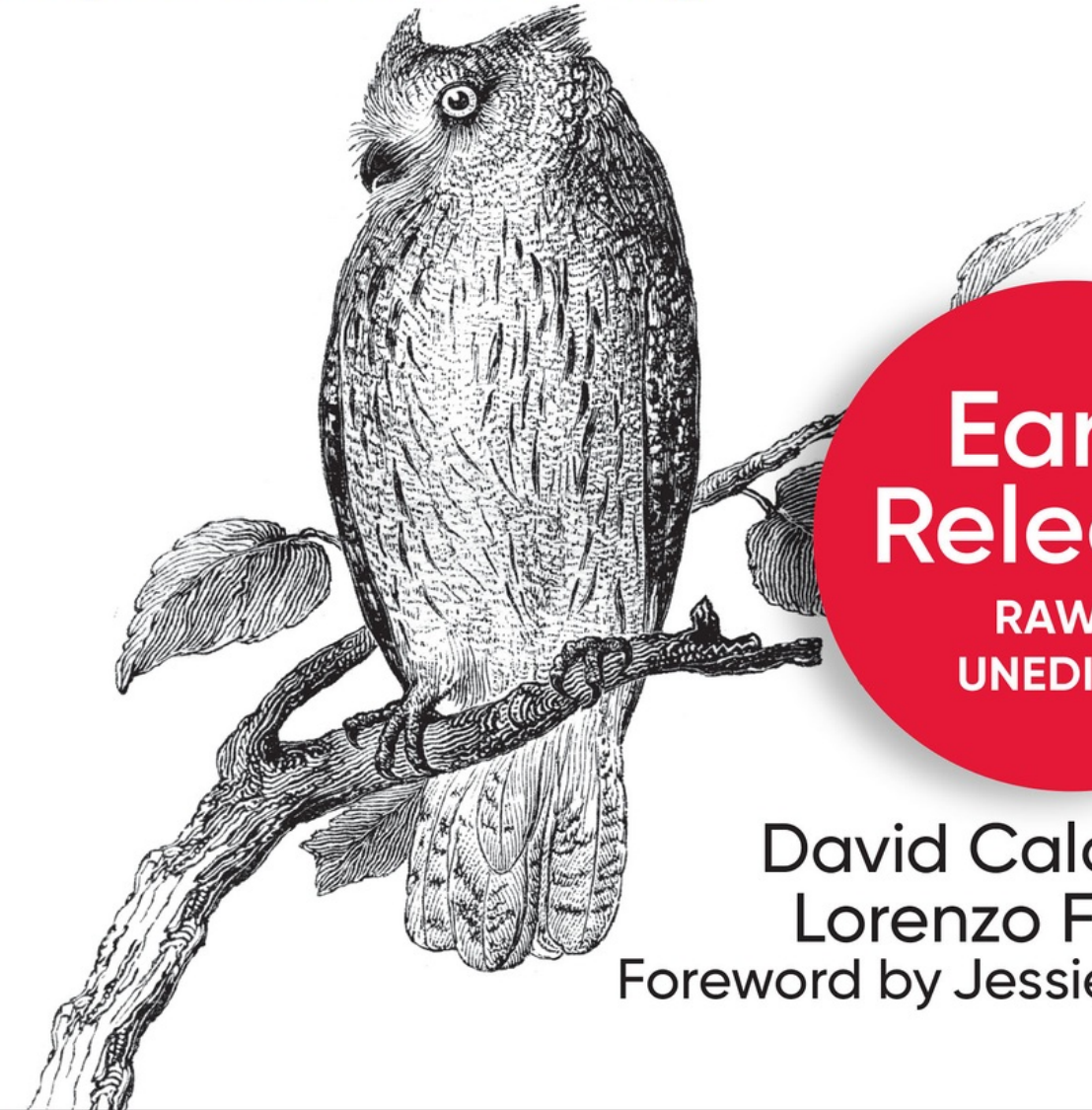
O'REILLY®

Linux Observability with BPF

Advanced Programming for Performance Analysis and Networking



Early Release

RAW & UNEDITED

David Calavera &
Lorenzo Fontana
Foreword by Jessie Frazelle

Linux Observability with BPF

FIRST EDITION

Advanced Programming for Performance Analysis and Networking

David Calavera and Lorenzo Fontana

Linux Observability with BPF

by David Calavera and Lorenzo Fontana

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.

Editors: John Devins and Eleanor Bru

Production Editor: Katherine Tozer

Copyeditor: TK

Proofreader: TK

Indexer: TK

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

October 2019: First Edition

Revision History for the First Edition

- 2019-04-04: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492050209 for release details.

978-1-492-05020-9

[TK]

Chapter 1. Running Your First BPF Programs

The BPF virtual machine is capable of running instructions in response to events triggered by the kernel. However, now all BPF programs have access to all events triggered by the kernel. When you load a program into the BPF virtual machine, you need to decide which type of program you're running. This informs the kernel about where your program is going to be triggered. It also tells the BPF verifier which helpers are going to be allowed in your program. When you choose the program type, you're also choosing the interface that your program is implementing. This interface ensures that you have access to the right type of data, and whether your program can access network packets directly or not.

In this chapter, we're going to show you how to write your first BPF programs. We'll also guide you around the different types of BPF programs that you can create at the moment of writing this book. Over the years, the kernel developers have been adding different entry points that you can attach BPF programs to. This work is not complete yet, and they find new ways to leverage BPF every day. We're going to focus in some of the most useful type of programs in this chapter with the intention of giving you a taste of what you can do with BPF. We'll go over many additional examples in future chapters on how to write BPF programs.

This chapter is also going to cover the role that the BPF verifier plays in running your programs. This component validates that your code is safe to execute, and helps you to write programs that won't cause unexpected

results, like memory exhaustion or suddent kernel crashes.

Writing BPF programs

The most common way to write BPF programs is by using a subset of C compiled with LLVM. LLVM. LLVM is a general purpose compiler that can emit different types of bytecode. In this case, LLVM will output BPF assembly code that we will load into the kernel later. We're not going to show you BPF assembly in this chapter, we've included an appendix to this book on BPF assembly that you can use as a reference if you need it. We'll also show very simple examples of BPF assembly in future chapters, where writing assembly is more useful than a C program.

The kernel provides the system call `bpf` to load programs into the BPF virtual machine once they are compiled. This system call is used for other operations besides loading programs, and we'll see more usage examples in future chapters. The kernel also provides several utilities that abstract the loading of bpf programs for you. In this first code example we're going to use those helpers to show you the hello world example of BPF.

```
1 #include <uapi/linux/bpf.h>
2 #define SEC(NAME) __attribute__((section(NAME), used))
3
4 SEC("tracepoint/syscalls/sys_enter_execve")
5 int bpf_prog(void *ctx) {
6   char msg[] = "Hello, BPF World!";
7   bpf_trace_printk(msg, sizeof(msg));
8   return 0;
9 }
10
11 char _license[] SEC("license") = "GPL";
```

There are a few interesting concepts in this first program. We're using the attribute `SEC` to inform the BPF virtual machine when we want to run this

program. In this case, we will run this BPF program a tracepoint in an execve system call is detected. Tracepoints are static marks in the kernel's binary code that allow developers to inject code to inspect the kernel's execution. We will talk in detail about tracepoints in Chapter 4. Execve is an instruction that executes other programs. So we're going to see the message "`Hello, BPF World!`" every time the kernel detects that a program executes other program.

At the end of this example we're also specifying the license for this program. Since the Linux Kernel is licensed under GPL, it can only load programs licensed as GPL too, if we set the license to something else, the kernel will refuse to load our program. We're using `bpf_trace_printk` to print a message in the kernel tracing log, you can find this log in `/sys/kernel/debug/tracing/trace_pipe`.

We're going to use `clang` to compile this first program into a valid elf binary file. This is the format that the kernel expects to load. We're going to save our first program in a file called `hello_world_kern.c` so we can compile it:

```
clang -O2 -target bpf -c hello_world_kern.c -o hello_world_kern.o
```

You'll find some scripts to compile these programs in the GitHub repository with the code example for the book, you don't have to memorize this `clang` command.

Now that we have compiled our first BPF program, we need to load it in the kernel. As we mentioned before, we're going to use a special helper that the kernel provides to abstract the boilerpate of compiling and loading the program, this helper is called `load_bpf_file`. This helper take a binary file and tries to load it in the kernel. You can find this helper in the

GitHub repository with all the examples in the book, in the `bpf_load.h` file.

```
#include <stdio.h>
#include "bpf_load.h"

int main(int argc, char **argv) {
  if (load_bpf_file("hello_world_kern.o") != 0) {
    printf("The kernel didn't load the BPF program\\n");
    return -1;
  }

   read_trace_pipe();

   return 0;
}
```

We're going to use `gcc` to compile this program and link it as an elf binary. In this case, we don't need to specify a target, since this program won't be loaded in the BPF virtual machine:

```
gcc -O2 -Wall -g -lelf -o hello_world_user hello_world_user.c
```

If you want to run this program, you can execute this final binary with sudo: `sudo ./hello_world_user`. If you don't run it with sudo, you'll get an error message because the BPF program cannot be loaded in the kernel by a non privileged user by default.

When you run this program, you'll start to see our hello world message after a few seconds, even if you're not doing anything with your computer. This is because programs running behind the scenes in your computer might be executing other programs.

When you stop this program, the message will stop showing up in your terminal. BPF programs are unloaded from the virtual machine as soon as

the programs that load them terminate. In future chapters we'll explore how to make BPF programs persistent, even after their loaders terminate, but we don't want to introduce too many concepts just yet.

Now that you've seen the basic structure for a BPF program, we can dive into which types of programs you can write, which will give you access to different subsystems within

BPF program types

Although there is no clear categorization within programs, you'll quickly realize that all the types we're going to cover in this section belong to two different categories.

The first category is tracing. Many of the programs that you can write will help you understand what's happening in your system better. They give you direct information about the behavior of your system and the hardware it's running on.

The second category is networking. This type of programs will allow your to inspect and manipulate the network traffic in your system. Different type of programs can be attached to different stages of network processing within the kernel. This has advantages and disadvantages. For example, you can attach BPF programs to network events as soon as your network driver receives a package, but this program will have access to less information about the packet, because the kernel doesn't have enough information to offer you yet. On the other end of the expectrum, you can attach BPF programs with network events right before they are passed to user-space. In this case, you'll have much more information about the packet that will help you make better informed decisions, but you'll have

to pay the cost of completely processing the packet.

The list we're going to show you next is not divided in categories, we're introducing these types in the chronological order that they were added to the kernel. We've moved the least used of these programs to the end of this section, and we're going to focus for now on the ones that will be more useful for you. You can learn more about all of them in the BPF documentation if you're curious about any program that we're not covering in detail here.

BPF_PROG_TYPE_SOCKET_FILTER

BPF_PROG_TYPE_SOCKET_FILTER was the first program type to be added to the Linux Kernel. When you attach a BPF program to a raw socket, you get access to all the packets processed by that socket. Socket filter programs don't allow you to modify the contents of those packets, or to change the destination for those packets, but they allow you to decide whether a packet in a socket should be delivered or not.

BPF_PROG_TYPE_KPROBE

As we saw in the chapter about tracing, Kprobes are functions that you can attach dynamically to certain call points in the kernel. BPF Kprobe program types allow you to use BPF programs as Kprobe handlers. The BPF virtual machine ensures that your Kprobe programs are always safe to run, which is an advantage from traditional Kprobe modules. You still need to remember that Kprobes are not considered stable entry points in the kernel, so you'll need to ensure that your Kprobe BPF programs are compatible with the specific kernel versions that you're using.

When you write a BPF program that's attached to a Kprobe, you need to

decide whether it will be executed right as the first instruction in the function call, or when the call completes. You need to declare this behavior in the section header for your BPF program. For example, if you want to inspect the arguments when the kernel invokes an `exec` syscall, you'll attach the program at the beginning of the call. In this case, you need to set the section header `SEC("kprobe/sys_exec")`. If you want to inspect the returned value of invoking an `exec` syscall, you need to set the section header `SEC("kretprobe/sys_exec")`.

We'll talk a lot more about Kprobes in future chapters of this book. They are a fundamental piece to understanding tracing with BPF.

BPF_PROG_TYPE_TRACEPOINT

This type of programs allow you to attach BPF programs to the tracepoint handler provided by the kernel. As we saw in the tracing chapter, tracepoints are static marks in the kernel's code base that allow you to inject arbitrary code for tracing and debugging purposes. They are less flexible than Kprobes, because they need to be defined by the kernel before hand, but they are guaranteed to be stable after their introduction in the kernel. This gives you a much higher level of predictability when you want to debug your system.

All tracepoints in your system are defined inside the directory `/sys/kernel/debug/tracing/events`. There you'll find each subsystem that includes any tracepoints, and that you can attach a BPF program to. One interesting fact is that BPF declares its own tracepoints, so you can write BPF programs that inspect the behavior of other BPF programs. The BPF tracepoints are defined in `/sys/kernel/debug/tracing/events/bpf`. There, for example, you can

find the tracepoint definition for `bpf_prog_load`. Which means that you can write a BPF program that inspects when other BPF programs are loaded.

Like Kprobes, tracepoints are another fundamental piece to understand tracing with BPF. We'll talk more about them in future chapters of this book, and we'll show you how to write programs to take advantage of them.

BPF_PROG_TYPE_XDP

XDP programs allow you to write code that's executed very early on when a network packet arrives to the kernel. It exposes only a very limited set of information from the packet since the kernel has almost not had much time to process the information itself. By being executed early on, you have much more level of control over how to handle that packet.

XDP programs define several actions that you can control, and that allow you to decide what to do with the packet. You can return XDP_PASS from your XDP program, which means that the packet should be passed to the next subsystem in the kernel. You can also return XDP_DROP, which means that the kernel should ignore this packet completely and do nothing else with it. And you can also return XDP_TX, which means that the packet should be forwarded back to the Network Interface Card(NIC) that received the packet in the first place.

This level of control opens the doors to many interesting programs in the networking layer. XDP has become one of the main components in BPF, that's why we've included a specific chapter about it in this book. In that chapter, we'll talk about many powerful use cases for XDP, like implementing programs to protect your network against Distributed Denial

of Service(DDOS) attacks.

BPF_PROG_TYPE_PERF_EVENT

These types of BPF programs allow you to attach your BPF code to Perf events. Perf is an internal profiler in the kernel that emits performance data events for Hardware and Software. You can use it to monitor many things, from your computer's CPU to any software running on your system. When you attach a BPF program to Perf events, your code will be executed every time Perf generates data for you to analyze.

BPF_PROG_TYPE_CGROUP_SKB

These types of programs allow you to attach BPF logic to Control Groups(CGroups). The allow Cgroups to control network traffic within the processes that they contained. With these programs you can decide what to do with a network packet before it's delivered to a process in the Cgroup. At the same time you can also decide what to do when a process in the Cgroup sends a network packet.

As you can see their behavior is very similar to BPF_PROG_TYPE_SOCKET_FILTER programs. The only main difference is that BPF_PROG_TYPE_CGROUP_SKB programs are attached to all processes within a control group, rather than a specific processes.

BPF_PROG_TYPE_CGROUP_SOCK

These types of programs allow you to execute code when any process in a CGroup opens a network socket. This behavior is very similar to the programs attached to CGroup socket buffers, but instead of giving you

access to the packets as the come through the network, they allow you to control what happens when a process opens a new socket.

BPF_PROG_TYPE_SOCK_OPS

These types of program allow you to modify socket connection options at runtime while a package transits through several stages in the kernel's networking stack. They are attached to Cgroups, much like BPF_PROG_TYPE_CGROUP_SOCK and BPF_PROG_TYPE_CGROUP_SKB, but unlike those program types, they can be invoked several times during the connection's lifecycle.

When you create a BPF program with this type, your function call receives an argument called "`op`" that represents the operation that the kernel is about to execute with the socket connection, so you can know at which point the program is invoked in the connection's lifecycle. With this information at hand, you can access data like network IPs and connection ports, and modify the connection options to set timeouts and alter the Round-trip delay time for a given packet.

For example, Facebook uses this to set very short recovery time objectives(RTO) for connections within the same datacenter. Recovery Time Objective is the time that a system, or network connection in this case, is expected to be recovered after a failure. This objective also represents how long the system can be unavailable before suffering from unacceptable consequences. In their case, they assume that machines in the same datacenter should have very short RTO, and they modify this threshold by using a BPF program.

BPF_PROG_TYPE_SK_SKB

BPF_PROG_TYPE_SK_SKB programs give you access to socket maps and socket redirects. As you'll learn in the next chapter, socket maps allow you to keep references to several sockets. When you have this references you can use special helpers to redirect incoming packet from a socket to a different socket.

BPF_PROG_TYPE_CGROUP_DEVICE

This type of programs allow you to decide whether an operation within a control group can be executed for a given device or not. The first implementation of Cgroups (v1) has a mechanism that allows you set permissions for specific devices, however, the second iteration of Cgroups lack this feature. This type of programs were introduced to supply that functionality. At the same time, being able to write a BPF program gives you more flexibility to set those permissions when you need them.

BPF_PROG_TYPE_SK_MSG

This type of programs let you control whether a message send to a socket should be delivered or not. When the kernel creates a socket, it stores them in what's called a socket map. This map gives the kernel quick access to specific groups of sockets. When you attach a socket message BPF program to a socket map, all messages send to those sockets will be filtered by the program before delivering them. Before filtering messages, the kernel copies the data in the message so you can read it and decide what to do with it. These programs have two possible return values, SK_PASS and SK_DROP, you'd use the first one if you want to allow sending the message to the socket, and the later one if you want the kernel to ignore the message and not deliver it to the socket.

BPF_PROG_TYPE_RAW_TRACEPOINT

We've already seen a type of programs that access tracepoints in the kernel. The kernel developers added a new tracepoint program to address the need of accessing the tracepoint arguments in the raw format hold by the kernel. This format gives you access to more detailed information about the task that the kernel is execution, however, it has a small performance overhead. Most of the time, you'll want to use regular tracepoints in your programs, but it's good to keep in mind that you can also access the raw arguments when needed by using raw tracepoints.

