all activity

eBPF and Process Isolation

Instead of per-pod sidecars, I'm advocating the consolidation of functionality into a single per-node, eBPF-based agent. If that agent has access to all the pods running on the machine, isn't that a security risk? Haven't we lost the isolation between applications that might prevent them from interfering with each other?

As someone who has spent a lot of time working in container security, I can relate to these concerns, but it's important to dig into the underlying mechanisms to really understand why it's not the flaw that it might appear to be at first.

The important thing to remember is that those pods all share one kernel, and the kernel does not have an innate understanding of pods or containers. Instead, the kernel operates on processes, and uses cgroups and namespaces to isolate processes from each other. Those structures stop processes in user space from being able to see or interfere with each other, as policed by the kernel. As soon as data is being processed within the kernel (for example, being read from disk or sent to a network) you are relying on the kernel behaving correctly. It is only the kernel's own code that says, for example, that it should respect, say, file permissions. There's no extra magic authoritative layer that would stop the kernel from ignoring file permissions and reading data out of any file it wanted to access—it's simply that the kernel itself is coded not to do so.

The security controls that exist on a Linux system assume that the kernel itself can be trusted. They are there to protect against bad behavior from code running in user space.

We saw in Chapter 2 that the eBPF verifier ensures that any eBPF program only attempts to access memory that it should have access to. The verifier checks that the program can't possibly go beyond its remit, including ensuring that the memory is owned by the current process or is part of the current network packet. This means that eBPF code is subject to much stricter controls than the surrounding kernel code, which doesn't have to pass any kind of verifier step.

If an attacker escapes a containerized application onto the node, and is able to escalate privileges, that attacker can compromise other applications on the same node. Since those escapes are not unknown, as a container security expert, I would not recommend running sensitive applications on a shared machine alongside untrusted applications or users without some level of additional security tooling. For highly sensitive data, you might not even want to run within a virtual machine on the same bare metal as untrusted users. But if you are prepared to run applications side-by-side on the same virtual machine (which is completely reasonable in many applications that are not particularly sensitive), then eBPF is not adding risk beyond what already exists by sharing a kernel.

Of course, a malicious eBPF program could wreak all kinds of havoc, and it would certainly be easy to write eBPF code that behaves badly—for example, taking copies of every network packet and sending it to an eavesdropper. By default, nonroot users don't have permission to load eBPF programs,[2] and you should only grant users or software systems this permission if you really trust them, much as for root permissions. So, you do have to be careful about the provenance of the code you run (and there is an initiative in play to support signature checking of eBPF programs to help with this). You can also use eBPF programs to keep a watchful eye on other eBPF programs!

Now that you have an overview of why eBPF is a powerful basis for cloud native instrumentation, the next chapter gives you some concrete examples of eBPF tooling from the cloud native ecosystem.

---

[2] The Linux capability CAP_BPF grants permission to load BPF programs.

CHAPTER 6

eBPF Tools

Now that you've learned about what eBPF is, and something of how eBPF programs work, let's turn to exploring some of the tools, built on this technology, that you might make use of in a production deployment today. We'll consider some examples of eBPF-based open source projects that provide capabilities in three important areas: networking, observability, and security.

Networking

eBPF programs can be attached to network interfaces and to various points in the kernel's network stack. At each point, they can drop packets, send them to different destinations, or even modify the contents. This enables some very powerful capabilities. Let's look at a few networking features that are now commonly implemented with eBPF.

Load Balancing

If you have any doubts about the scalability of eBPF for networking, know that it is being used at massive scale at Facebook. They were an early adopter of BPF and introduced Katran in 2018, an open source, layer 4 load balancer.

Another example of a highly scaled load balancer comes from Cloudflare's Unimog edge load balancer. By running within the kernel, eBPF programs can manipulate network packets and forward them to an appropriate destination, without each packet having to pass through the networking stack and on to user space.

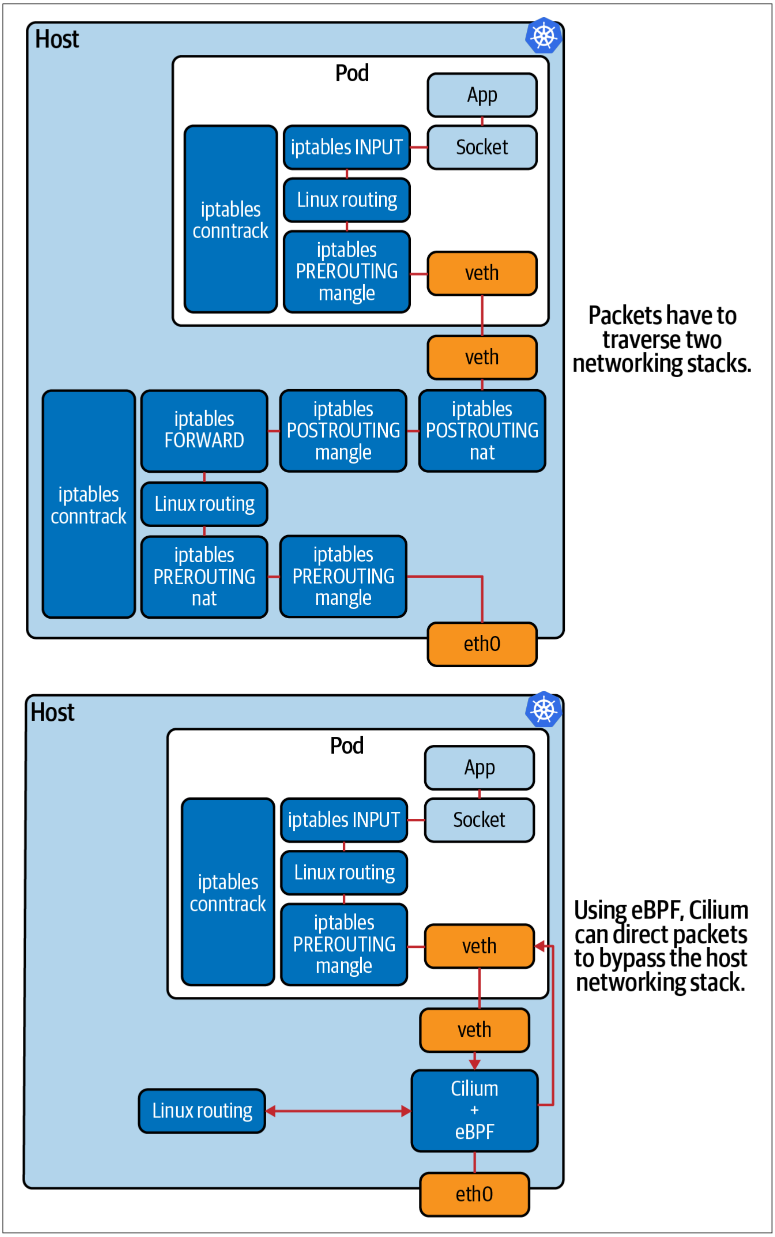
The Cilium project is better known as an eBPF Kubernetes networking plug-in (as I'll discuss in a moment) but it's also in use in large telecommunication and on-premises deployments as a standalone load balancer. Again, the ability to process packets at an early stage without them having to transition into user space makes this highly performant.

Kubernetes Networking

CNCF project Cilium was the original eBPF-based CNI implementation. It was originally started by a group of kernel maintainers working on eBPF who recognized the potential for its use in cloud native networking. It's now used as the default data plane for Google Kubernetes Engine, Amazon EKS Anywhere, and Alibaba Cloud.

In a cloud native world, pods stop and start all the time, and each pod gets assigned an IP address. Prior to eBPF-enabled networking, each node had to keep updating a set of iptables rules for each of these changes in order to route between pods; and managing these iptables rules gets unwieldy at scale. As illustrated in Figure 6-1, Cilium dramatically simplifies routing so that it's essentially a simple lookup table in eBPF, leading to measurable performance improvements.

Another Kubernetes CNI that added an eBPF implementation, alongside their traditional iptables version, is Project Calico.