BPF_PROG_TYPE_CGROUP_SOCK_ADDR

This type of programs allow you to manipulate the IPs and port numbers that user-space programs are attached to when they are control by specific Cgroups. There are use cases when you system uses several IPs when you want to ensure that a specific set of user-space programs use the same IP and port. These BPF programs give you the flexibility to manipulate those bindings when you put those user-space programs in the same Cgroup. This ensures that all incoming and outgoing connections from those applications use the IP and port that the BPF program provides.

BPF_PROG_TYPE_SK_REUSEPORT

SO_REUSEPORT is an option in the kernel that allows multiple processes in the same host to be bound to the same port. This option allows higher performance in accepted network connections when you want to distribute load across multiple threads.

The BPF_PROG_TYPE_SK_REUSEPORT program type allows you to write BPF programs that hook into the logic that the kernel uses to decide whether it's going to reuse a port or not. You can prevent programs from reusing the same port if your BPF program returns SK_DROP, and you

also can inform the kernel to follow its own reuse routine when you return SK_PASS from these BPF programs.

BPF_PROG_TYPE_FLOW_DISSECTOR

The flow dissector is a component of the kernel that keeps track of the different layers that a network packet needs to go through from when it arrives to your system to when it's delivered to a user-space program. It allows to control the flow of the packet using different classification methods. The built-in dissector in the kernel is called Flower classifier, and it's used by firewalls and other filtering devices to decide what to do with specific packets.

BPF_PROG_TYPE_FLOW_DISSECTOR programs are designed to hook logic in the flow dissector path. They provide security guarantees that the built-in dissector cannot provide, like ensuring that the program always terminates, which might not be guaranteed in the built-in dissector. These BPF programs can modify the flow network packets follow within the kernel.

Uncovered programs

These are programs that we're not going to get into in much details:

BPF_PROG_TYPE_SCHED_CLS AND BPF_PROG_TYPE_SCHED_ACT

This type of BPF programs allow you to classify network traffic and modify some properties of the packets in the socket buffer.

BPF_PROG_TYPE_LWT_IN, BPF_PROG_TYPE_LWT_OUT, BPF_PROG_TYPE_LWT_XMIT AND BPF_PROG_TYPE_LWT_SEG6LOCAL

Attach BPF programs to the kernel's lightweight tunnel infrastructure.

BPF_PROG_TYPE_LIRC_MODE2

Attach BPF programs to connections to Infra Red devices, like remote controllers, for fun.

The BPF verifier

Allowing anyone to execute arbitrary code inside the Linux Kernel always sounds like a terrible idea at first. The risk of running BPF programs in production systems would be too high if it wasn't for the BPF verifier. In the words of Dave S. Miller, one of the Kernel Networking maintainers:

```
The only thing sitting between our eBPF programs and a deep dark chasm
of destruction is the eBPF verifier.
```

Obviously, the BPF verifier is also a program running on your system, and it's the object of high scrutiny to ensure that it does its job correctly. In the past years, security researchers have discovered some vulnerabilities in the verifier that allowed attackers to access random memory in the kernel, even as unprivileged users. You can read more about vulnerabilities like that one in the Common Vulnerabilities and Exposures(CVE) catalog, a list of known security threads sponsored by the United States Department of Homeland Security. For example, CVE-2017-16995 describes how any user could read and write kernel memory bypassing the BPF verifier.

In this section, we're going to guide you through the measures that the verifier takes to prevent problems like the one described above.

The first check that the verifier performs is a static analisys of the code that the virtual machine is going to load. The objective of this first check

is to ensure that the program has an expected end. To do this, the verifier creates a Direct Acyclic Graph(DAG) with the code and performs a Depth First Search(DFS) to ensure that the program finishes and the code doesn't include dangerous paths. These are the conditions why the verifier might reject your code during this first check:

- The program includes control loops. To ensure that the program doesn't get stuck in an infinite loop, the verifier rejects any kind of control loop. There have been proposals to allow loops in BPF programs, but at the time of writing this book, none of them have been adopted.
- The program doesn't try to execute more instructions than the maximum allowed by the kernel. At this time, the maximum number of instructions to execute is 4096. This limitation is in place to prevent BPF to run forever too. In the next chapter, we'll talk about how to nest different BPF programs to work around this limitation in a safe way.
- The program doesn't include any unreachable instruction, like conditions or functions that are never executed. This prevents loading dead code in the virtual machine, which would also delay and termination of the BPF program.
- The program doesn't try to jump outside its bounds.

The second check that the verifier performs is a dry run of the BPF program. This means that the verifier will try to analyze every instruction that the program is going to execute to ensure that it doesn't execute any invalid instruction. This dry execution also checks that all memory pointers are accessed and de-referenced correctly. Finally, the dry run informs the verifier about the control flows in the program to ensure that no matter which control path the program takes, it arrives to the BPF_EXIT instruction. In order to do this, the verifier keeps track of all visited branch paths in a stack, which it evaluates before taking a new path

to ensure that it doesn't visit a specific path more than once. Once these two checks pass, the verifier considers that the program is safe to execute.

The `bpf` system call allows you to debug the verifier's checks if you're interested in seeing how your programs are analyzed. When you load a program with this system call, you can set several attributes that will make the verifier to print its operation log:

```
union bpf_attr attr = {
  .prog_type = type,
  .insns     = ptr_to_u64(insns),
  .insn_cnt  = insn_cnt,
  .license   = ptr_to_u64(license),
  .log_buf   = ptr_to_u64(bpf_log_buf),
  .log_size  = LOG_BUF_SIZE,
  .log_level = 1,
};

bpf(BPF_PROG_LOAD, &attr, sizeof(attr));
```

The `log_level` field tells the verifier whether to print any log or not. It will print its log when you set it to 1, and it won't print anything if you set it to 0. If you want to print the verifier log, you also need to provide a log buffer and its size. This buffer is a multi-line string that you can print to inspect the decisions that the verifier took.

The BPF verifier plays a big role in keeping your system secure and available while you run arbitrary programs inside the kernel, although, tt might be hard to understand why it takes some decisions sometimes. Don't desperate if you bump into verification issues trying to load your programs. During the rest of this book we'll guide you though safe examples that will help you understand how to write your own programs in a secure way too.

Conclusion

In this chapter we've guided you through the first code examples to understand BPF programs. We've also given you a glance of all the type of programs that you can write with BPF. Don't worry if some of the concepts presented here don't make sense yet, as we advance though the book, we'll show you more examples of those programs. We've also covered the important verification steps that BPF takes to ensure that your programs are safe to run.

In the next chapter, we're going to dive a little bit more in those programs with more examples. We're also going to talk about how BPF programs communicate with their counterparts in user-space, and how they share information.

Chapter 2. Understanding BPF Data Structures and User-Space Communication

Message passing for invoking behavior in a program is a widely used technique in Software Engineering. A program can modify another program's behavior by sending messages, this also allows the exchange of information between those programs. One of the most fascinating aspects about BPF, is that the code running on the kernel and the program that loaded said code can communicate with each other at runtime using message passing.

In this chapter you'll learn how BPF programs and user-space programs can talk to each other. We'll describe the different channels of communication between the kernel and user-space, and how they store information. We'll also show you use cases for those channels, and how to make the data in those channels persistent between programs initialization.

BPF Maps are key/value stores that live in the kernel. They can be accessed by any BPF program that knows about them. Programs that run in user-space can also access these maps by using file descriptors. You can store any kind of data in a map, as long as you specify the data size correctly before hand. The kernel treats keys and values as binary blobs and it doesn't care about what you keep in a map.

The BPF verifier includes several safeguards to ensure that the way you

create and access maps is safe. We'll talk about these guarantees when we explain how to access data in these maps.

Creating BPF Maps

The most direct way to create a BPF map is by using the `bpf` syscall. When the first argument in the call is `BPF_MAP_CREATE`, you're telling the kernel that you want to create a new map. This call will return the file descriptor identifier associated to the map you just created. The second argument in the syscall is the configuration for this map.

```
union bpf_attr {
  struct {
    __u32 map_type;     /* one of the values from bpf_map_type */
    __u32 key_size;     /* size of the keys, in bytes */
    __u32 value_size;   /* size of the values, in bytes */
    __u32 max_entries;  /* maximum number of entries in the map */
    __u32 map_flags;    /* flags to modify how we create the map */
  };
}
```

The third argument in the syscall is the size of this configuration attribute.

For example, you can create a hash-table map to store unsigned integers as keys and values:

```
union bpf_attr my_map {
   .map_type = BPF_MAP_TYPE_HASH,
   .key_size = sizeof(int),
   .value_size = sizeof(int),
   .max_entries = 100,
   .map_flags = BPF_F_NO_PREALLOC,
};

int fd = bpf(BPF_MAP_CREATE, &my_map, sizeof(my_map));
```

If the call fails, the kernel returns a value of `-1`. There might be three

reasons why it fails. If one of the attributes is invalid, the kernel sets the `errno` variable to EINVAL. If the user executing operation doesn't have enough privileges, the kernel sets the `errno` variable to EPERM. Finally, if there is no enough memory to store the map, the kernel sets the `errno` variable to ENOMEM.

ELF conventions to create BPF Maps

The kernel includes several conventions and helpers to generate and work with BPF Maps. You'll probably find these conventions more frequently than direct syscall executions because they are more readable and easy to follow. Keep in mind that these conventions still use the `bpf` syscall to create the maps, even when ran directly in the kernel, and you'll find using the syscall directly more useful if you don't know which kind of maps you're going to need beforehand.

The helper function `bpf_map_create` wraps the code we just saw to make it easier to initialize maps on demand. We can use it to create the previous map with only one single line of code:

```
1 int fd;
2 fd = bpf_create_map(BPF_MAP_TYPE_HASH, sizeof(int), sizeof(int), 100,
BPF_F_NO_PREALOC);
```

If you know which kind of map you're going to need in your program, you can also pre-define it in advance. This is very helpful to get more visibility in the maps your program is using beforehand:

```
1 struct bpf_map_def SEC("maps") my_map = {
2       .type        = BPF_MAP_TYPE_HASH,
3       .key_size    = sizeof(int),
4       .value_size  = sizeof(int),
5       .max_entries = 100,
```

```
6       .map_flags    = BPF_F_NO_PREALLOC,
7 };
```

When you define a map in this way, you're using what's called a "`section attribute`", in this case `SEC(``"``maps``"``)`. This macro tells the kernel that this structure is a BPF Map and it should be created accordingly.

You might have noticed that we don't have the file descriptor identifier associated with the map in this new example. In this case, the kernel uses a global variable called `map_data` to store information about the maps in your program. This variable is an array of structures, and it's ordered by the order in which you specified each map in your code. For example, if the previous map was the first one specified in your code, you'd get the file descriptor identifier from the first element in the array:

```
fd = map_data[0].fd;
```

You can also access the map's name and its definition from this structure, this information is sometimes useful for debugging and tracing purposes.

Once you have initialized the map, you can start sending messages between the kernel an user-space with them.

Working with BFP Maps

Communication between the kernel and user-space is going to be a fundamental piece in every BPF program you write. The APIs to access maps differ when you're writing the code for the kernel and when you're writing the code for the user-space program. In this section, we're going to introduce the semantics and specific details of each implementation.

Updating elements in a BPF Map

One of the first things you'll probably want to do is to write information into your maps. The kernel helpers provide the function `bpf_map_update_elem` for this purpose. This function's signature is different if you load it from `bpf/bpf_helpers.h`, inside the program running on the kernel, than if you load it from `tools/lib/bpf/bpf.h`, inside the program running in user-space. It's behavior is also slightly different, the code running on the kernel can access the map in memory directly, and it's able to update elements atomically in place. While the code running in user-space has to send the message to the kernel, which copies the value supplied before updating the map. This function returns 0 when the operation succeeds, and it returns a negative number when it fails. In case of failure, the global variable `errno` is populated with the failure cause.

The `bpf_map_update_elem` function within the kernel takes four arguments. The first one is the pointer to the map we've already defined. The second one is a pointer to the key we want to update. Since the kernel doesn't know the type of key we're updating, this method is defined as an opaque pointer to `void`, which means that we can pass any data. The third argument is the value we want to insert. This argument uses the same semantics as the key argument, we'll show you some advance examples through this book on how to take advantage of opaque pointers. You can use the fourth argument in this function to change the way the map is updated. This argument can take three values. If you pass `0`, you're telling the kernel that you want to update the element if it exists, or it should create the element in the map if it doesn't exist. If you pass `1`, you're telling the kernel to create the element only when it doesn't exist. If you pass `2`, the kernel will only update the element when it exists. These values are defined as constants that you can also use, instead of having to

remember the integer semantics. `BPF_ANY` for `0`, `BPF_NOEXIST` for `1`, and `BPF_EXIST` for `2`.

Let's use the map we defined in the previous section to write some examples. In our first example, we're going to add a new value to the map. Since the map is empty, we can assume that any update behavior is good for us:

```
int key, value, result;
key = 1, value = 1234;

result = bpf_map_update_elem(&my_map, &key, &value, BPF_ANY);
if (result == 0)
  printf("Map updated with new element\n");
else
  printf("Failed to update map with new value: %d (%s)\n", result,
strerror(errno));
```

In this example we're using `strerror` to describe the error set in the `errno` variable. You can learn more about this function using the manual pages in your computer with `man strerror`.

Now let's see which result we get when we try to create an element with the same key:

```
int key, value, result;
key = 1, value = 5678;

result = bpf_map_update_elem(&my_map, &key, &value, BPF_NOEXIST);
if (result == 0)
  printf("Map updated with new element\n");
else
  printf("Failed to update map with new value: %d (%s)\n", result,
strerror(errno));
```

Since we have already created an element with key `1` in our map, the result

from calling `bpf_map_update_elem` will be `-1`, and the `errno` value will be `EEXIST`. This program will print this in the screen:

```
Failed to update map with new value: -1 (File exists)
```

Similarly, let's change this program to try to update an element that doesn't exist yet:

```
int key, value, result;
key = 1234, value = 5678;

result = bpf_map_update_elem(&my_map, &key, &value, BPF_EXIST);
if (result == 0)
  printf("Map updated with new element\n");
else
  printf("Failed to update map with new value: %d (%s)\n", result,
strerror(errno));
```

With the flag `BPF_EXIST`, the result of this operation is going to be `-1` again. The kernel will set the `errno` variable to `ENOENT`, and the program will print:

```
Failed to update map with new value: -1 (No such file or
directory)
```

These examples show how you can update maps from within the kernel program. You can also update maps from within user-space program. The helpers to do these are very similar to the ones we just saw, the only difference is that they use the file descriptor to access the map, rather than using the pointer to the map directly. As you remember, user-space programs always access maps using file descriptors. So in our examples, we'd replace the argument `my_map` with the global file descriptor identifier `map_data[0].fd`. Let's see how the original code looks like in this case:

```
int key, value, result;
key = 1, value = 1234;

result = bpf_map_update_elem(map_data[0].fd, &key, &value, BPF_ANY);
if (result == 0)
  printf("Map updated with new element\n");
else
  printf("Failed to update map with new value: %d (%s)\n", result,
strerror(errno));
```

Reading elements from a BPF Map

Now that we've populated our map with new elements, we're can start reading them from other points in our code. The reading api will look familiar after learning about `bpf_map_update_element`.

BPF also provides two different helpers to read from a map depending on where your code is running. Both helpers are called `bpf_map_lookup_elem`. Like the update helpers, they differ in their first argument, the kernel method takes a reference to the map, while the use space helper takes the map's file descriptor identifier as its first argument. Both methods return an integer to represent whether the operation failed or succeeded, just like the update helpers. The third argument in these helpers is a pointer to the variable in your code that's going to store the value read from the map. Let's see two examples following the code we saw in the previous section.

The first example reads the value inserted in the map when the bpf program is running on the kernel:

```
int key, value, result; // value is going to store the expected element's
value
key = 1;

result = bpf_map_lookup_elem(&my_map, &key, &value);
```

```
5 if (result == 0)
6   printf("Value read from the map: '%d'\n", value);
7 else
8   printf("Failed to read value from the map: %d (%s)\n", result,
strerror(errno));
```

If they key we were trying to read, `bpf_map_lookup_elem` would return a negative number, and it would set the error in the errno variable. For example, if we had not inserted the value before trying to read it, the kernel would have returned the not found error `ENOENT`.

This second example is very similar to the one we just saw, but this time we're reading the map from the program running in user-space:

```
1 int key, value, result; // value is going to store the expected element's
value
2 key = 1;
3
4 result = bpf_map_lookup_elem(map_data[0].fd, &key, &value);
5 if (result == 0)
6   printf("Value read from the map: '%d'\n", value);
7 else
8   printf("Failed to read value from the map: %d (%s)\n", result,
strerror(errno));
```

As you can see, we've replaced the first argument in `bpf_map_lookup_elem` with the map's file descriptor identifier. The helper behavior is exactly the same than the previous example.

Removing element from a BPF Map

The third operation we can execute on maps is for removing elements. Like with writing and reading elements, BPF gives us two different helpers to remove elements, both called `bpf_map_delete_element`. Like in the previous examples, these helper use the direct reference to the map

when you use them in the program running on the kernel, and the map's file descriptor identifier when you use them in the program running on user-space.

The first example deletes the value inserted in the map when the bpf program is running on the kernel:

```
int key, result;
key = 1;

result = bpf_map_delete_element(&my_map, &key);
if (result == 0)
  printf("Element deleted from the map\n");
else
  printf("Failed to delete element from the map: %d (%s)\n", result,
strerror(errno));
```

If the element that you're trying to delete doesn't exist, the kernel returns a negative number. In that case, it also populates the `errno` variable with the not found error `ENOENT`.

This second example deletes the value when the bpf program is running on user-space:

```
int key, result;
key = 1;

result = bpf_map_delete_element(map_data[0].fd, &key);
if (result == 0)
  printf("Element deleted from the map\n");
else
  printf("Failed to delete element from the map: %d (%s)\n", result,
strerror(errno));
```

You can see that we've changed the first argument again to use the file descriptor identifier. It's behavior is going to be consistent with the

kernel's helper.