Host
Pod
App
Socket
iptables INPUT
Linux routing
iptables PREROUTING mangle
iptables conntrack
veth
veth
iptables FORWARD
iptables POSTROUTING mangle
iptables POSTROUTING nat
iptables conntrack
Linux routing
iptables PREROUTING nat
iptables PREROUTING mangle
eth0
Host
Pod
App
Socket
iptables INPUT
Linux routing
iptables PREROUTING mangle
iptables conntrack
veth
veth
Linux routing
Cilium + eBPF
eth0

Packets have to traverse two networking stacks.

Using eBPF, Cilium can direct packets to bypass the host networking stack.

Figure 6-1. Bypassing the host networking stack with eBPF

Service Mesh

eBPF also makes real sense as the basis for a more efficient data plane for service mesh. Many service mesh features operate at layer 7, the application layer, and use a proxy component such as Envoy to act on behalf of an application. In Kubernetes, these proxies are often deployed in a sidecar model, with one proxy container per pod, so that the proxy has access to the pod's network namespace. As you saw in Chapter 5, eBPF allows a more efficient approach than the sidecar model. Since the kernel has access to all the pod namespaces, we can use eBPF to make connections between applications in pods and a single proxy on the host, as shown in Figure 6-2.

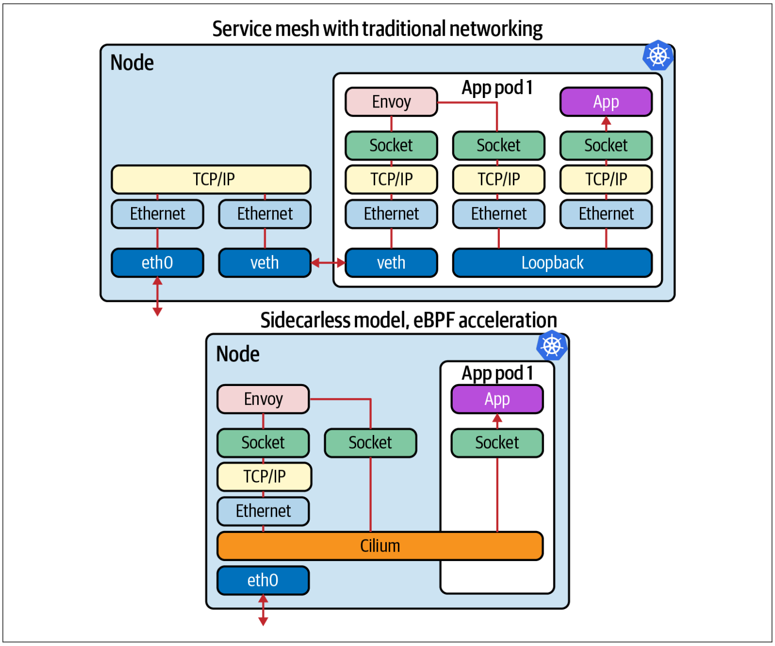

Service mesh with traditional networking
Node
App pod 1
Envoy
App
Socket
Socket
Socket
TCP/IP
TCP/IP
TCP/IP
TCP/IP
Ethernet
Ethernet
Ethernet
Ethernet
Ethernet
eth0
veth
veth
Loopback
Sidecarless model, eBPF acceleration
Node
App pod 1
Envoy
App
Socket
Socket
Socket
TCP/IP
Ethernet
Cilium
eth0

Figure 6-2. eBPF enables an efficient sidecarless model for service mesh, with one proxy per node rather than one per application pod

I have another article about using eBPF for more efficient service mesh data planes, as have Solo.io. At the time of writing, the Cilium Service Mesh is available in beta and showing early performance gains over the traditional sidecar proxy approach.

Observability

As you've seen earlier in this report, eBPF programs can get visibility into everything that's happening on a machine. By collecting data about events and passing them to user space, eBPF enables a range of powerful observability tools that can show you how your applications are performing and behaving, without having to make any changes to instrument those apps. eBPF also enables observability over the entire system, not just individual applications, so you can understand the behavior of your host machines.

You've come across the BCC project earlier in this report, and over several years Brendan Gregg has done pioneering work at Netflix to show how these eBPF tools can be used to observe practically any metrics you're interested in, at scale and with high performance.

Kinvolk's Inspektor Gadget takes some of these tools with their origins in BCC into the world of Kubernetes, so that you can easily observe specific workloads on the command line.

A new generation of projects and tools is building on this work to provide GUI-based observability. CNCF project Pixie lets you run prewritten or custom scripts and see metrics and logs through a powerful and visually appealing UI. Because it's based on eBPF, this means you can automatically instrument all your applications and get performance data without making any code changes or configuration. Figure 6-3 shows just one example of the many visualizations available in Pixie.

Another observability project called Parca focuses on continuous profiling, using eBPF to efficiently sample metrics like CPU usage that you can use to detect performance bottlenecks.

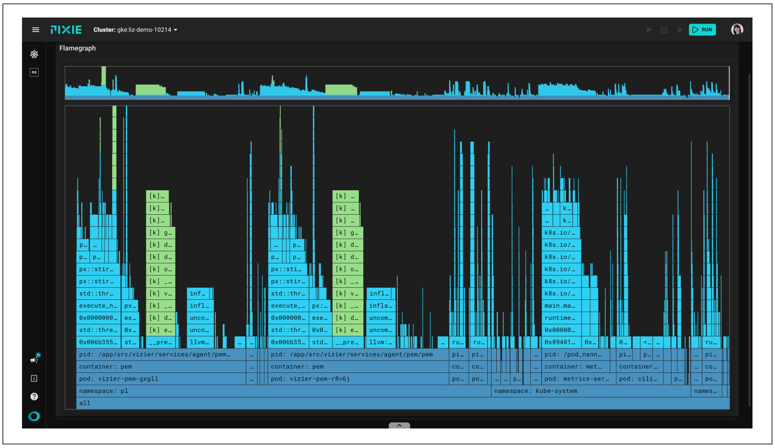
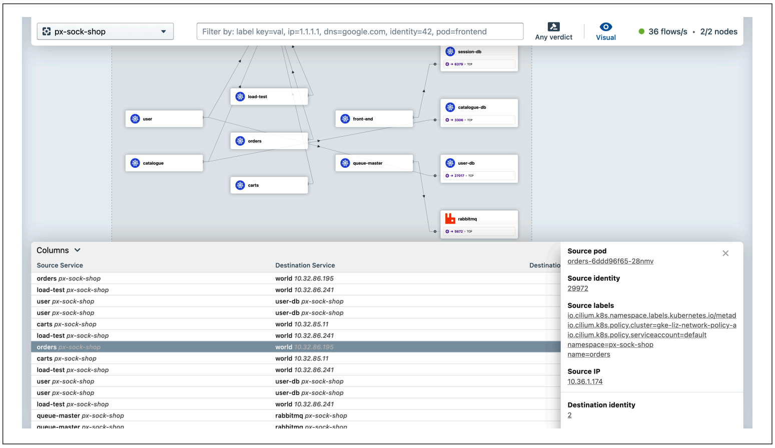

PIXIE
Cluster: gke:liz-demo-10214
RUN
Flamegraph
pid: /app/src/vizier/services/agent/pem…
container: pem
pod: vizier-pem-gxgll
namespace: pl
pid: /app/src/vizier/services/agent/pem/pem
container: pem
pod: vizier-pem-r8v6j
namespace: kube-system
pod: metrics-ser…
pod: cili…
all

Figure 6-3. A Pixie flamegraph of everything running on a small Kubernetes cluster

The Hubble component of Cilium is an observability tool with both a command line interface and UI (shown in Figure 6-4) that focuses on network flows in your Kubernetes clusters.


px-sock-shop
Filter by: label key=val, ip=1.1.1.1, dns=google.com, identity=42, pod=frontend
Any verdict
Visual
36 flows/s • 2/2 nodes
session-db
6379 • TCP
load-test
user
front-end
catalogue-db
3306 • TCP
orders
catalogue
queue-master
user-db
27017 • TCP
carts
rabbitmq
5672 • TCP
Columns