Iterating over elements in a BPF Map

The final operation we're going to see in this section is going to help you find arbitrary elements in a BPF. There will be occasions when you don't know exactly the key for the element you're looking for, or you just want to see what's inside a map. BPF provides an instruction for this called `bpf_map_get_next_key`. Unlike the helpers we've seen until now, this instruction is only available for programs running on user-space.

This helper gives you a deterministic way to iterate over the elements on a map, but it's behavior is less intuitive than iterators in most programming languages. It takes three arguments, the first one is the map's file descriptor identifier, like the other user-space helpers we've seen until now. The next two arguments are where it becomes tricky. According to the official documentation, the second argument `key` is the identifier you're looking for, and the third one, `next_key` will be the next key in the map. We prefer to call the first argument `lookup_key`, it's going to become apparent why in a second. When you call this helper, BPF tries to find the element in this map which key you pass as the lookup key, then, it sets the `next_key` argument with the adjacent key in the map. So if you want to know which key comes after the key `1`, you need to set `1` as your lookup key, and if the map has an adjacent key to this one, bpf will set it as the value for the `next_key` argument.

Before seeing how `bpf_map_get_next_key` works in an example, let's add a few more elements to our map:

```
1 int new_key, new_value, it;
2
```

```
for (it = 2; it < 6 ; it++) {
  new_key = it;
  new_value = 1234 + it;
  bpf_map_update_elem(map_data[0].fd, &new_key, &new_value, BPF_NOEXIST);
}
```

If you want to print all the values in the map, you can use `bpf_map_get_next_key` with a lookup key that doesn't exist in the map. That will make BPF to start from the beginning of the map:

```
int next_key, lookup_key;
lookup_key = -1;

while(bpf_map_get_next_key(map_data[0].fd, &lookup_key, &next_key) == 0) {
  printf("The next key in the map is: '%d'\n", next_key);
  lookup_key = next_key;
}
```

This code will print something like this:

```
The next key in the map is: '1'
The next key in the map is: '2'
The next key in the map is: '3'
The next key in the map is: '4'
The next key in the map is: '5'
```

You can see how we're assigning the next key to `lookup_key` at the end of the loop, that way we continue iterating over the map until we reach the end. When `bpf_map_get_next_key` arrives at the end of the map, the value returned is a negative number and the `errno` variable is set to `ENOENT`., this will abort the loop execution.

As you can imagine, `bpf_map_get_next_key` can lookup keys starting at any point in the map, you don't need to start at the beginning of the map if you only want the next key for another specific key.

The tricks that `bpf_map_get_next_key` can play on you don't end here, there is another behavior that you need to be aware of. Many programming languages copy the values in a map before iterating over its elements. This prevents unknown behaviors if some other code in your program decides to mutate the map. This is specially dangerous if that code deletes elements from the map. BPF doesn't copy the values in a map before looping over them with `bpf_map_get_next_key`. If another part of your program deletes an element from the map while you're looping over the values, `bpf_map_get_next_key` will start over when it tries to find the next value for the element's key that it was removed. Let's see this with an example:

```
1 int next_key, lookup_key;
2 lookup_key = -1;
3
4 while(bpf_map_get_next_key(map_data[0].fd, &lookup_key, &next_key) == 0) {
5   printf("The next key in the map is: '%d'\n", next_key);
6   if (next_key == 2) {
7     printf("Deleting key '2'\n");
8     bpf_map_delete_element(map_data[0].fd &next_key);
9   }
10   lookup_key = next_key;
11 }
```

This program prints the next output:

```
The next key in the map is: '1'
The next key in the map is: '2'
Deleteing key '2'
The next key in the map is: '1'
The next key in the map is: '3'
The next key in the map is: '4'
The next key in the map is: '5'
```

This behavior is not very intuitive, so keep it in mind when you use `bpf_map_get_next_key`.

Specialized map helpers

Although the helpers that we've seen until now can cover most of the use cases to work with maps, the kernel offers several other helpers functions to work with more specialized maps. The most interesting of those helpers is `bpf_map_lookup_and_delete_elem`. This function search for a given key in the map and deletes the element from it. At the same time, it writes the value of the element in a variable for your program to use. This function will come handy when you use queue and stack maps, that we'll describe in the next section. However, it's not restricted to be used only with those types of maps. Let's see an example of how to use it with the map we've been using in our previous examples:

```
1 int key, value, result, it;
2 key = 1;
3
4 for (it = 0; it < 2; it++) {
5   result = bpf_map_lookup_and_delete_element(map_data[0].fd, &key, &value);
6   if (result == 0)
7     printf("Value read from the map: '%d'\n", value);
8   else
9     printf("Failed to read value from the map: %d (%s)\n", result,
strerror(errno));
10 }
```

In this example, we try to fetch the same element from the map twice. In the first iteration, this code will print the value of the element in the map. However, since we're using `bpf_map_lookup_and_delete_element`, this first iteration will also delete the element from the map. The second time the loop tries to fetch the element, this code will fail and it will populate the `errno` variable with the not found error `ENOENT`.

In this section, we've seen the possible operations you can do with BPF

Maps, however, we've only worked with one type of map so far. BPF includes many more map types that you can use in different situations. We'll explain all types of maps that BPF defines and we'll show you specific examples on how to use them for different situations.

Types of BPF Maps

Linux documentation defines Maps as generic data structures where you can store different types of data. Odver the years, the kernel developers have added many specialized data structures that are more efficient in specific use cases. In this section, we're going to explore each type of map and how you can use them.

BPF_MAP_TYPE_HASH

Hash-table maps were the first generic map added to bpf. Their implementation and usage is very similar to other hash tables you might be familiar with. You can use keys and values of any size, the kernel takes care of allocating and freeing them for you as needed. When you use `bpf_map_update_elem` on a hash-table map, the kernel replaces the elements atomically.

Hash-table maps are optimized to be very fast at lookup, they are very useful to keep structured data that's read frequently. Let's see an example program that uses them to keep track of network IPs and their rate limits:

```
#define IPV4_FAMILY 1
struct ip_key {
  union {
    __u32 v4_addr;
    __u8 v6_addr[16];
  };
  __u8 family;
```

```
8 };
9
10 struct bpf_map_def SEC("maps") counters = {
11       .type        = BPF_MAP_TYPE_HASH,
12       .key_size    = sizeof(struct ip_key),
13       .value_size  = sizeof(uint64_t),
14       .max_entries = 100,
15       .map_flags   = BPF_F_NO_PREALLOC
16 };
```

In the code above, we've declared a structured key, we're going to use it to keep information about IP addresses. In line 9, we define the map that our program will use to keep track of rate limits. You can see that we're using the IP addresses as keys in this map. The values are going to be the number of times that our BPF program receives a network packet from a specific IP address.

Let's write small code snippet that updates those counters in the kernel:

```
1 uint64_t update_counter(uint32_t ipv4) {
2   uint64_t value;
3   struct ip_key key = {};
4   key.v4_addr = ip4;
5   key.family = IPV4_FAMILY;
6
7   bpf_map_lookup_elem(counters, &key, &value);
8   (*value) += 1;
9 }
```

This function takes an IP address extracted from a network packet and performs the map lookup with the compound key that we're declaring. In this case, we're assuming that we've previously initialized the counter with a zero value, otherwise, the `bpf_map_lookup_elem` call would return a negative number.

BPF_MAP_TYPE_ARRAY

Array maps were the second type of bpf map added to the kernel. When you initialize an array map, all its elements are pre-allocated in memory and set to their zero value. Since these maps are backed by a slice of elements, the keys are indexes in the array, and their size must be exactly four bytes.

A disadvantage of using array maps is that the elements in the map cannot be removed, you cannot make the array smaller than it is. If you try to use `map_delete_elem` on an array map, the call will fail and you'll get an error `EINVAL` as a result.

Array maps are commonly used to store information that can change in value, but it's usually fixed in behavior. People use them to store global variables with a predefined assignment rule. Since you cannot remove elements, you can assume that the element in a specific position always represent the same element.

Something else to keep in mind is that `map_update_elem` is not atomic, like we saw with hash-table maps. The same program can read different values from the same position at the same time if there is an update in process. If you're storing counters in an array map, you can use the kernel's built-in function `__sync_fetch_and_add` to perform atomic operations on the map's values.

BPF_MAP_TYPE_PROG_ARRAY

Program array maps were the first specialized map added to the kernel. You can use this type of maps to store references to BPF programs using their file descriptor identifiers. In conjunction with the helper

`bpf_tail_call`, this map allows you to jump between programs, bypassing the maximum instruction limit of single BPF programs and reducing implementation complexity.

There are a few things you need to consider when you use this very specialized map. The first aspect to remember is that both key and value sizes must be four bytes. The second aspect to remember is that when you jump to a new program, the new program will reuse the same memory stack, so your program doesn't consume all the memory available. Finally, if you try to jump to a program that doesn't exist in the map, the tail call will fail, and the current program will continue its execution.

Let's dive into a detailed example to understand how to use this type of map better:

```
struct bpf_map_def SEC("maps") programs = {
  .type = BPF_MAP_TYPE_PROG_ARRAY,
  .key_size = 4,
  .value_size = 4,
  .max_entries = 1024,
};
```

First, we need to declare our new program map. As we mentioned before, key and value sizes are always four bytes.

```
int key = 1;
struct bpf_insn prog[] = {
  BPF_MOV64_IMM(BPF_REG_0, 0), // assign r0 = 0
  BPF_EXIT_INSN(),  // return r0
};

prog_fd = bpf_prog_load(BPF_PROG_TYPE_KPROBE, prog, sizeof(prog), "GPL");
bpf_map_update_elem(&programs, &key, &prog_fd, BPF_ANY);
```

We need to declare the program that we're going to jump to. In this case, we're writing a BPF program in place that its only purpose is returning 0. We use `bpf_prog_load` to load it in the kernel, and then, we add its file descriptor identifier to our program map.

Now that we have that program stored, we can write another BPF program that will jump to it. BPF programs can only jump to other programs of the same type, in this case, we're attaching the program to a kprobe trace, like we saw in Chapter 2:

```
1 SEC("kprobe/seccomp_phase1")
2 int bpf_kprobe_program(struct pt_regs *ctx) {
3   int key = 1;
4   /* dispatch into next BPF program */
5   bpf_tail_call(ctx, &programs, &key);
6
7   /* fall through when the program descriptor is not in the map */
8   char fmt[] = "missing program in prog_array map\n";
9   bpf_trace_printk(fmt, sizeof(fmt));
10   return 0;
11 }
```

You can chain up to 32 nested calls using `bpf_tail_call` and `BPF_MAP_TYPE_PROG_ARRAY`. This is an explicit limit to prevent infinite loops and memory exhaustion.

BPF_MAP_TYPE_PERF_EVENT_ARRAY

These type of maps store perf_events data in a buffer ring that communicates BPF programs with user-space programs in real time. They are designed to forward events that the kernel's tracing tools emit to user-space programs for further processing. This is one of the most interesting types of maps and are the base for many observability tools that we'll talk about in the next chapters. The user-space program acts as a listener that

waits for events coming from the kernel, so you need to make sure that your code starts listening before the BPF program in the kernel is initialized.

Let's see an example of how we can trace all the programs that our computer executes. Before jumping into the BPF program code, we need to declare the event structure that we're going to send from the kernel to user-space:

```
struct data_t {
  u32 pid;
  char program_name[16];
};
```

Now, we need to create the map that's going to send the events to user-space:

```
struct bpf_map_def SEC("maps") events = {
  .type = BPF_MAP_TYPE_PERF_EVENT_ARRAY,
  .key_size = sizeof(int),
  .value_size = sizeof(u32),
  .max_entries = 2,
};
```

Once we have our data type and map declared, we can create the BPF program that captures the data and sends it to user-space:

```
SEC("kprobe/sys_exec")
int bpf_capture_exec(struct pt_regs *ctx) {
  data_t data;
  // bpf_get_current_pid_tgid returns the current process identifier
  data.pid = bpf_get_current_pid_tgid() >> 32;
  // bpf_get_current_comm loads the current executable name
  bpf_get_current_comm(&data.program_name, sizeof(data.program_name));
  bpf_perf_event_output(ctx, &events, 0, &data, sizeof(data));
  return 0;
```

```
10 }
```

In line 8, we're using `bpf_perf_event_output` to append the data into the map. Since this is a real time buffer, you don't need worry about keys for the elements in the map, the kernel takes care of adding the new element to the map and flush it after the user-space program processes it.

In the chapter dedicated to tracing we'll talk about more advanced usages for these types of maps and we'll see examples of processing programs in user-space.

BPF_MAP_TYPE_PERCPU_HASH

This type of map is a refined version of BPF_MAP_TYPE_HASH. When you allocate one of these maps, each CPU sees its own isolated version of the map, which makes it much more efficient for high performant lookups and aggregations. This type of map is very useful if your BPF program collects metrics and aggregates them in hash-table maps.

BPF_MAP_TYPE_PERCPU_ARRAY

This type of map is also a refined version of **BPF_MAP_TYPE_ARRAY**. Just like the previous map, when you allocate one of these maps, each CPU sees its own isolated version of the map, which makes it much more efficient for high performant lookups and aggregations.

BPF_MAP_TYPE_STACK_TRACE

This type of map stores stack traces from the running process. Along this map, the kernel developers already added the helper `bpf_get_stackid` to help you populate this map with stack traces. This helper takes the map as an argument and a series of flags so you can specify whether you want

traces only from the kernel, only from user-space, or both. The helper returns the key associated with the element added to the map.

BPF_MAP_TYPE_CGROUP_ARRAY

This type of map stores references to Control Groups. In essence, its behavior is very similar to BPF_MAP_TYPE_PROG_ARRAY, but it stores file descriptor identifiers that point to cgroups.

This map is very useful when you want to share cgroup references between BPF maps for traffic control, debugging and testing. Let's see an example of how to populate this map.

We start with the map definition:

```
1 struct bpf_map_def SEC("maps") cgroups_map = {
2   .type = BPF_MAP_TYPE_CGROUP_ARRAY,
3   .key_size = sizeof(uint32_t),
4   .value_size = sizeof(uint32_t),
5   .max_entries = 1,
6 };
```

We can retrieve a cgroup's file descriptor by opening the file containing its information. We're going to open the cgroup that controls the base CPU shares for Docker containers, and we're going to store it in our map:

```
1 int cgroup_fd, key = 0;
2 cgroup_fd = open("/sys/fs/cgroup/cpu/docker/cpu.shares", O_RDONLY);
3
4 bpf_update_elem(&cgroups_map, &key, &cgroup_fd, 0);
```

BPF_MAP_TYPE_LRU_HASH and BPF_MAP_TYPE_LRU_PERCPU_HASH

These two types of map are hash-table maps, like the ones we saw earlier, but they also implement an internal LRU cache. LRU stands for Least Recently Used, which means that if the map is full these maps will erase elements that are not used frequently in order to make room for new elements in the map. Therefore, you can use these maps to insert elements beyond the maximum limit, as long as you don't mind loosing elements that have not been used recently.

The "`per cpu`" version of this map is slightly different than the other "`per cpu`" maps we saw earlier. This map keeps only one hash-table to store all the elements in the map, and it uses different LRU caches per cpu, that way, it ensures that the most used elements in each cpu remain in the map.

BPF_MAP_TYPE_LPM_TRIE

LMP trie maps are types of map that use Longest Prefix Match(LPM) to lookup elements in the map. Longest Prefix Match is an algorithm that selects an element in a tree which match with a lookup key is the longest from any other match in the tree. This algorithm is used in routers and other devices that keep traffic forwarding tables to match IP addresses with specific routes.

This kind of map require their key sizes to be multiple of 8, and in a range from 8 to 2048. The kernel provides a struct that you can use for these keys called `bpf_lpm_trie_key` if you don't to implement your own key.

In this example, we're going to add two forwarding routes to the map, and then we'll try to match an IP address to the correct route. First we need to create the map:

```
1 struct bpf_map_def SEC("maps") routing_map = {
```

```
  .type = BPF_MAP_TYPE_LPM_TRIE,
  .key_size = 8,
  .value_size = sizeof(uint64_t),
  .max_entries = 10000,
  .map_flags = BPF_F_NO_PREALLOC,
};
```

We're going to populate this map with three forwarding routes, `192.168.0.0/16`, `192.168.0.0/24` and `192.168.1.0/24`:

```
uint64_t value_1 = 1;
struct bpf_lpm_trie_key route_1 = {.data = {192, 168, 0, 0}, .prefixlen = 16};
uint64_t value_2 = 2;
struct bpf_lpm_trie_key route_2 = {.data = {192, 168, 0, 0}, .prefixlen = 24};
uint64_t value_3 = 3;
struct bpf_lpm_trie_key route_3 = {.data = {192, 168, 1, 0}, .prefixlen = 24};

bpf_map_update_elem(&routing_map, &route_1, &value_1, BPF_ANY);
bpf_map_update_elem(&routing_map, &route_2, &value_2, BPF_ANY);
bpf_map_update_elem(&routing_map, &route_3, &value_3, BPF_ANY);
```

Now, We'll use the same key structure to lookup the correct match for the IP `192.168.1.1/32`:

```
uint64_t result;
struct bpf_lpm_trie_key lookup = {.data = {192, 168, 1, 1}, .prefixlen = 32};

int ret = bpf_map_lookup_elem(&routing_map, &lookup, &result);
if (ret == 0)
  printf("Value read from the map: '%d'\n", result);
```

In this example, both `192.168.0.0/24` and `192.168.1.0/24` could match the lookup IP because it's within both ranges. However, since this map uses the LMP algorithm, the result will be populated with the value for the

key `192.168.1.0/24`.

BPF_MAP_TYPE_ARRAY_OF_MAPS and BPF_MAP_TYPE_HASH_OF_MAPS

BPF_MAP_TYPE_ARRAY_OF_MAPS and BPF_MAP_TYPE_HASH_OF_MAPS are two type of maps that store references to other maps. They only support one level of indirection, so you cannot use them to store maps of maps of maps, and so on. This ensures that you don't consume all the memory by accidentally storing infinite chained maps.