Source Service	Destination Service	Destinatio
orders *px-sock-shop*	world *10.32.86.195*	
load-test *px-sock-shop*	world *10.32.86.241*	
user *px-sock-shop*	user-db *px-sock-shop*	
user *px-sock-shop*	user-db *px-sock-shop*	
carts *px-sock-shop*	world *10.32.85.11*	
load-test *px-sock-shop*	world *10.32.86.241*	
orders *px-sock-shop*	world *10.32.86.195*	
carts *px-sock-shop*	world *10.32.85.11*	
load-test *px-sock-shop*	world *10.32.86.241*	
user *px-sock-shop*	user-db *px-sock-shop*	
user *px-sock-shop*	user-db *px-sock-shop*	
load-test *px-sock-shop*	world *10.32.86.241*	
queue-master *px-sock-shop*	rabbitmq *px-sock-shop*	
queue-master *px-sock-shop*	rabbitmq *px-sock-shop*	

Source pod
orders-6ddd96f65-28nmv

Source identity
29972

Source labels
io.cilium.k8s.namespace.labels.kubernetes.io/metad
io.cilium.k8s.policy.cluster=gke-liz-network-policy-a
io.cilium.k8s.policy.serviceaccount=default
namespace=px-sock-shop
name=orders

Source IP
10.36.1.174

Destination identity
2

Figure 6-4. Cilium's Hubble UI shows network flows in a Kubernetes cluster

In a cloud native environment, where IP addresses are continually being dynamically assigned and reassigned, traditional network observability tools based on IP addresses are of very limited use. As a CNI, Cilium has access to workload identity information which means that Hubble can show service maps and flow data identified by Kubernetes pods, services, and namespaces. This is invaluable for diagnosing network issues.

If you can observe activity, this is a basis for security tools that compare what's happening with policies or rules to understand whether that activity is expected or suspicious. Let's turn to some tools that use eBPF to provide cloud native security capabilities.

Security

There are powerful cloud native tools available that enhance security by using eBPF to detect and even prevent malicious activity. I've considered these in two groups: securing network activity and securing the expected behavior of applications at runtime.

Network Security

Because eBPF allows inspecting and manipulating network packets, it has many uses in network security. The basic principle is that if a network packet is deemed to be malicious or problematic because it does not meet some security validation criteria, it can simply be dropped. eBPF is a highly efficient way to implement this because it can hook into the relevant parts of the network stack in the kernel, or even on the network interface card.[1] This means out-of-policy or malicious packets can be dropped before incurring the processing costs of being handled by the networking stack and passed to user space.

One of the early uses of eBPF in production at scale was for DDoS (distributed denial of service) protection at Cloudflare. A DDoS attacker floods a target machine with many network messages, in the hope that the target can't process them quickly enough and becomes so busy handling these messages that it can't do useful work. Cloudflare engineers use eBPF programs to examine packets as soon as they arrive, and quickly determine whether a packet is part of such an attack, discarding them if so. The packet doesn't have to pass through the kernel's networking stack so it takes far fewer resources to process, and the target can cope with a much higher rate of malicious traffic.

---

[1] XDP or eXpress Data Path hooks are supported by some network interface cards and drivers, allowing the eBPF program to be offloaded out of the kernel entirely.

eBPF programs have also been used as a dynamic mitigation against "packet of death" kernel vulnerabilities.[2] An attacker crafts a network packet in such a way that it exploits a bug in the kernel that prevents it from processing that packet properly. Rather than waiting for a kernel patch to roll out, the attack can be mitigated by loading an eBPF program that looks for these specifically crafted packets and drops them. The real beauty of this is that the eBPF program can be loaded dynamically without having to change anything on the machine.

In Kubernetes, network policy is a first-class resource, but it's left to the networking plug-in to enforce it. Some CNIs, including Cilium and Calico, offer extended network policy capabilities for more powerful rules, such as allowing or disallowing traffic to a destination specified by fully qualified domain name rather than just by IP address. There's a good tool for exploring network policies and their effects at *app.networkpolicy.io*, shown in Figure 6-5.

Standard Kubernetes network policy rules apply to traffic to and from application pods, but since eBPF has visibility over all network traffic, it can be used for host firewall capabilities too, restricting traffic to and from a host (virtual) machine.[3]

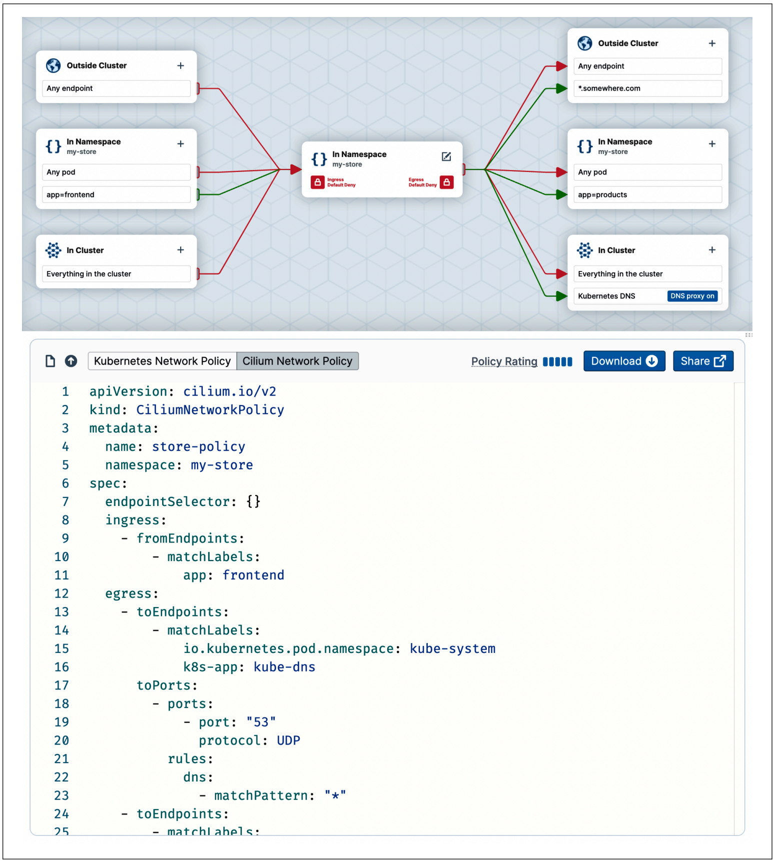
eBPF can also be leveraged to provide transparent encryption, whether through WireGuard or IPsec.[4] Here, *transparent* means that the application doesn't need any modifications—in fact the application can be entirely unaware that its network traffic is encrypted.

---

[2] Daniel Borkmann discussed this in his talk, "BPF as a Fundamentally Better Dataplane" (eBPF Summit (virtual), 2020).

[3] See Cilium's Host Firewall documentation.

[4] Tailscale has a comparison of these two encryption protocols.


Outside Cluster
Any endpoint
In Namespace
my-store
Any pod
app=frontend
In Cluster
Everything in the cluster
In Namespace
my-store
Ingress Default Deny
Egress Default Deny
Outside Cluster
Any endpoint
*.somewhere.com
In Namespace
my-store
Any pod
app=products
In Cluster
Everything in the cluster
Kubernetes DNS
DNS proxy on
Kubernetes Network Policy
Cilium Network Policy
Policy Rating
Download
Share

```
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: store-policy
  namespace: my-store
spec:
  endpointSelector: {}
  ingress:
    - fromEndpoints:
        - matchLabels:
            app: frontend
  egress:
    - toEndpoints:
        - matchLabels:
            io.kubernetes.pod.namespace: kube-system
            k8s-app: kube-dns
      toPorts:
        - ports:
            - port: "53"
              protocol: UDP
          rules:
            dns:
              - matchPattern: "*"
    - toEndpoints:
        - matchLabels:
```

Figure 6-5. The network policy editor shows a visual representation of the effects of a policy

Runtime Security

eBPF is also being used to build tools that detect when applications behave in unexpected or malicious ways, and some of these tools can also be used to prevent bad behavior. A few examples of suspicious behavior might include accessing files unexpectedly, running executable programs, or attempting to gain additional privileges.

In fact, you may well have used BPF-based security enforcement in the form of seccomp, a Linux feature for limiting the set of syscalls that any application can call.