These type of maps are useful when you want to be able to replace entire maps at runtime. You can create full state snapshots if all your maps are children of a global map. The kernel ensures that any update operation in the parent map waits until all the references to old children maps are dropped before completing the operation.

BPF_MAP_TYPE_DEVMAP

This specialized type of map stores references to network devices. It's useful for network applications that want to manipulate traffic at the kernel level. You can build a virtual map of ports that point to specific network devices and then redirect packets by using the helper `bpf_redirect_map`.

BPF_MAP_TYPE_CPUMAP

BPF_MAP_TYPE_CPUMAP is another type of map that allows you to forward network traffic. In this case, the map stores references a different CPUs in your host. Like the previous map, you can use this map with the `bpf_redirect_map` helper to redirect packets. However, this map sends

packets to a different CPU. This allows you to assign specific CPUs to network stacks for scalability and isolation purposes.

BPF_MAP_TYPE_XSKMAP

BPF_MAP_TYPE_XSKMAP is a type of map that stores references to open sockets. Like the previous maps that we just talked about, these maps are useful for forwarding packets, between sockets in this case.

BPF_MAP_TYPE_SOCKMAP and BPF_MAP_TYPE_SOCKHASH

BPF_MAP_TYPE_SOCKMAP and BPF_MAP_TYPE_SOCKHASH are two specialized maps that store references to open sockets in the kernel. Like the previous maps we saw, this type of maps are used in conjuntion with the helper `bpf_redirect_map` to forward socket buffers from the current XDP program to a different socket.

Their main difference is that one of them uses an array to store the sockets and the other one uses a hash-table. The advantage of using a hash-table is that you can access a socket directly by its key without having to traverse the full map to find it. Each socket in the kernel is identified by a 5-tuple key. These 5 tuples include the necessary information to establish bidirectional network connections. You can use this key as the lookup key in your map when you use the hash-table version of this map.

BPF_MAP_TYPE_CGROUP_STORAGE and BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE

These two types of maps were introduced to help developers work with BPF programs attached to Control Groups. As we saw in the previous chapter about BPF program types, by using

BPF_PROG_TYPE_CGROUP_SKB, you can attach and detach BPF programs from control groups, and isolate their runtime to the specific cgroups.

These types of maps are very similar to hash-table maps from the developer point of view. The kernel provides a structure helper to generate keys for this map, `bpf_cgroup_storage_key`, which includes information about the cgroup inode identifier and the attachment type. You can add any value you want to this map, it's access will be restricted to the BPF program inside the attached cgroup.

There are two limitations with these maps. The first one is that you cannot create new elements in the map from user-space. The BPF program in the kernel can create elements with `bpf_map_update_elem`, but if you use this method from user-space and the key doesn't exist already, `bpf_map_update_elem` will fail and errno will be set to `ENOENT`. The second limitation is that you cannot remove elements from this map. `bpf_map_delete_elem` always fails and sets errno to `EINVAL`.

The main difference between these two types of maps, as we saw with other similar maps earlier, is that BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE keeps a different hash-table per CPU.

BPF_MAP_TYPE_REUSEPORT_SOCKARRAY

This specialized type of map stores references to sockets that can be reused by an open port in the system. These maps are mainly used with BPF_PROG_TYPE_SK_REUSEPORT program types. Combined they give you control to decide how to filter and serve incoming packets from the network device. For example, you can decide which packets go to

which socket, even though both sockets are attached to the same port.

BPF_MAP_TYPE_QUEUE

The Queue map uses a First In First Out(FIFO) storage to keep the elements in the map. FIFO means that when you fetch an element from the map, the result is going to be the element that has been in the map for a longer time.

The bpf map helpers work in a predictable way for this data structure. When you use `bpf_map_lookup_elem`, this map will always look for the oldest element in the map. When you use `bpf_map_update_elem`, this map will always append the element to the end of the queue, so you'll need to read the rest elements in the map before being able to fetch this element. You can also use the helper `bpf_map_lookup_and_delete` to fetch the older element and remove it from the map in an atomic way. This map doesn't support the helpers `bpf_map_delete_elem` and `bpf_map_get_next_key`. If you try to use them, they will always fail and set the errno variable to `EINVAL` as a result.

Something else that you need to keep in mind about this map is that they don't use the map keys for lookups, and the key size must always be 0 when you initialize these maps. When you push elements to these maps, the key must be a null value.

Let see an example of how to use this map:

```
1  struct bpf_map_def SEC("maps") queue_map = {
2    .type = BPF_MAP_TYPE_QUEUE,
3    .key_size = 0,
4    .value_size = sizeof(int),
5    .max_entries = 100,
```

```
    .map_flags = 0,
};
```

Let’s insert several elements in this map and retrieve them in the same order that we inserted them:

```
int i;
for (i = 0; i < 5; i++)
  bpf_map_update_elem(&queue_map, NULL, &i, BPF_ANY);

int value;
for (i = 0; i < 5; i++) {
  bpf_map_lookup_and_delete(&queue_map, NULL, &value);
  printf("Value read from the map: '%d'\n", value);
}
```

This program will print:

```
Value read from the map: '0'
Value read from the map: '1'
Value read from the map: '2'
Value read from the map: '3'
Value read from the map: '4'
```

If we try to pop a new element from the map, `bpf_map_lookup_and_delete` will return a negative number and the errno variable will be set to `ENOENT`.

BPF_MAP_TYPE_STACK

The Stack map uses a Last In First Out(LIFO) storage to keep the elements in the map. LIFO means that when you fetch an element from the map, the result is going to be the element that was added to the map most recently.

The bpf map helpers also work in a predictable way for this data structure.

When you use `bpf_map_lookup_elem`, this map will alway look for the newest element in the map. When you use `bpf_map_update_elem`, this map will always append the element to the top of the stack, so it's the first one to fetch. You can also use the helper `bpf_map_lookup_and_delete` to fetch the newest element and remove it from the map in an atomic way. This map doesn't support the helpers `bpf_map_delete_elem` and `bpf_map_get_next_key`. If you try to use them, they will always fail and set the errno variable to `EINVAL` as a result.

Let see an example of how to use this map:

```
struct bpf_map_def SEC("maps") stack_map = {
  .type = BPF_MAP_TYPE_STACK,
  .key_size = 0,
  .value_size = sizeof(int),
  .max_entries = 100,
  .map_flags = 0,
};
```

Let's insert several elements in this map and retrieve them in the same order that we inserted them:

```
int i;
for (i = 0; i < 5; i++)
  bpf_map_update_elem(&stack_map, NULL, &i, BPF_ANY);

int value;
for (i = 0; i < 5; i++) {
  bpf_map_lookup_and_delete(&stack_map, NULL, &value);
  printf("Value read from the map: '%d'\n", value);
}
```

This program will print:

```
Value read from the map: '4'
Value read from the map: '3'
```

```
Value read from the map: '2'
Value read from the map: '1'
Value read from the map: '0'
```

If we try to pop a new element from the map, `bpf_map_lookup_and_delete` will return a negative number and the errno variable will be set to `ENOENT`.

These are all the map types that you can use in a BPF program. You'll find some of them more useful than others, it will depend on the kind of program that you're writing. We'll see more usage examples over the book that will help you cement the fundamentals that we just saw.

Concurrent access to map elements

One of the challenges of working with BPF maps is that many programs can access the same maps concurrently. This can introduce race conditions in our BPF programs, and make accessing resources in map unpredictable. To prevent race conditions, BPF introduced the concept of BPF Spin Locks, which allow you to lock access to a map's element while you're operating on it. Spin Locks work only on Array, Hash, and Cgroup storage maps.

There are two BPF helper functions to work with spin locks, one to lock at element, `bpf_spin_lock`, and one to unlock that same element `bpf_spin_unlock`. Those helper work with a structure that acts as a semaphone to access an element that includes this semaphore. When the semaphore is locked, other programs cannot access the element's value, and they wait until the semaphore is unlocked. At the same time, BPF Spin Lock introduces a new that user-space programs can use to change the state of that lock, that flag is called `BPF_F_LOCK`.

The first thing we need to do to work with Spin Locks is to create the element that we want to lock access to, and add our semaphore:

```
struct concurrent_element {
  struct bpf_spin_lock semaphore;
  int count;
}
```

We'll store this structure in our BPF map, and we'll use the semaphore within the element to prevent undesired access to it. Now, we can declare the map that's going to hold these elements. This map must be annotated with BPF Type Format(BTF), so the verifier knows how to intepret the structure. We talked about BTF in the previous chapter, but as a refresher, the type format give the kernel and other tools a much richer understanding of BPF data structures. Since this code will run inside the kernel, we can use the kernel macros that libbpf provides to annotate this concurrent map.

```
struct bpf_map_def SEC("maps") concurrent_map = {
      .type        = BPF_MAP_TYPE_HASH,
      .key_size    = sizeof(int),
      .value_size  = sizeof(struct concurrent_element),
      .max_entries = 100,
};

BPF_ANNOTATE_KV_PAIR(concurrent_map, int, struct concurrent_element);
```

Whithin a BPF program with can use the two locking helpers to protect from race conditions on those elements. While the semaphore is locked, our program is guaranteed to be able to modify the element's value safely:

```
int bpf_program(struct pt_regs *ctx) {
  int key = 0;
  struct concurrent_element init_value = {};
  struct concurrent_element *read_value;
```

```
5
6   bpf_map_create_elem(&concurrent_map, &key, &init_value, BPF_NOEXIST);
7
8   read_value = bpf_map_lookup_elem(&concurrent_map, &key);
9   bpf_spin_lock(&read_value->semaphore);
10   read_value->count += 100;
11   bpf_spin_unlock(&read_value->semaphore);
12 }
```

This example will initialize our concurrent map with a new entry that can lock accessing its value. Then, it will fetch that value from the map and lock its semaphore so it can hold the count value preventing data races. When it's done using the value, it will release the lock so other maps can access the element safely.

From user-space, we can hold the reference to an element in our concurrent map by using the flag `BPF_F_LOCK`. You can use this flag with the helper functions `bpf_map_update_elem` and `bpf_map_lookup_elem_flags`. This flag allows you to update elements in place without having to worry about data races.

> **NOTE**
>
> BPF_F_LOCK has a slightly different behavior updating hash maps, and updating array and cgroups storage maps. With the two later, the updates happen in place, the elements that you're updating must exist in the map before executing the update. In the case of hash maps, if the element doesn't exist already, the program locks the bucket in the map for the element, and inserts a new element.

Spin Locks are not always necessary. You don't need them if you're only aggregating values in a map. However, they are very useful if you want to ensure that concurrent programs don't change elements in a map when you're performing several operations on them, preserving atomicity.

The BPF Virtual File System

A fundamental characteristic of BPF Maps is that being based on file descriptors means that when a descriptor is closed, the map and all the information it holds disappear. The original implementation of BPF Maps was focused on short lived, and isolated, programs, that didn't share any information between each other. Erasing all data when the file descriptor was closed made a lot of sense in those scenarios. However, with the introduction of more complex maps and integrations within the kernel, its developers realized that they needed a way to persist the information that maps held, even after a program terminated and it closed the map's file descriptor. The version 4.4 of the Linux kernel introduced two new system calls to allow pinning and fetching maps and bpf programs from a virtual file system. Maps and BPF programs pinned to this file system will remain in memory after the program that created them terminates. In this section, we're going to explain how to work with this virtual file system.

The default directory where BPF expects to find this virtual file system is `/sys/fs/bpf`. It's usually not mounted by default, you can mount it yourself by using the mount command:

```
sudo mount -t bpf /sys/fs/bpf /sys/fs/bpf
```

BPF's virtual file system is a singleton file system. This means that if you mount it in several places within the same namespace, all mounts will see the same content. On the other hand, when you mount this file system in different mount namespaces, you'll get a different version for each namespace. We'll talk more about namespaces and BPF in the chapter about containers, we're going to focus on how to work with this file system for now.

Like any other file hierarchy, persistent BPF objects in the file system are identified by paths. You can organize these paths in any way that makes sense for your programs. For example, if you want to share a specific map with IPs information between programs, you might want to store it in `/sys/fs/bpf/shared/ips`. As we mentioned earlier, there are two types of objects that you can persist in this file system, BPF maps and full BPF programs, both of them are identified by file descriptors, so the interface to work with them is the same. These objects can only be manipulated by the `bpf` system call. Although the kernel provides high level helpers to help you interact with them, you cannot do things like trying to open these files with the `open` system call.

`BPF_PIN_FD` is the command to save BPF objects in this file system. When the command succeeds, the object will be visible in the file system in the path that you specified. If the command fails, it returns a negative number, and the global `errno` variable is set with the error code.

`BPF_OBJ_GET` is the command to fetch BPF objects that have been pinned to the file system. This command uses the path you assigned the object to to load it. When this command succeeds, it returns the file descriptor identifier associated to the object. If it fails, it returns a negative number, and the global `errno` variable is set with the specific error code.

Let's see an example of how to take advantage of these two commands in different programs using the helper functions that the kernel provides:

First, we're going to write a program that creates a map, populates it with several elements and saves it in the files system:

```
1 static const char * file_path = "/sys/fs/bpf/my_map";
2
```

```
3 int main(int argc, char **argv) {
4   int key, value, fd, added, pinned;
5
6   fd = bpf_create_map(BPF_MAP_TYPE_HASH, sizeof(int), sizeof(int), 100,
BPF_F_NO_PREALLOC);
7   if (fd < 0) {
8     printf("Failed to create map: %d (%s)\n", fd, strerror(errno));
9     return -1;
10  }
11
12  key = 1, value = 1234;
13  added = bpf_map_update_elem(fd, &key, &value, BPF_ANY);
14  if (added < 0) {
15    printf("Failed to update map: %d (%s)\n", added, strerror(errno));
16    return -1;
17  }
18
19  pinned = bpf_obj_pin(fd, file_path);
20  if (pinned < 0) {
21    printf("Failed to pin map to the file system: %d (%s)\n", pinned,
strerror(errno));
22    return -1;
23  }
24
25  return 0;
26 }
```

The lines from 6 to 17 should already look familiar to you from our previous examples. First, we create an hash-table map with a fixed size of 1 element. Then we update the map to add that only element. If we tried to add more elements, `bpf_map_update_elem` would fails because we were overflowing the map.

In line 19, we pin the map to the file system. You can actually check that you have a new file under that path in your machine after the program has terminated:

```
ls -la /sys/fs/bpf
```

```
total 0
drwxrwxrwt 2 root  root  0 Nov 24 13:56 .
drwxr-xr-x 9 root  root  0 Nov 24 09:29 ..
-rw------- 1 david david 0 Nov 24 13:56 my_map
```

Now we can write a similar program that loads that map from the file system and prints the elements we inserted, that way we can verify that we persisted the map correctly:

```
1 static const char * file_path = "/sys/fs/bpf/my_map";
2
3 int main(int argc, char **argv) {
4   int fd, key, value, result;
5
6   fd = bpf_obj_get(file_path);
7   if (fd < 0) {
8     printf("Failed to fetch the map: %d (%s)\n", fd, strerror(errno));
9     return -1;
10   }
11
12   key = 1;
13   result = bpf_map_lookup_elem(fd, &key, &value);
14   if (result < 0) {
15     printf("Failed to read value from the map: %d (%s)\n", result,
strerror(errno));
16     return -1;
17   }
18
19   printf("Value read from the map: '%d'\n", value);
20   return 0;
21 }
```

Being able to persist BPF objects in the file system opens the door to more interesting applications. Your data and programs are not tied to a single execution thread anymore. Information can be shared by different applications, and BPF programs can run even after the application that created them terminates. This gives them an extra level or availability that could have not been possible without the BPF file system.

Conclusion

Establishing communication channels between the kernel and user-space is fundamental to take full advantage of any BPF program. In this chapter, we've seen how to create BPF Maps to establish that communication and how to work with them. We've also described the types of maps that you can use in your programs. As you progress in the book, you'll see more specific map examples. Finally, we've learned how to pin entire maps to the system to make them and the information the hold durable to crashes and interruptions.

Chapter 3. Tracing with BPF

The objective of tracing is to provide useful data at runtime for future analysis. The main advantage of using BPF for tracing is that you can access almost any piece of information from the Linux Kernel and your applications. BPF adds a minimum overhead to the systems performance and latency in comparison with other tracing technologies, and it doesn't require developers to modify their applications for the only purpose of gathering data from them.

The Linux Kernel provides several instrumentation capabilities that can be used in conjunction with BPF. In this chapter, we're going to talk about those different capabilities. We're also going show you how the kernel exposes those capabilities in your operating system, so you know how to find the information available to your BPF programs.

Tracing's end goal it to provide you with a deep understanding of any system by taking all the available data and present it to you in an useful way. In this chapter, we're also going to go through a few different data representations, and how you can use in different scenarios. At the end of the chapter, you'll be equipped with very powerful techniques and tools to debug any Linux system.

Probes

One of the definitions in the English dictionary for the word "`Probe`" reads:

> *An unmanned exploratory spacecraft designed to transmit information about its environment.*

This definition evokes memories of Sci-Fi movies and epic NASA missions in our minds, and probably in yours too. When we talk about tracing probes, we can use a very similar definition.

> *Tracing probes are exploratory programs designed to transmit information about the environment in which they are executed.*

They collect data in your system, and make it available for you to explore and analyze. Traditionally, using probes in Linux involved writing programs that got compiled into kernel modules, which could cause catastrophic problem in production systems. Over the years, they evolved to be safer to execute, but still cumbersome to write and test. Tools like SystemTap established new protocols to write probes, and paved the way to get much richer information from the Linux Kernel and all programs running on user-space.

BPF piggybacks on tracing probes to collect information for debugging and analysis. The safety nature of BPF programs makes them more compelling to use that tools that still rely on re-compiling the kernel. The BPF verifier guarantees that this will never happen. The BPF developers took advantage of probes definitions, and modified the kernel to execute BPF programs rather than kernel modules when a code execution finds one of those definitions.

Understanding the different types of probes that you can define is fundamental to explore what’s happening within our system. In this section, we’re going to classify the different probe definitions, how to discover them in your system, and how to attach BPF programs to them.

Kernel probes

Kernel probes is a feature that allows you to set dynamic flags, or breaks, in almost any kernel instruction with a minimum overhead. When the kernel reaches one of these flags, it executes the code attached to the probe, and then resumes its usual routine. Kernel probes can give you information about anything happening in your system, such as files opened in your system and binaries being executed. One important thing to keep in mind about Kernel probes is that they don’t have a stable Application Binary Interface, or ABI, which means that they might change between kernel versions. The same code might stop working if you try to attach the same probe to two systems with two different kernel versions.

Kernel probes are divided in two categories, depending on where in the execution cycle you can insert your BPF program. This section is going to guide you on how to use each one of them to attach BPF programs to those probes and extract information from the kernel.

KPROBES

Kprobes are going to allow you to insert BPF programs before any kernel instruction is executed. You need to know the function signature that you want to break into, and as we mentioned earlier, this is not a stable ABI, so you’ll want to be careful setting these probes if you’re going to run the same program in different kernel versions. When the kernel execution arrives to the instruction where you set your probe, it side steps into your code, runs your BPF program, and returns the execution to the original instruction.

To show you how to use Kprobes, we’re going to write a BPF program that prints the name of any binary that’s executed in your system. We’re

going to use the Python frontend for the BCC tools in this example, but you can write it with any other BPF tooling:

```
1 from bcc import BPF
2
3 bpf_source = """
4 int do_sys_execve(struct pt_regs *ctx, void filename, void argv, void envp)
{ ❶
5   char comm[16];
6   bpf_get_current_comm(&comm, sizeof(comm));
7   bpf_trace_printk("executing program: %s", comm);
8   return 0;
9 }
10 """
11
12 bpf = BPF(text = bpf_source) ❷
13 execve_function = bpf.get_syscall_fnname("execve") ❸
14 bpf.attach_kprobe(event = execve_function, fn_name = "do_sys_execve")
15 bpf.trace_print()
```

❶ This function implements our BPF program. The helper `bpf_get_current_comm` is going to fetch the current command's name that the kernel is running and store it in our `comm` variable. We've defined this as a fixed length array because the kernel has a 16 character limit for command names. After getting the command name, we print it in our debug trace, so the person running the Python script can see all commands captured by BPF.

❷ Here, we load the BPF program into the Kernel.

❸ Next, we associate the program with the `execve` system call. The name of this system call has changed in different kernel versions, and BCC provides a function to retrieve this name without you having to remember which kernel version you're running.

Finally, line 15 outputs the trace log, so we can see all the commands that we're tracing with this program.

KRETPROBES

Kretprobes are going to insert your BPF program when a kernel

instruction returns a value after being executed. Usually, you'll want to combine both, kprobes, and kretprobes into a single BPF program, so you have a full picture of the instruction's behavior.

We're going to use a very similar example than the one in the previous section to show you how kretprobes work:

```
1 from bcc import BPF
2
3 bpf_source = """
4 int ret_sys_execve(struct pt_regs *ctx) { ❶
5   int return_value;
6   char comm[16];
7   bpf_get_current_comm(&comm, sizeof(comm));
8   return_value = PT_REGS_RC(ctx);
9
10   bpf_trace_printk("program: %s, return: %d", comm, return_value);
11   return 0;
12 }
13 """
14
15 bpf = BPF(text = bpf_source) ❷
16 execve_function = bpf.get_syscall_fnname("execve")
17 bpf.attach_kretprobe(event = execve_function, fn_name = "ret_sys_execve")
❸
18 bpf.trace_print()
```

❶ This function implements the BPF program that the kernel is going to execute when it returns from running an `execve` system call. `PT_REGS_RC` is a macro that's going to read the returned value from BPF register for this specific context. We also use `bpf_trace_printk` to print the command and it returned value in our debug log.

❷ Here, we initialize the BPF program and load it in the kernel.

❸ You can see in this line that we've changed the attachment function to `attach_kretprobe`.

What's that context argument?

You might have noticed that both BPF programs have the same first

argument in the attached function, identified as `ctx`. This structure is used by the CPU to store different information about the current task that the kernel is executing. This structure is dependent to your system architecture, an ARM processor will include a different set or registers than an x64 processor. You can access those registers without having to worry about the architecture with macros defined by the kernel, like `PT_REGS_RC`.

Tracepoints

Tracepoints are static markers in the kernel's code that can be used to attach code in a running kernel. The main difference with Kprobes is that they are codified by the kernel developers when they implement changes in the kernel, that's why we refer to them as static. Since they are static, the ABI for tracepoints is more stable, the kernel always guarantees that a tracepoint in an old version is going to exist in new versions. However, given that developers need to add them to the kernel, they might not cover all the subsystems that form the kernel.

As we mentioned in Chapter 2, you can see all the available tracepoints in your system by listing what's all files inside `/sys/kernel/debug/tracing/events`. For example, we can find all the tracepoints for BPF itself by listing the events defined in ``/sys/kernel/debug/tracing/events/bpf`:

```
sudo ls -la /sys/kernel/debug/tracing/events/bpf
total 0
drwxr-xr-x  14 root root 0 Feb  4 16:13 .
drwxr-xr-x 106 root root 0 Feb  4 16:14 ..
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_map_create
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_map_delete_elem
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_map_lookup_elem
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_map_next_key
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_map_update_elem
```

```
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_obj_get_map
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_obj_get_prog
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_obj_pin_map
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_obj_pin_prog
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_prog_get_type
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_prog_load
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_prog_put_rcu
-rw-r--r--   1 root root 0 Feb  4 16:13 enable
-rw-r--r--   1 root root 0 Feb  4 16:13 filter
```

Every subdirectory listed in that output corresponds with a tracepoints that we can attach BPF programs to. But there are two additional files there. The first file `enable`, allow us to enable and disable all tracepoints for the BPF subsystem. If the content of the file is 0, the tracepoints are disabled, if the content of the file is 1, the tracepoints are enabled. The `filter` file allows you to write expressions that the Trace subsystem in the kernel will use to filter events. BPF doesn't use this file, you can read more about these events in the kernel's Tracing documentation.

Writing BPF programs to take advantage of tracepoints is very similar to tracing with Kprobes. We're going to show you an example that uses a BPF program to trace all the applications in your system that load other BPF programs:

```
1 from bcc import BPF
2
3 bpf_source = """
4 int trace_bpf_prog_load(void ctx) { ❶
5   char comm[16];
6   bpf_get_current_comm(&comm, sizeof(comm));
7
8   bpf_trace_printk("%s is loading a BPF program", comm);
9   return 0;
10 }
11 """
12
13 bpf = BPF(text = bpf_source)
14 bpf.attach_tracepoint(tp = "bpf:bpf_prog_load", fn_name =
"trace_bpf_prog_load") ❷
```

```
15 bpf.trace_print()
```

1. This function defines our BPF program. This code must look familiar to you already, there are only a few syntactic changes from the first example you saw when we talked about Kprobes.
2. Here, we find the main difference. Instead of attaching the program to a Kprobe, we're attaching it to a tracepoint. BCC follows a convention to name tracepoints, first you specify the subsystem to trace, `bpf` in this case, followed by a colon, followed by the tracepoint in the subsystem, `pbf_prog_load`. This means that every time the kernel executes the function `bpf_prog_load`, this program will receive the event and it will print the name of the application that's executing that `bpf_prog_load` instruction.

User-space probes

User-space probes allow you to set dynamic flags in programs running in user-space. They are the equivalent of Kernel-probes for instrumenting programs that run outside the kernel. When you define a Uprobe, the kernel creates a trap around the attached instruction. When your application reaches that instruction, the kernel triggers an event that has your probe function as a callback. Uprobes also give you access to any library that your program is linked to, and you can trace those calls if you know the right name for the instruction.

Much like Kernel probes, User-space probes are also divided in two categories, depending on where in the execution cycle you can insert your BPF program. Let jump right in with some examples.

UPROBES

Generally speaking, we call Uprobes to traps inserted before executing a program instruction. You need to be careful when you attach Uprobes to

different versions of the same program because function signatures might change internally between those versions. The only way to guarantee that a BPF program is going to run in two different versions is to ensure that the signature has not changed. You can use the command `nm` in Linux to list all the symbols included in an Elf object file, this is a good way to check that the instruction that you're tracing still exists in your program. For example, given this Go program:

```
1 package main
2 import "fmt"
3
4 func main() {
5         fmt.Println("Hello, BPF")
6 }
```

You can compile it by using `go build -o hello-bpf main.go`. We can use the command `nm` too get information about all the instruction points that the binary file includes. Nm is a program included in the GNU Development Tools that lists symbols from object files. If we filter the symbols with `main` in their name, we get a list very similar to this one:

```
nm hello-bpf | grep main
0000000004850b0 T main.init
00000000567f06 B main.initdone.
00000000485040 T main.main
000000004c84a0 R main.statictmp_0
00000000428660 T runtime.main
0000000044da30 T runtime.main.func1
00000000044da80 T runtime.main.func2
000000000054b928 B runtime.main_init_done
00000000004c8180 R runtime.mainPC
0000000000567f1a B runtime.mainStarted
```

Now that we have a list of symbols, we can trace when they are executed, even between different processes executing the same binary.

In order to trace when the main function in our previous Go example is

executed, we're going to write a BPF program and we're going to attach it to an Uprobe that will break before any process invokes that instruction.

```
from bcc import BPF

bpf_source = """
int trace_go_main(struct pt_regs *ctx) {
  u64 pid = bpf_get_current_pid_tgid(); ❶
  bpf_trace_printk("New hello-bpf process running with PID: %d", pid);
}
"""

bpf = BPF(text = bpf_source)
bpf.attach_uprobe(name = "hello-bpf", sym = "main.main", fn_name =
"trace_go_main") ❷
bpf.trace_print()
```

```
<1> First, we're using the function  `bpf_get_current_pid_tgid` to get the
process identifier(PID) for the process that's running our hello-bpf program.
<2> Here, we're attaching this program to a Uprobe. This call needs to know
the object that we want to trace, `hello-bpf` is the absolute path to the
object file. It also needs the symbol that we're tracing inside the object,
`main.main` in this case, and the BPF program that we want to run. With this,
every time someone runs `hello-bpf` in our system, we'll get a new log in our
trace pipe.
```

URETPROBES

Uretprobes are the parallel probe to Kretprobes, but for user-space programs. They attach BPF programs to instructions that return values, and give you access to those returned values by accessing the registers from your BPF code.

Combining Uprobes and Uretprobes will allow you to write more complex BPF programs. They can give you a more holistic view of applications running in your system. When you can inject tracing code before a function runs, and right after it completes, you can start gathering more data and measure application behaviors. A common use case is to measure

how long a function takes to execute, without having to change a single line of code in your application.

We're going to reuse the Go program we wrote to show you how Uprobes work, and measure how long it takes to execute the main function. This example is longer than the previous examples that you've seen before, so we're going to divide it in different blocks of code.

```
1 from bcc import BPF
2
3 bpf_source = """
4 BPF_HASH(cache, u64, u64); ❶
5
6 static int trace_start_time(struct pt_regs *ctx) {
7   u64 pid = bpf_get_current_pid_tgid();
8   u64 start_time_ns = bpf_ktime_get_ns(); ❷
9   cache.update(&pid, &start_time_ns); ❸
10   return 0
11 }
12 """
```

❶ In line 4, we're creating a BPF Hashmap, this table will allow us to share data between the Uprove and Uretprobe functions. In this case, we're going to use the application PID as the table key, and we'll store the function start time as the value. The two most interesting operations in our Uprobe function happen in lines 8 and 9.

❷ In line 8, we're capturing the current time in the system in nanoseconds, as seen by the kernel.

❸ In line 9, we're creating an entry in our cache with the program PID and the current time. We can assume that this time is the application's function start time. Let's declare our Uretprobe function now:

Next, we're going to implement the function to attach when your instruction finishes. This Uretprove function is very similar to other's that you saw in the Kretprobes section.

```
1 bpf_source += """
2 static int print_duration(struct pt_regs *ctx) {
```

```
3   u64 pid = bpf_get_current_pid_tgid(); ❶
4   u64 start_time_ns = cache.lookup(&pid);
5   if (start_time_ns == 0) {
6     return 0;
7   }
8   u64 duration_ns = bpf_ktime_get_ns() - start_time_ns; ❷
9   bpf_trace_printk("Function call duration: %d", duration_ns); ❸
10   return 0;
11 }
```

❶ In line 3, we obtain the PID for our application, we need it to find its starting time. We use the map function `lookup` to fetch that time from the map where we stored it before the function ran.

❷ Once we have the starting time, we can calculate the function duration by subtracting that time from the current time, you can see that code in this line.

❸ In line 9, we print the latency in our trace log, so we can display it in the terminal.

Now, the rest of the program needs to attach these two BPF functions to the right probes:

```
1 bpf = BPF(text = bpf_source)
2 bpf.attach_uprobe(name = "hello-bpf", sym = "main.main", fn_name =
"trace_start_time")
3 bpf.attach_uretprobe(name = "hello-bpf", sym = "main.main", fn_name =
"print_duration") ❶
4 bpf.trace_print()
```

❶ We've added line 3 to our original Uprobe example. There we're attaching our print function to the Uretprove for our application.

In this section we've seen how to trace operations that happen in user-space with BPF. By combining BPF functions that get executed at different points in your application's lifecycle, we can start extracting much richer information from it. However, as we mentioned at the beginning of this section, user-space probes are very powerful, but they are also very unstable. Our BPF examples can stop working only because

someone decides to rename an application's function. Next, we're going to see a more stable way to trace user-space programs.

User Statically Defined Tracepoints

User Statically Defined Tracepoints(USDT) provide static tracepoints for applications in user-space. This is a convenient way to instrument applications because they give you a low-overhead entrypoint to the tracing capabilities that BPF offers. You can also use them as a convention to trace applications in production, regardless of the programming language that these applications are written with.

USDTs were popularized by DTrace, a tool originally developed at Sun Microsystems for dynamic instrumentation of unix systems. DTrace was not available in Linux until recently due to licensing issues, however the Linux kernel developers took a lot of inspiration from the original work in DTrace to implement USDTs.

Much like the static kernel tracepoints we saw earlier, USDTs require developers to instrument their code with instructions that the kernel will use as traps to execute BPF programs. The Hello World of USDTs is only a few lines of code:

```
1  #include <sys/sdt.h>
2  int main() {
3    DTRACE_PROBE("hello-usdt", "probe-main");
4  }
```

In this example, we're using a macro that Linux provides to define our first USDT. You can already see where the kernel takes inspiration from. `DTRACE_PROBE` is going to register the tracepoint that the kernel will use to inject our BPF function callback. The first argument in this macro is the

program that's reporting the trace. The second one is the name of the trace that we're reporting.

Many applications that you might have installed in your system use this type of probes to give you access to runtime tracing data in a predictable way. The popular database MySQL, for example, exposes all kinds of information using statically defined tracepoints. You can gather information from queries executed in the server as well as from many other user operations. Node.js, the javascript runtime built on top of Chrome's V8 engine, also provides tracepoints that you can use to know extract runtime information.

Before showing you how to attach BPF programs to user defined tracepoint we need to talk about discoverability. Since these tracepoints are defined in binary format inside the executable files, we need a way to list the probes defined by a program without having to dig through the source code. One way to extra this information is by reading the ELF binary directly. First, we're going to compile our previous Hello USDT example, we can use GCC for that:

```
gcc -o hello_usdt hello_usdt.c
```

This command is going to generate a binary file called `hello_usdt` that we can use to start playing with several tools to discover the tracepoints that it defines. Linux provides a utility called `readelf` to show you information about ELF files. You can use it with our compiled example:

```
readelf -n ./hello_usdt
```

You can see the USDT that we defined in the output of this command:

```
Displaying notes found in: .note.stapsdt
```

```
  Owner                 Data size        Description
  stapsdt               0x00000033       NT_STAPSDT (SystemTap probe
descriptors)
    Provider: "hello-usdt"
    Name: "probe-main"
    Location: 0x00000000000005fe, Base: 0x0000000000000694, Semaphore:
0x0000000000000000
    Arguments:
```

`readelf` can give you a lot of information about a binary file, in our small example, it only shows a few lines of information, but its output becomes very cumbersome to parse for more complicated binaries.

A better option to discover the tracepoints defined in a binary file is to use BCC's tplist tool. Tplist is a tool that can display both, Kernel tracepoints and user statically defined tracepoints. The advantage of this tool is the simplicity of its output, it only shows you tracepoint definitions, without any additional information about the executable. It's usage is very similar to `readelf`:

```
tplist -l ./hello_usdt
```

It lists every tracepoint that you define in individual lines. In our example, it only displays a single line with our probe-main definition:

```
./hello_usdt "hello-usdt":"probe-main"
```

Once you know the supported tracepoints in your binary, you can attach BPF programs to them in a very similar way than we've seen in previous examples.

```
1 from bcc import BPF, USDT
2
3 bpf_source = """
4 #include <uapi/linux/ptrace.h>
5 int trace_binary_exec(struct pt_regs *ctx) {
6   u64 pid = bpf_get_current_pid_tgid();
```

```
7   bpf_trace_printk("New hello_usdt process running with PID: %d", pid);
8 }
9 """
10
11 usdt = USDT(path = "./hello_usdt") ❶
12 usdt.enable_probe(probe = "probe-main", fn_name = "trace_binary_exec") ❷
13 bpf = BPF(text = bpf_source, usdt = usdt)
14 bpf.trace_print()
```

❶ There is a major change in this example that requires some explanation. In line 11 we're creating an USDT object, we've not done this in our previous examples. USDTs are not part of BPF, in the sense that you can use them without having to interact with the BPF virtual machine. Since they are independent of each other, it makes sense that their usage is independent of the BPF code.

❷ In line 12, we're attaching the BPF function to trace program executions to the probe in our application. In line 13, we're initializating our BPF environment with the tracepoint definition that we just created. This will inform BCC that it needs to generate the code to connect our BPF program with the probe definition in our binary file. Once both of them are connected, we can print the traces generated by our BPF program to discover new executions of our binary example.