The CNCF project Falco extended this idea of limiting the syscalls that an application can make. Falco's rule definitions are created in YAML, which is easier for humans to read and interpret than seccomp profiles. The default Falco driver is a kernel module, but there is also an eBPF probe driver that attaches to "raw syscall" events. It doesn't prevent those syscalls being completed, but it can generate logs or other notifications to alert operators to a potentially malicious event.

As we saw in Chapter 3, eBPF programs can be attached to the LSM interface to prevent malicious behaviors or to mitigate known vulnerabilities. For example, Denis Efremov wrote an eBPF program to prevent `exec*()` system calls being run if they are not passed any arguments, in order to mitigate the *PwnKit*[5] high-severity vulnerability. eBPF can also be used to mitigate against speculative execution "Spectre" attacks.[6]

Tracee is another open source project for runtime security using eBPF. As well as syscall-based checks, it also uses the LSM interface. This helps avoid being vulnerable to TOCTTOU race conditions that are possible when only checking syscalls. Tracee supports rules defined in Open Policy Agent's Rego language, and also allows for plug-in rules defined in Go.

The Tetragon component of Cilium offers another powerful approach, using eBPF to monitor the *four golden signals of container security observability*: process execution, network sockets, file access, and layer 7 network identity. This allows operators to see exactly what was responsible for any malicious or suspicious event, down to the executable name and user identity within a specific pod. For example, if you were subject to a cryptocurrency mining attack, you could see exactly what executable opened a network connection to a mining pool, from what pod, and when. These forensics are invaluable for understanding how the compromise happened and making it easy to build security policies to prevent similar attacks in future.

---

[5] See Bharat Jogi's blog, "PwnKit: Local Privilege Escalation Vulnerability" (Qualys, January 25, 2022).

[6] See Daniel Borkmann's talk, "BPF and Spectre: Mitigating Transient Execution Attacks" (eBPF Summit (virtual), August 18–19, 2021).

If you'd like to dive deeper into the topic of security observability with eBPF, check out the report[7] by Natália Ivánkó and Jed Salazar. Keep an eye on the cloud native eBPF space, as it won't be long before we see tools that leverage BPF LSM and other eBPF customization to provide security enforcement as well as observability.

We've taken a tour around several cloud native tools in networking, observability, and security. Their use of eBPF gives them two key advantages over previous generations:

1. From their vantage point in the kernel, eBPF programs have visibility across all processes.
2. By avoiding transitions between kernel and user space execution, eBPF programs provide an extremely performant way to collect event data or handle network packets.

This doesn't mean we should use eBPF for everything! It's unlikely to make sense to write business-specific applications in eBPF, any more than we would typically write applications as kernel modules. There may be some exceptions to this rule, perhaps for extremely high performance requirements like high-frequency trading. In the main, eBPF comes into its own for tooling that instruments other applications, as we've seen in this chapter.

---

[7] Natália Ivánkó and Jed Salazar, *Security Observability with eBPF* (O'Reilly, 2022).

CHAPTER 7

Conclusion

I hope this short report has given you an understanding of eBPF and why it's so powerful. What I really hope is that you're ready to try out some eBPF-based tools for yourself!

If you want to dive deeper on the technical side, a good place to start is *ebpf.io*, where you'll find more information about the technology and the eBPF Foundation. For coding examples, I have some resources in my ebpf-beginners repository on GitHub.

To learn about how others are leveraging eBPF tools, join events like eBPF Summit and Cloud Native eBPF Day where users share their successes and learnings. There is an active Slack channel that you can reach from *ebpf.io/slack*. I hope to see you there!

About the Author

Liz Rice is the chief open source officer with cloud native networking and security specialists Isovalent, creators of the Cilium eBPF-based networking project. She was chair of the CNCF's Technical Oversight Committee from 2019–2022, and was co-chair of KubeCon + CloudNativeCon in 2018. She is also the author of *Container Security*, published by O'Reilly. She has a wealth of software development, team, and product management experience from working on network protocols and distributed systems, and in digital technology sectors such as VOD, music, and VoIP. When not writing code, or talking about it, Liz loves riding bikes in places with better weather than her native London, and competing in virtual races on Zwift.