In this section, we've shown you how to introspect applications that define tracepoints statically. Many of well-known libraries and programing languages include those probes to help you debug running application, gaining more visibility when they run in production environments. This is only the tip of the iceberg, once you have the data, you need to make sense of it. This is what we're going to explore next.

Visualizing tracing data

So far, we've only talked about examples that print data in our debug output. This is not very useful in production environments. You want to

make sense of that data, and nobody likes to make sense of long and complicated logs. If we want to monitor changes in latency and CPU utilization, it's easier to do it by looking and graphs over a time period than aggregating numbers from a file stream.

In this section, we're going to explore different ways to present BPF tracing data. On one hand, we'll show you how BPF programs can structure information in aggregates for you. On the other hand, you'll also learn how to export that information in a portable representation and use off the shelf tools to access a more rich representation and share your findings with other people.

Histograms

Histograms are diagrams that show you how frequent several ranges of values occur. The numeric data to represent is divided into buckets, and each bucket contains the number of occurrences of any datapoint within the bucket. The frequency that histograms measure is the combination of height and width of each bucket. If the buckets are divided in equal ranges, this frequency matches the histogram's height, but if the ranges are not divided equally, you need to multiply each height by each width to find the correct frequency.

Histograms are a fundamental component to do systems performance analysis. They are great tool to represent distribution of measurable events, like instruction latency, because they show you a more correct information than you can get with other measurements, like averages.

BPF programs can create histograms based on many metrics. You can use BPF maps to collect the information, classify it in buckets, and then, generate the histogram representation for your data. Implementing this

logic is not very complicated, but it becomes tedious if you want to print histograms every time you need to analyze a program's output. BCC includes an implementation out of the box that you can reuse in every program, without having to calculate bucketing and frequency manually every single time. However, the Kernel source has a fantastic implementation that we encourage you to check out in the BPF samples.

As a fun experiment, we're going to show you how to use BCC's histograms to visualize the latency introduced by loading BPF programs when an application calls the `bpf_prog_load` instruction. We're going to use Kprobes to collect how long it takes to that instruction to complete, and we'll accumulate the results in a histogram that we'll visualize later. We're going to divide this example in several parts to talk to make it easier to follow.

This first part includes the initial source for our BPF program:

```
1 from bcc import BPF
2
3 bpf_source = """
4 #include <uapi/linux/ptrace.h>
5
6 BPF_HASH(cache, u64, u64); ❶
7 BPF_HISTOGRAM(histogram); ❷
8
9 int trace_bpf_prog_load_start(void ctx) {
10   u64 pid = bpf_get_current_pid_tgid();
11   u64 start_time_ns = bpf_ktime_get_ns();
12   cache.update(&pid, &start_time_ns);
13   return 0;
14 }
15 """
```

❶ In line 6, we're using a macro to create a BPF Hash map to store the initial time when the `bpf_prog_load` instruction is triggered.

❷ In line 7, we're using a new macro to create a BPF Histogram map. This is not a native BPF map, BCC includes this macro to make it

easier for you to create these visualizations. Under the hood, this BPF Histograms use Array maps to store the information, and several helpers to do the bucketing and create the final graph. The function in line 9 will look familiar to you, we took it from the previous Uprobes example, we're using the programs PID to store when the application triggers the instruction we want to trace.

Let's see how we calculate the delta for the latency and we store it in our histogram. The initial lines of this new block of code will also look familiar, we're still following the example we talked about in the Uprobes section of this chapter. The interesting lines in this block are line 9 and 10.

```
1 bpf_source += """
2 int trace_bpf_prog_load_return(void ctx) {
3   u64 *start_time_ns, delta;
4   u64 pid = bpf_get_current_pid_tgid();
5   start_time_ns = cache.lookup(&pid);
6   if (start_time_ns == 0)
7     return 0;
8
9   delta = bpf_ktime_get_ns() - *start_time_ns; ❶
10   histogram.increment(bpf_log2l(delta)); ❷
11   return 0;
12 }
13 """
```

❶ In line 9, we're calculating the delta between the time the instruction was invoked and the time it took our program to arrive here, we can assume it's also the time the instruction completed.

❷ In line 10, we're storing that delta in our histogram. We do two operations in this line. First, we use the built-in function `bpf_log2l` to generate the bucket identifier for the value of delta. This function will create a stable distribution of values over time. Then, we use the `increment` function to add a new item to this bucket. By default, `increment` adds one to the value if the bucket existed in the histogram, or it starts a new bucket with the value of one, so you don't have to worry about whether the value exist or not in advance.

The last piece of code that we need to write will attach these two functions

to the valid Kprobes, and it will print the histogram in the screen so we can see the latency distribution. This section is where we initialize our BPF program and we wait for events to generate the histogram:

```
1 bpf = BPF(text = bpf_source) ❶
2 bpf.attach_kprobe(event = "bpf_prog_load", fn_name =
"trace_bpf_prog_load_start")
3 bpf.attach_kretprobe(event = "bpf_prog_load", fn_name =
"trace_bpf_prog_load_return")
4
5 try: ❷
6   sleep(99999999)
7 except KeyboardInterrupt:
8   print()
9
10 bpf["histogram"].print_log2_hist("msecs") ❸
```

❶ In lines from 1 to 3, we initialize BPF, and we attach our functions to Kprobes.

❷ Lines from 5 to 8 make our program to wait, so we can gather as many events as we need from our system.

❸ Line 10 is another BCC macro that's going to allow us to get the Histogram map, and it's going to print it in our terminal with the traced distribution of events.

As we mentioned at the beginning of this section, histograms can be very useful to observe anomalies in your system. The BCC tools include numerous scripts that make use of histograms to represent data, we highly recommend you to take a look at them when you need inspiration to dive into your system.

Perf events

In the words of my co-author Lorenzo, Perf events are probably the most important communication method that you need to master to use BPF tracing successfully. We talked about BPF Perf Event Array maps in our

previous chapter. They allow you to put data in a buffer ring that's synchronized in real time with user-space programs. This is ideal when you're collecting a large amount of data in your BPF program and want to off load processing and visualization to a user-space program. That will allow you to have more control over the presentation layer since you're not restricted by the BPF virtual machine regarding programming capabilities. Most of the BPF tracing programs that you can find only use perf events for this purpose.

Here, we're going to show you how to use them to extract information about binary execution, and classify that information to print which binaries are the most executed in your system. We're going to divide this example in two blocks of code so you can follow the example easily. In the first block, we're going to define our BPF program and attach it to a Kprobe, like we did in our Probes section.

```
1 from collections import Counter ❶
2 from bcc import BPF
3
4 bpf_source = """
5 #include <uapi/linux/ptrace.h>
6
7 BPF_PERF_OUTPUT(events); ❷
8
9 int do_sys_execve(struct pt_regs *ctx, void filename, void argv, void envp)
{
10   char comm[16];
11   bpf_get_current_comm(&comm, sizeof(comm));
12
13   events.perf_submit(ctx, &comm, sizeof(comm)); ❸
14   return 0;
15 }
16 """
17
18 bpf = BPF(text = bpf_source) ❹
19 execve_function = bpf.get_syscall_fnname("execve")
20 bpf.attach_kprobe(event = execve_function, fn_name = "do_sys_execve")
```

❶ We're going to use a Python Counter to aggregate the events we're receiving from our BPF program.

❷ BPF_PERF_OUTPUT is a convinient macro that BCC provides to declare Perf Events maps. We're naming this map *events*.

❸ Once we have the name of the program that the kernel is executed, we want to send it to user-space for aggregation. We do that with `perf_submit`, this function updates the perf events map with our new piece of information.

❹ We initialize the BPF program and attach it to the Kprobe to be triggered when a new program is executed in our system.

Now that we wrote the code to collect all programs that are executed in our system, we need to aggregate them in user-space. There is a lot of information in the next code snippet, we're going to walk you through the most important lines:

```
1 aggregates = Counter() ❶
2 def aggregate_programs(cpu, data, size): ❷
3   comm = bpf["events"].event(data)
4   aggregates[comm] += aggregates.get(comm, 0) + 1 ❸
5
6 bpf["events"].open_perf_buffer(aggregate_programs) ❹
7 while True:
8     try:
9       bpf.perf_buffer_poll() ❺
10     except KeyboardInterrupt:
11       break
12
13 for (comm, times) in aggregates.most_common(): ❻
14   print("Program {} executed {} times".format(comm, times))
```

❶ First, we declare a Counter to store our program information. We'll use the name of the program as the key, and the values will be counters.

❷ We are going to use the `aggregate_programs` function to collect the data from the Perf Events map. In the next line, you can see how we use the BCC macro to access the map and extract the next incoming data event from the top of the stack.

❸ Here, We increment the number of times we've received an event with the same program name.

❹ We use the function `open_perf_buffer` to tell BCC that it needs to execute the function `aggregate_programs` every time it receives an event from the Perf Events map.

❺ BCC is going to poll events until we interrupt this Python program, the longer you wait, the more information you're going to process.

❻ Finally, We use the `most_common` function to get the list of elements in the counter and loop to print the top executed programs in your system first.

Perf events can open the doors to processing all the data that BPF exposes in new novel and unexpected ways. We've shown you an example to inspire your imagination when you need to collect some kind of arbitrary data from the kernel, you can find many other examples in the tools that BCC provides for tracing.

Conclusion

In this chapter we've only scratched the surface of tracing with BPF. The Linux kernel gives you access to information that's harder to obtain with other tools. BPF makes this process more predictable because it provides a common interface to access this data. In future chapters, you'll see more examples that use some of the techniques described here, like attaching functions to tracepoints. They will help you cement what you've learned here.

We've used the BCC framework in this chapter to write most of the examples. You can implement the same examples in C, like we did in previous chapters, but BCC provides several builtin features that make writing tracing programs much more accessible than C. If you're up for a fun challenge, try rewriting these examples in C.

In the next chapter, we are going to show you some tools that the Systems community has built on top of BPF to do performance analysis and tracing. Writing your own programs is very powerful, but these dedicated tools give you access to much of the information we've seen here in packaged format. This way, you don't need to rewrite tools that already exist.

Chapter 4. Linux Networking and BPF

From a networking point of view, BPF programs are used for two main use cases: packet capturing and filtering.

This means that a user-space program can attach a filter to any socket and extract information about packets flowing through it and allow/disallow/redirect certain kinds of packets as they are seen at that level.

The goal of this chapter is to explain how BPF programs can interact with the Socket Buffer structure at different stages of the network data path in the Linux Kernel network stack. We identified, as common use cases two types of programs:

- Program types related to *sockets*
- Programs written for the BPF-based classifier for *Traffic Control*

Non goals of this chapter:

- Explain XDP, that is left for the next chapter
- Explain all the set of possible scenarios where BPF programs are involved in networking because they are all very similar and that’s immediately understandable by the fact that while looking at the two examples we have below, they are loaded in two completely different contexts but are still getting almost the same input and can do very similar actions. However, if you need an

hint on the other programs types you can take inspiration in Chapter 2.

THE SK_BUFF STRUCT

The Socket Buffer structure also called SKB or `sk_buff` is the one in the Kernel that gets created and used for every packet sent or received. By reading the SKB we can pass or drop packets and populate BPF maps to create statistics and flow metrics about the current traffic flowing, in addition, some BPF programs allow to manipulate it and by extension transform the final packets in order to redirect them or change their fundamental structure. For example on an IPv6 only system one might write a program that converts all the received packets form IPv4 to IPv6, this can be accomplished by mangling with the packets' SKB.

BPF and Packet filtering

Originally, BPF programs were synonymous with packet filtering, which is still one of the most important use cases that had been expanded from classic BPF (cBPF) to the modern eBPF in Linux 3.19 with the addition of maps related functions to the filters' program type `BPF_PROG_TYPE_SOCKET_FILTER`.

Filters can be used mainly in three high lever scenarios:

- Live traffic dropping, e.g: allow only UDP traffic and discard anything else
- Live observation of a filtered set of packets flowing into a live system;
- Retrospective analysis of network traffic captured on a live system, using the *pcap format* for example

THE PCAP FORMAT

The term pcap comes from the conjunction of two words (p) Packet and (cap) Capture. The pcap format is implemented as a domain specific application programming interface for packet capturing in a library called Packet Capture Library (libpcap). Such format is useful in debugging scenarios when you want to save a set of packets that have been captured on a live system directly to a file to analyze them later using a tool that can read a stream of packets exported in the pcap format.

In the following sections we are going to see two different ways to apply the concept of packet filtering using BPF programs. At first we will see how, a very common and widespread tool like `tcpdump` acts as an higher level interface for BPF programs used as filters, then we will write and load our own program using the BPF program type `BPF_PROG_TYPE_SOCKET_FILTER`.

Tcpdmp and BPF expressions

When talking about live traffic analysis and observation, one of the command line tools that almost anyone knows about is `tcpdump`. Tcpdump, essentially is a frontend for `libpcap`, it allows the user to define high level filtering expressions. What tcpdump does it that it reads packets from a network interface of your choice (or any interface) and then writes to stdout or a file the content of the packets it received. The packet stream can then be filtered using the pcap filter syntax. The **pcap filter** syntax is a DSL (domain specific language) that is used to filter packets using an higher level set of expressions made by a set of primitives that are generally easier to remember than BPF assembly. It's out of the scope of this chapter to explain all the primitives and expressions possible in the pcap filter syntax since the whole set can be found in `man 7 pcap-filter` but we will go through some examples to understand its power.

The scenario is: we are in a Linux box that is serving a webserver on port `8080`, this webserver is not logging the requests it receives and we really want to know if it is receiving any request and how those requests are flowing into it because a customer of the served application is complaining about not being able to get any response while browsing the products page. At this point, we only know that the customer is connecting to one of our products pages using our web application served by that web server, and as almost always happens we have no idea on what could be the cause of that because end-users generally don't try to debug your services for you and unfortunately we didn't deploy any logging or error reporting strategy into this system so we are completely blind while investigating the problem. Fortunately, there's a tool that can come to our rescue! It is `tcpdump` that can be told to filter only IPv4 packets flowing in our system that are using the TCP protocol on port `8080`. In that way we will be able to analyse the traffic of the webserver and understand what are the faulty requests.

Here's the command to conduct that filtering with `tcpdump`:

```
# tcpdump -n 'ip and tcp port 8080'
```

- `-n` is there to tell tcpdump to not convert addresses to the respective names, we want to see the addresses for source and destination;
- `ip and tcp port 8080` is the pcap filter expression that tcpdump will use to filter your packets. `ip` means `IPv4`, `and` is a conjunction to express a more complex filter to allow adding more expressions to match, and then we specify that we are only interested in TCP packets coming from or to port `8080` using `tcp port 8080`. In this specific case a better filter would've been `tcp dst port 8080` because we are only interested in packets having as destination the port `8080` and not also packets coming from it.

The output of that will be something like this (without the redundant parts like complete TCP handshakes)

```
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on wlp4s0, link-type EN10MB (Ethernet), capture size 262144 bytes
12:04:29.593703 IP 192.168.1.249.44206 > 192.168.1.63.8080: Flags [P.],
   seq 1:325, ack 1, win 343, options [nop,nop,TS val 25580829 ecr
595195678],
   length 324: HTTP: GET / HTTP/1.1
12:04:29.596073 IP 192.168.1.63.8080 > 192.168.1.249.44206: Flags [.],
   seq 1:1449, ack 325, win 507, options [nop,nop,TS val 595195731 ecr
25580829],
   length 1448: HTTP: HTTP/1.1 200 OK
12:04:29.596139 IP 192.168.1.63.8080 > 192.168.1.249.44206: Flags [P.],
   seq 1449:2390, ack 325, win 507, options [nop,nop,TS val 595195731 ecr
25580829],
   length 941: HTTP
12:04:46.242924 IP 192.168.1.249.44206 > 192.168.1.63.8080: Flags [P.],
   seq 660:996, ack 4779, win 388, options [nop,nop,TS val 25584934 ecr
595204802],
   length 336: HTTP: GET /api/products HTTP/1.1
12:04:46.243594 IP 192.168.1.63.8080 > 192.168.1.249.44206: Flags [P.],
   seq 4779:4873, ack 996, win 503, options [nop,nop,TS val 595212378 ecr
25584934],
   length 94: HTTP: HTTP/1.1 500 Internal Server Error
12:04:46.329245 IP 192.168.1.249.44234 > 192.168.1.63.8080: Flags [P.],
   seq 471:706, ack 4779, win 388, options [nop,nop,TS val 25585013 ecr
595205622],
   length 235: HTTP: GET /favicon.ico HTTP/1.1
12:04:46.331659 IP 192.168.1.63.8080 > 192.168.1.249.44234: Flags [.],
   seq 4779:6227, ack 706, win 506, options [nop,nop,TS val 595212466 ecr
25585013],
   length 1448: HTTP: HTTP/1.1 200 OK
12:04:46.331739 IP 192.168.1.63.8080 > 192.168.1.249.44234: Flags [P.],
   seq 6227:7168, ack 706, win 506, options [nop,nop,TS val 595212466 ecr
25585013],
   length 941: HTTP
```

The situation is a lot more clear now! We have a bunch of requests going well, returning a `200 OK` status code but there is also one with a `500 Internal Server Error` on the `/api/products` endpoint. Our customer is right, we have a problem listing the products!

At this point, you might ask yourself, what does all this pcap filtering stuff

and tcpdump have to do with BPF programs if they have their own syntax? Pcap filters, on Linux are **compiled to BPF programs**! And since tcpdump uses pcap filters for the filtering this means that everytime you execute `tcpdump` using a filter you are actually compiling and loading a BPF program to filter your packets. Fortunately, by passing the `-d` flag to `tcpdump` we can dump the BPF instructions that it will load while using the specified filter:

```
tcpdump  -d  'ip and tcp port 8080'
```

The filter is the same as the one used in the example above but the output now is a set of BPF assembly instructions because of the `-d` flag:

Here's the output:

```
(000) ldh      [12]
(001) jeq      #0x800           jt 2    jf 12
(002) ldb      [23]
(003) jeq      #0x6             jt 4    jf 12
(004) ldh      [20]
(005) jset     #0x1fff          jt 12   jf 6
(006) ldxb     4*([14]&0xf)
(007) ldh      [x + 14]
(008) jeq      #0x1f90          jt 11   jf 9
(009) ldh      [x + 16]
(010) jeq      #0x1f90          jt 11   jf 12
(011) ret      #262144
(012) ret      #0
```

Let's analyze it:

1. `ldh [12]`: (ld) Load a (h) half word (16 bit) from the accumulator at offset 12 that is the EtherType field as shown in Figure 4-1;
2. `jeq #0x800 jt 2 jf 12`: (j) jump if (eq) equal, check if the EtherType value from the previous instruction is equal to `0x800` which is the identifier for IPv4 and then use the jump destinations

that are `2` if true (jt) and `12` if false (jf), so this will continue to the next instruction if the internet protocol is IPv4 otherwise will jump to the end and return zero;

3. `ldb [23]`: Load byte into (ldb), will load the higher layer protocol field from the IP frame that can be found at offset `23`, offset `23` comes from the addition of the 14 bytes of the headers in the Ethernet Layer 2 frame (see Figure 4-1) plus the position the protocol has in the IPv4 header which is the 9th, so $14 + 9 = 23$;

4. `jeq #0x6 jt 4 jf 12`: again a jump if equal as in step 2, in this case we check that the previous extracted protocol is `0x6` which is TCP, if it is we jump to the next instruction (4) or we go to the end (12) if it is not and we drop the packet;

5. `ldh [20]`: Another load half word instruction, in this case it is to load the value of packet offset + fragment offset from the IPv4 header

6. `jset #0x1fff jt 12 jf 6`: This `jset` instruction will jump to `12` if any of the data we found in the fragment offset is true, otherwise go to `6` which is the next instruction. The offset after the instruction `0x1fff` says to the jset instruction to only look at the last 13 bytes of data, expanded it becomes `0001 1111 1111 1111`.

7. `ldxb 4*([14]&0xf)`: (ld) load, into x (x), what (b) - the byte. This instruction will load the value of the ip header length into `x`;

8. `ldh [x + 14]`: Another load half word instruction that will go get the value at offset (x + 14), ip header length + 14, the location of the source port within the packet;

9. `jeq #0x1f90 jt 11 jf 9`: If the value at (x+14) is equal to `0x1f90` (`8080` in decimal) that means that the source port will be `8080` so let's continue to `11` or go check if destination is on port `8080` by continuing to `9` if this is false;

10. `ldh [x + 16]`: Same as step 8, a load half word instruction that will go get the value at offset (x + 16), the location of destination port in the packet;
11. `jeq #0x1f90 jt 11 jf 12`: ssame check as step 9, this time if the destination is `8080` go to `11` if not go to `12` and discard the packet;
12. `ret #262144`: when this instruction is reached a match is found, thus return the matched snap length, by default this value is of 262144 bytes. It can be tuned using the `-s` parameter in tcpdump.

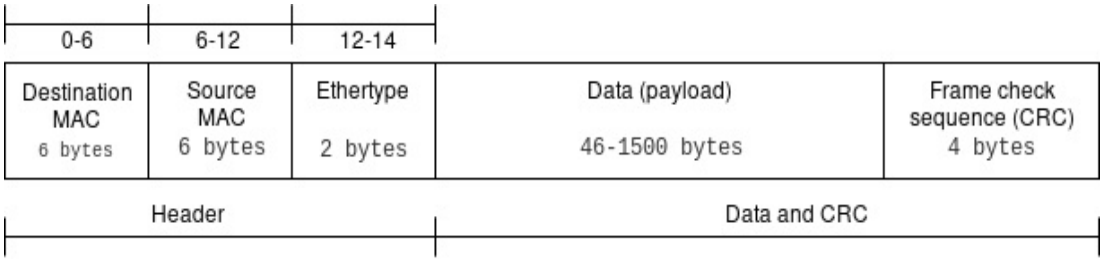


Figure 4-1. Layer 2 Ethernet frame structure

Here's the "correct" example, as said, in the case of our webserver, we only need to take in account packet having `8080` as destination not as a source, so the tcpdump filter can specify it with the `dst` destination field.

```
tcpdump -d 'ip and tcp dst port 8080'
```

In this case the dumped set of instructions is very similar to the previous example, but as you can see it lacks the whole part relative to matching the packets having as source a port 8080, in fact there's no `ldh [x + 14]` and the relative `jeq #0x1f90 jt 11 jf 9`.

```
(000) ldh      [12]
(001) jeq      #0x800           jt 2    jf 10
(002) ldb      [23]
(003) jeq      #0x6             jt 4    jf 10
(004) ldh      [20]
(005) jset     #0x1fff          jt 10   jf 6
(006) ldxb     4*([14]&0xf)
(007) ldh      [x + 16]
```

```
(008) jeq      #0x1f90          jt 9    jf 10
(009) ret      #262144
(010) ret      #0
```

Beside just analyzing the generated assembly from Tcpdump, as we did, one might want to write their own code to filter network packets, it turns out that the biggest challenge in that case would be to actually debug the execution of the code to make sure it matches our expectations, in this case, in the kernel source tree there's a tool in `tools/bpf` called `bpf_dbg.c` that is essentially a debugger that allows you to load a program and a pcap file to test the execution step by step.

> **NOTE**
>
> Tcpdump can also read directly from a `.pcap` file and apply BPF filters to it.

Packet filtering for raw sockets (BPF_PROG_TYPE_SOCKET_FILTER)

The `BPF_PROG_TYPE_SOCKET_FILTER` program type allows to attach the BPF program to a socket, all the packets received by it will be passed to the program in the form of an `sk_buff` struct and then the program can decide whether to discard or allow them. This kind of programs also has the ability to access and work on maps.

But let's see how this kind of BPF program can be used with an example!

The purpose of our example program is to count the number of TCP, UDP and ICMP packets flowing in the interface under observation. To do that we need:

- The BPF program that can see the packets flowing;

- The code to load the program and attach it to a network interface;
- A script to compile the program and launch the loader;

At this point, our BPF program can be written in two ways, as C code that is then compiled to an ELF file or directly as BPF Assembly, for this example we opted to use C code in order to show an higher level abstraction and how to use Clang to compile the program, it's important to note that to make this program we are using headers and helpers only available in the Linux Kernel's source tree, so the first thing to do is to obtain a copy of it using Git, to avoid differences you can checkout the same commit SHA we've used to make this example:

```
export KERNEL_SRCTREE=/tmp/linux-stable
git clone  git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-
stable.git $KERNEL_SRCTREE
cd $KERNEL_SRCTREE
git checkout 4b3c31c8d4dda4d70f3f24a165f3be99499e0328
```

CLANG SUPPORT FOR BPF

In order to contain BPF support you will need clang >= 3.4.0 with llvm >= 3.7.1, to verify bpf support in your installation you can use the command `llc -version` and look if it has the bpf target.

THE BPF PROGRAM

The main duty of the BPF program here is to access the packet it receives, check if its protocol is TCP, UDP or ICMP and then increment the counter on the map array on the specific key for the found protocol.

For this program we are going to leverage the loading mechanism that parses ELF files using the helpers located in `samples/bpf/bpf_load.c` in the kernel source tree. The load function `load_bpf_file` is able to

recognize some specific ELF section headers and can associate them to the respective program types, here's how that code looks like:

```
        bool is_socket = strncmp(event, "socket", 6) == 0;
        bool is_kprobe = strncmp(event, "kprobe/", 7) == 0;
        bool is_kretprobe = strncmp(event, "kretprobe/", 10) == 0;
        bool is_tracepoint = strncmp(event, "tracepoint/", 11) == 0;
        bool is_raw_tracepoint = strncmp(event, "raw_tracepoint/", 15) == 0;
        bool is_xdp = strncmp(event, "xdp", 3) == 0;
        bool is_perf_event = strncmp(event, "perf_event", 10) == 0;
        bool is_cgroup_skb = strncmp(event, "cgroup/skb", 10) == 0;
        bool is_cgroup_sk = strncmp(event, "cgroup/sock", 11) == 0;
        bool is_sockops = strncmp(event, "sockops", 7) == 0;
        bool is_sk_skb = strncmp(event, "sk_skb", 6) == 0;
        bool is_sk_msg = strncmp(event, "sk_msg", 6) == 0;
```

The first thing that the code does is an association between the section header and an internal variable, like for `SEC("socket")` we will end up with `bool is_socket=true`.

Later in the same file, we see a set of if instructions that created the association between the header and the actual prog type, so for `is_socket` we end up with `BPF_PROG_TYPE_SOCKET_FILTER`.

```
        if (is_socket) {
                prog_type = BPF_PROG_TYPE_SOCKET_FILTER;
        } else if (is_kprobe || is_kretprobe) {
                prog_type = BPF_PROG_TYPE_KPROBE;
        } else if (is_tracepoint) {
                prog_type = BPF_PROG_TYPE_TRACEPOINT;
        } else if (is_raw_tracepoint) {
                prog_type = BPF_PROG_TYPE_RAW_TRACEPOINT;
        } else if (is_xdp) {
                prog_type = BPF_PROG_TYPE_XDP;
        } else if (is_perf_event) {
                prog_type = BPF_PROG_TYPE_PERF_EVENT;
        } else if (is_cgroup_skb) {
```

```
                prog_type = BPF_PROG_TYPE_CGROUP_SKB;
        } else if (is_cgroup_sk) {
                prog_type = BPF_PROG_TYPE_CGROUP_SOCK;
        } else if (is_sockops) {
                prog_type = BPF_PROG_TYPE_SOCK_OPS;
        } else if (is_sk_skb) {
                prog_type = BPF_PROG_TYPE_SK_SKB;
        } else if (is_sk_msg) {
                prog_type = BPF_PROG_TYPE_SK_MSG;
        } else {
                printf("Unknown event '%s'\n", event);
                return -1;
        }
```

Good, so in our case the program we wanted to write is of type `BPF_PROG_TYPE_SOCKET_FILTER` so we will need to specify a `SEC("socket")` as ELF header to our function that will act as entrypoint for our BPF program.

As you can see by that list, there are a variety of program types related to sockets and in general network operations. In this chapter we are doing examples with `BPF_PROG_TYPE_SOCKET_FILTER`, however you can find a definition of all the other program types in Chapter 2. Moreover in the next chapter we are going to discuss about XDP programs with the program type `BPF_PROG_TYPE_XDP`.

Since we want to count packets for every protocol we encounter, we need to create a map that can hold a key valuer pair <protocol, count of packets>. We can use a `BPF_MAP_TYPE_ARRAY` for that purpose:

```
1 struct bpf_map_def SEC("maps") countmap = {
2     .type = BPF_MAP_TYPE_ARRAY,
3     .key_size = sizeof(int),
4     .value_size = sizeof(int),
5     .max_entries = 256,
```

```
6 };
```

The map is defined using the `bpf_map_def` struct, and it will be named `countmap` for reference in the program.

At this point, we can write some code to actually count the packets, we know that programs of type `BPF_PROG_TYPE_SOCKET_FILTER` are one of our options because by using such kind of programs we can see all the packets flowing through an interface so we will attach the program the right header with `SEC("socket")`.

```
1 SEC("socket")
2 int socket_prog(struct __sk_buff *skb) {
3   int proto = load_byte(skb, ETH_HLEN + offsetof(struct iphdr, protocol));
4   int one = 1;
5   int *el = bpf_map_lookup_elem(&countmap, &proto);
6   if (el) {
7     (*el)++;
8   } else {
9     el = &one;
10   }
11   bpf_map_update_elem(&countmap, &proto, el, BPF_ANY);
12   return 0;
13 }
```

After the ELF header attachment we can now use the `load_byte` function to extract to extract the protocol section from the `sk_buff` struct, then we use the protocol id as a key to do a `bpf_map_lookup_elem` to extract the current counter value from our `countmap` so that we can increment it or set it to one if it is the first packet ever. At this point we can update the map with the incremented value, using `bpf_map_update_elem`.

To compile the program to an ELF file then, we just use Clang with `-target bpf`, this command will create a `bpf_program.o` file that we will

load using the loader.

```
#!/bin/bash
clang -O2 -target bpf -c bpf_program.c -o bpf_program.o
```

LOAD AND ATTACH TO A NETWORK INTERFACE

The loader is the program that actually opens our compiled BPF ELF binary `bpf_program.o`, and attach the defined BPF program and its maps to a socket that is created against the interface under observation, in our case `lo`, the loopback interface.

The most important part of the loader is the actual load of the ELF file: .loader.c

```
if (load_bpf_file(filename)) {
  printf("%s", bpf_log_buf);
  return 1;
}

sock = open_raw_sock("lo");

if (setsockopt(sock, SOL_SOCKET, SO_ATTACH_BPF, prog_fd,
               sizeof(prog_fd[0]))) {
   printf("setsockopt %s\n", strerror(errno));
   return 0;
 }
```

This will populate the `prog_fd` array by adding one element that is the file descriptor of our loaded program that we can now attach to the socket descriptor of our loopback interface `lo` opened with `open_raw_sock`.

The attach is done by setting an option `SO_ATTACH_BPF` to the raw socket opened for the interface.

At this point our user space loader is able lookup map element while the

kernel sends them:

```
  for (i = 0; i < 10; i++) {
    key = IPPROTO_TCP;
    assert(bpf_map_lookup_elem(map_fd[0], &key, &tcp_cnt) == 0);

    key = IPPROTO_UDP;
    assert(bpf_map_lookup_elem(map_fd[0], &key, &udp_cnt) == 0);

    key = IPPROTO_ICMP;
    assert(bpf_map_lookup_elem(map_fd[0], &key, &icmp_cnt) == 0);

     printf("TCP %d UDP %d ICMP %d packets\n", tcp_cnt, udp_cnt, icmp_cnt);
     sleep(1);
   }
```

To do the lookup we attach to the array map using a for loop and `bpf_map_lookup_elem` so that we can read and print the values for the TCP, UDP and ICMP packet counters respectively.

At this point the only thing left is to compile our program!

Since this program is using `libbpf`, we need to compile it from the kernel source tree we just cloned:

```
$ cd $KERNEL_SRCTREE/tools/lib/bpf
$ make
```

Now that we have `libbpf` we can compile the loader using this script:

```
#!/bin/bash

KERNEL_SRCTREE=$1
LIBBPF=${KERNEL_SRCTREE}/tools/lib/bpf/libbpf.a
clang -o loader-bin -I${KERNEL_SRCTREE}/tools/lib/bpf/ \
  -I${KERNEL_SRCTREE}/tools/lib -I${KERNEL_SRCTREE}/tools/include \
  -I${KERNEL_SRCTREE}/tools/perf -I${KERNEL_SRCTREE}/samples \
  ${KERNEL_SRCTREE}/samples/bpf/bpf_load.c \
  loader.c "${LIBBPF}" -lelf
```

As you can see, the script includes a bunch of headers and the `libbpf` library from the kernel itself, so it has to know where to find the kernel source code, to do that you can replace `$KERNEL_SRCTREE` in it or just write that script into a file and use it

```
$ ./build-loader.sh /tmp/linux-stable
```

At this point the loader will have created a `loader-bin` file that can be finally started along with the BPF program's ELF file (requires root privileges):

```
# ./loader-bin bpf_program.o
```

Once the program is loaded and started it will do ten dumps, one every second showing the packet count for each one of the three considered protocols. Since the program is attached to the loopback device `lo`, along with the loader you can run ping and see the ICMP counter increasing.

So run ping to generate ICMP traffic to localhost:

```
$ ping -c 100 127.0.0.1
```

This will start pinging localhost 100 times and will output something like:

```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.100 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.107 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.093 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.102 ms
64 bytes from 127.0.0.1: icmp_seq=5 ttl=64 time=0.105 ms
64 bytes from 127.0.0.1: icmp_seq=6 ttl=64 time=0.093 ms
64 bytes from 127.0.0.1: icmp_seq=7 ttl=64 time=0.104 ms
64 bytes from 127.0.0.1: icmp_seq=8 ttl=64 time=0.142 ms
```

Then, in another terminal we can finally run our BPF program:

```
# ./loader-bin bpf_program.o
```

It will start dumping out the

```
TCP 0 UDP 0 ICMP 0 packets
TCP 0 UDP 0 ICMP 4 packets
TCP 0 UDP 0 ICMP 8 packets
TCP 0 UDP 0 ICMP 12 packets
TCP 0 UDP 0 ICMP 16 packets
TCP 0 UDP 0 ICMP 20 packets
TCP 0 UDP 0 ICMP 24 packets
TCP 0 UDP 0 ICMP 28 packets
TCP 0 UDP 0 ICMP 32 packets
TCP 0 UDP 0 ICMP 36 packets
```

BPF Based Traffic Control Classifier

Traffic control is the kernel packet scheduling subsystem architecture, it is made of mechanisms and queuing systems that can decide how packets flow and how they are accepted.

Some use cases for Traffic control are, but not limited to:

- Prioritize certain kinds of packets
- Drop specific kind of packet
- Bandwidth distribution

Given that, in general Traffic Control is the way to go when you need to redistribute network resources in a system, and to get the best out of it, specific Traffic Control configurations should be deployed based on the kind of applications one wants to run. Traffic control, provides a programmable classifier, called `cls_bpf`, to let the usage of BPF programs as hook at different levels of the scheduling operations where they can read and update socket buffer and packet metadata in order to do things like traffic shaping, tracing, pre-processing and more.

Support for eBPF, in `cls_bpf`, was implemented in Kernel 4.1, that means that this kind of programs have access to eBPF maps, have tail calls support, can access IPv4/IPv6 tunnel metadata and in general leverage helpers and utilities coming with eBPF.

The tooling used to interact with networking configuration related to traffic control are part of the iproute2 suite, which contains `ip` and `tc`, used respecitevly to manipulate network interfaces and traffic control configuration.

THE SK_BUFF STRUCT

The Socket Buffer structure also called SKB or `sk_buff` is the one in the Kernel that gets created and used for every packet sent or received. By reading the SKB we can pass or drop packets and populate BPF maps to create statistics and flow metrics about the current traffic. In addition, some BPF programs allow to manipulate it and by extension transform the final packets in order to redirect them or change their fundamental structure. For example on an IPv6 only system one might write a program that converts all the received packets form IPv4 to IPv6, this can be accomplished by mangling with the packets' SKB.

Terminology

As said, there are interaction points between Traffic Control and BPF programs but in order to understand that we need to introduce som Traffic Control concepts that not everyone might be familiar with, if you already master Traffic Control, feel free to skip this terminology section and go straight to the examples.

QUEUEING DISCIPLINES (QDISC)

Queuing disciplines define the scheduling objects used to enqueue packets

going to an interface by changing the way they are sent, those objects can be classless or classful.

The default qdisc is `pfifo_fast` is classless and enqueues packets on three FIFO (first in first out) queues that are dequeued based on their priority, this qdisc is not used for virtual devices like the loopback (lo) or Virtual Ethernet devices (veth) that use `noqueue` instead. Besides being a good default for its scheduling algorithm, `pfifo_fast` also doesn't require any configuration to work.

Virtual interfaces can be distinguished from physical interfaces (devices) by asking the `/sys` pseudo filesystem:

```
ls -la /sys/class/net
total 0
drwxr-xr-x  2 root root 0 Feb 13 21:52 .
drwxr-xr-x 64 root root 0 Feb 13 18:38 ..
lrwxrwxrwx  1 root root 0 Feb 13 23:26 docker0 ->
../../devices/virtual/net/docker0
lrwxrwxrwx  1 root root 0 Feb 13 23:26 enp0s31f6 ->
../../devices/pci0000:00/0000:00:1f.6/net/enp0s31f6
lrwxrwxrwx  1 root root 0 Feb 13 23:26 lo -> ../../devices/virtual/net/lo
```

At this point, some confusion is normal, if you never heard about qdiscs one thing you can do is to use the `ip a` command to show the list of the network interfaces configured in the current system:

```
ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
    valid_lft forever preferred_lft forever
2: enp0s31f6: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc fq_codel
stateDOWN group default
qlen 1000
link/ether 8c:16:45:00:a7:7e brd ff:ff:ff:ff:ff:ff
```

```
6: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state
DOWN group default
link/ether 02:42:38:54:3c:98 brd ff:ff:ff:ff:ff:ff
inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
   valid_lft forever preferred_lft forever
inet6 fe80::42:38ff:fe54:3c98/64 scope link
   valid_lft forever preferred_lft forever
```

This list already tells us us something, can you find the word `qdisc` in it? Let's analyze the situation:

- We have three network interfaces in this system, `lo`, `enp0s31f6` and `docker0`
- The `lo` interface, is a virtual interface, so has qdisc `noqueue`
- The `enp0s31f6` is a physical interface, wait, why the qdisc here is `fq_codel` (Fair queue controlled delay)? Wasn't `pfifo_fast` the default? It turns out that the system I'm testing the commands on is running systemd, which is setting the default qdisc differently using the kernel parameter `net.core.default_qdisc`.
- The `docker0` interface is a bridge interface and so it uses a `virtual` device, and has `noqueue` qdisc.

The `noqueue` qdisc doesn't have classes, a scheduler or a classifier, what it does is that it tries to send the packets immediately, as said, `noqueue` is used by default by virtual devices, but it's also the qdisc that becomes effective to any interface when you delete it's current associated qdisc.

Fair queue controlled delay `fq_codel` is a classless qdisc that classifies the incoming packets using a stochastic model in order to be able to queue traffic flows in a **fair** way.

The situation should be more clear now, we used the `ip` command to find information about `qdiscs` but it turns out that in the `iproute2` toolbelt there's also a tool called `tc` that has a specific subcommand for qdiscs you

can use to list them:

```
tc qdisc ls
qdisc noqueue 0: dev lo root refcnt 2
qdisc fq_codel 0: dev enp0s31f6 root refcnt 2 limit 10240p flows 1024 quantum
1514
target 5.0ms interval 100.0ms memory_limit 32Mb ecn
qdisc noqueue 0: dev docker0 root refcnt 2
```

There's much more going on here! For `docker0` and `lo` we basically see the same information as with `ip a` but for `enp0s31f6`, for example it has:

- A limit of 10240 incoming packets it can handle
- As said the stochastic model used by `fq_codel` wants to queue traffic into different flows, this output contains the information on how many of them we have, 1024

CLASSFUL QDISCS, FILTERS AND CLASSES

Classful qdiscs allow the definition of classes for different kinds of traffic in order to apply differnt rules to them. Having a class for a qdisc means that it can contain further qdiscs, with this kind of hiearchy, then, a filter (classifier) can be used to **classify the traffic** by determining the next class where the packet should be enqueued.

Filters are used to assign packets to a particular class based on their type. Filters are used inside a classful qdiscs to determine in which class the packet should be enqueued, two or more filters can map to the same class as shown in Figure 4-2. Every filter uses a **classifier** to classify packets based on their information.

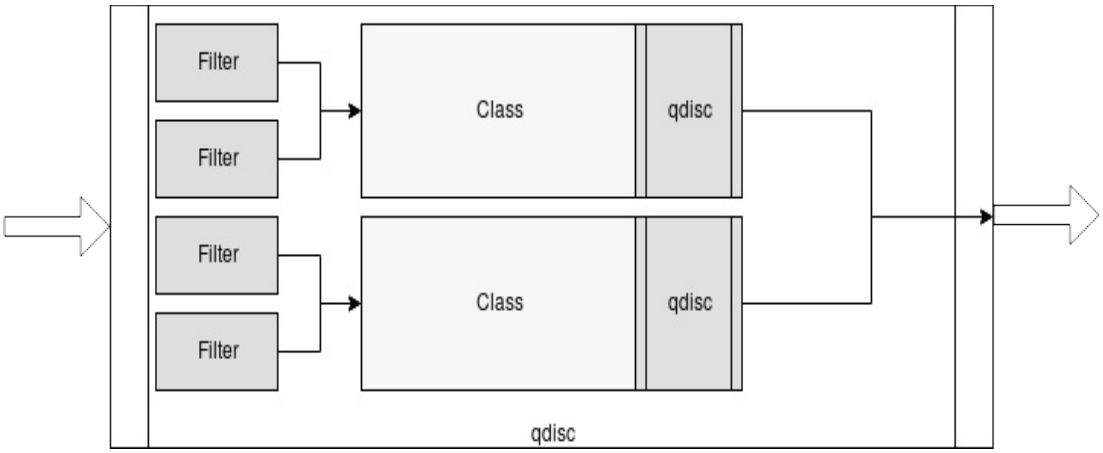


Figure 4-2. Classful qdisc with filters

As said in the introudction, `cls_bpf` is the classifier we want to use to write BPF programs for Traffic Control, this chapter has a concrete example on how to use it in the next sections.

Classes are objects that can only live in a classful qdisc, classes are used in traffic control to create hierarchies. Complex hierarchies are made possible by the fact that a class can have filters attached to it, which can then be used as an entrypoint for another class or for a qdisc.

CLASSLESS QDISCS

A classless qdiscs is just a qdisc but that can’t have any children because it is not allowed to have any classes associated, this means that is not possible to attach filters to them.

Traffic Control classifier program using cls_bpf

As we said, Traffic Control is a very powerful mechanism that is made even more powerful thanks to classifiers, however, among all the classifiers there is one that allows to program the network datapath `cls_bpf` classifier. This classifier is special because it can run BPF programs, but what does that mean? That means that `cls_bpf` will allow

you to hook your BPF programs right in the Ingress and Egress layers and running BPF programs hooked to those layers means that they will be able to access the `sk_buff` struct for the respective packets.

To understand better this relationship b etween Traffic Control and BPF programs, see Figure 4-3 that shows how BPF programs are loaded against the `cls_bpf` Classifier and hooked on Ingress and Egress and that describes all the other interactions in context, by taking the Network Interface as the entry point for network traffic you will notice that traffic then goes to the Traffic Control's Ingress hook, that has a BPF program loaded into it from userspace, once that program is executed, the control is given to the Networking Stack that then informs the user's application that in turn does the processing and gives a response that is now handled in the same way by the Traffic Control's Egress using another BPF program that once finished gives back control and then the user gets a response.

INGRESS AND EGRESS

Ingress and Egress qdiscs allow to hook traffic control respectively into Inboud (Ingress) and Outbound (Egress) traffic.

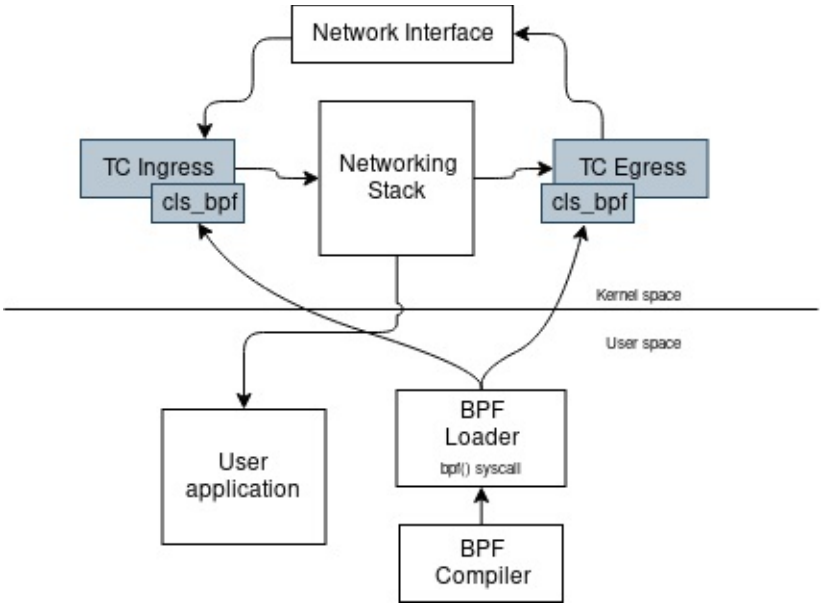


Figure 4-3. Loading of BPF programs using Traffic Control

BPF programs for Traffic Control can be written in C and compiled using LLVM/Clang with the bpf backend.

In order to make this example work, we will need to run it on a kernel that had been compiled with `cls_bpf` directly or as a module. To verify that you have everything you need you can do:

```
cat /proc/config.gz| zcat  | grep -i BPF
```

From the output, make sure you get at least the following ones with either `y` or `m`:

```
CONFIG_BPF=y
CONFIG_BPF_SYSCALL=y
CONFIG_NET_CLS_BPF=m
CONFIG_BPF_JIT=y
CONFIG_HAVE_EBPF_JIT=y
CONFIG_BPF_EVENTS=y
```

Let’s now see how we write our classifier:

```
1 SEC("classifier")
```

```
2 static inline int classification(struct __sk_buff *skb) {
3   void *data_end = (void *)(long)skb->data_end;
4   void *data = (void *)(long)skb->data;
5   struct ethhdr *eth = data;
6
7   int rc = TC_ACT_OK;
8   __u16 h_proto;
9   __u64 nh_off = 0;
10   nh_off = sizeof(*eth);
11
12   if (data + nh_off > data_end) {
13     return TC_ACT_OK;
14   }
```

The “main” of our classifier is the `classification` function. This function is annotated with a section header called `classifier` so that `tc` can know that this is the classifier to use.

At this point we need to extract some information from the `skb`, the `data` member contains all the data for the current packet and all its protocol details.

Given that, in order to let our program know what’s inside of it we will need to cast it to an Ethernet frame in our case with the `*eth` variable.

To make the static verifier happy we need to check that the data, summed up with the size of the `eth` pointer does not exceed the space where `data_end` is.

After that, we can go one level inner and get the protocol type from the `h_proto` member in `*eth`.

```
1   h_proto = eth->h_proto;
2
3   if (h_proto == bpf_htons(ETH_P_IP)) {
```

```
    if (is_http(skb, nh_off) == 1) {
      trace_printk("Yes! It is HTTP!\n");
    }
  }

  return TC_ACT_OK;
}
```

Once we have the protocol, it needs to be converted from host to check if it is equal to the IPv4 protocol, the one we are interested in and if it is then we check whether the inner packet is http using our own `is_http` function, if it is, we print a debug message stating that we found an http packet.

```
static inline int is_http(struct __sk_buff *skb, __u64 nh_off) {
  void *data_end = (void *)(long)skb->data_end;
  void *data = (void *)(long)skb->data;
  struct iphdr *iph = data + nh_off;

  if (iph + 1 > data_end) {
    return 0;
  }

   if (iph->protocol != IPPROTO_TCP) {
     return 0;
   }
```

The `is_http` function is very similar to our classifier function but it will start from an `skb` by knowing already the start offset for the IPv4 protocol data.

As we did before, we need to do a check before accessing the ip protocol data with the `*iph` variable to let the static verifier know our intentions.

Once that is done, we just check if the IPv4 header contains a TCP packet so that we can go ahead, if the packet's protocol is of type `IPPROTO_TCP`

then we have to do some more checks again to go and get the actual TCP header in the `*tcph` variable.

```
1   poffset = ETH_HLEN + ip_hlen + tcp_hlen;
2   plength = ip_total_length - ip_hlen - tcp_hlen;
3   if (plength >= 7) {
4     unsigned long p[7];
5     int i = 0;
6     for (i = 0; i < 7; i++) {
7
8       p[i] = load_byte(skb, poffset + i);
9     }
10     int *value;
11     if ((p[0] == 'H') && (p[1] == 'T') && (p[2] == 'T') && (p[3] ==
'P')) {
12       return 1;
13     }
14   }
15
16   return 0;
```

Now that the TCP header is ours, we can go ahead and load the first seven bytes from the `skb` struct at the offset of the TCP payload `poffset`. At this point we can check if the bytes array is a sequence saying `HTTP`, then we know that the Layer 7 protocol is HTTP and we can return 1, otherwise zero.

As you can see, our program is very simple, it will basically allow everything and when receiving an HTTP packet it will let us know with a debugging message.

The program can be compiled with Clang, using the bpf target, as we did before with the socket filter example. We can no compile this program for traffic control in the exact same way, this will generate an ELF file `classifier.o` that will be loaded by `tc` this time and not by an our own

custom loader.

```
clang -O2 -target bpf -c classifier.c -o classifier.o
```

TRAFFIC CONTROL RETURN CODES

From: `man 8 tc-bpf`

- TC_ACT_OK (0) , will terminate the packet processing pipeline and allows the packet to proceed
- TC_ACT_SHOT (2) , will terminate the packet processing pipeline and drops the packet
- TC_ACT_UNSPEC (-1) , will use the default action configured from tc (similarly as returning -1 from a classifier)
- TC_ACT_PIPE (3) , will iterate to the next action, if available
- TC_ACT_RECLASSIFY (1) , will terminate the packet processing pipeline and start classification from the beginning else , everything else is an unspecified return code

Yes! Now we can install the program on the interface we want our program to operate on, in my case it was `eth0`.

The first command will replace the default qdisc for the `eth0` device and then second one will actually load our `cls_bpf` classifier into that `ingress` classful qdisc. This means that our program will handle all traffic going into that interface, if we wanted to handle outgoing traffic we had to use the `egress` qdisc instead.

```
# tc qdisc add dev eth0 handle 0: ingress
# tc filter add dev eth0 ingress bpf obj classifier.o flowid 0:
```

Ok our program is loaded now, what we need is to send some HTTP traffic to that interface.

To do that you need any HTTP server on that interface, then you can curl the interface ip:

In case you haven't one, you can obtain a test http server using Python 3 with the `http.server` module. It will open the port `8000` with a directory listing of the current working directory.

```
python3 -m http.server
```

At this point you can call the server with cURL:

```
$ curl http://192.168.1.63:8080
```

After doing that you should see your HTTP response already, good that is an HTTP server then, you can now get your debugging messages (created with `trace_printk`), confirming that using the dedicated `tc` command:

```
# tc exec bpf dbg
```

The output will be something like this

```
Running! Hang up with ^C!

         python3-18456 [000] ..s1 283544.114997: 0: Yes! It is HTTP!
         python3-18754 [002] ..s1 283566.008163: 0: Yes! It is HTTP!
```

Congratulations! You just made your first BPF Traffic Control Classifier!

> **TIP**
>
> Instead of using a debugging message like we did in this example, you could use a map to communicate to user space that the interface just received an HTTP packet. Doing this is left as an exercise to the reader. If you look at `classifier.c` from the previous example, you can take an hint on how to do that by looking at how we used the map `countmap` there.

At this point what you might want is to unload the classifier, you can do that by deleting the ingress qdisc we just attached to the interface:

```
# tc qdisc del dev eth0 ingress
```

NOTES ON ACT_BPF AND HOW CLS_BPF IS DIFFERENT

The careful reader, might have noticed that another object exists for BPF programs, called `act_bpf`. It turns out that `act_bpf` is an action and not a classifier this makes it operationally different since actions are object attached to filters and because of this it is not able to perform filtering directly requiring TC to consider all the packets first. `act_bpf` can be attached to any classifier.

Differences between TC and XDP

Even if TC `cls_bpf` and XDP programs look very similar, they are pretty different, XDP programs are executed earlier in the ingress data path, before entering in the main kernel network stack so our program does not have access to a Socket buffer struct `sk_buff` like with tc, XDP programs instead take a different structure called `xdp_buff` that is an eager representation of the packet without metadata. All this comes with advantages and disadvantages, e.g: being executed even before the kernel code, XDP programs can drop packets in a very efficient way. Compared to tc programs, XDP programs can only be attached to traffic in ingress to the system. But at this point, let’s not spoil all the content of the next chapter!

Conclusion

At this point it should be pretty clear to you that BPF programs are very useful to get visibility and control at different levels of the networking data path. We've seen how to take advantage of them to filter packets using high level tools that generate BPF assembly, then we loaded a program to a network socket and in the end we attached our programs to the traffic control ingress qdisc to do traffic classification using BPF programs. In this chapter we also briefly discussed about XDP, but be prepared, because in the next chapter we're covering the topic in its entirety.

Index