Programmatic on Linux Kernel with eBPF

- DevClub Tech Meetup#6 eBPF -

Jumpbox®

JoJo

(Cloud Native Stylist)

Jumpbox®

JoJo

(Cloud Native Stylist)

THAI PROGRAMMER
สมาคมโปรแกรมเมอร์ไทย

กรรมการ และประธานคณะทำงาน

Jumpbox®

"เราเชื่อว่าการเรียนรู้ทำให้ชีวิตคุณดีขึ้น"

-Jumpbox Team-

Jumpbox®

Jumpbox®

Tech Passion | Sharing | Society

Why eBPF?

eBPF Foundation

Initial release: 2014; 10 years ago

Open source community, Meta, Google, Isovalent, Microsoft, Netflix

Reference:

- eBPF Foundation
- eBPF

Everything You Need to Know in 5 Minutes

- DevClub Tech Meetup#6 eBPF -

Jumpbox®

eBPF has become the key technology for

Jumpbox®

eBPF has become the key technology for
infrastructure software

Jumpbox®

Accurate Definition

"eBPF is a programming language & runtime to extend operating system"

-Thomas Graf, CTO, Isovalent -

CLOUD NATIVE eBPF DAY NORTH AMERICA

KubeCon 2022

Reference:

- Keynote: eBPF
- KubeCon2022 eBPF Day North America

Practical Comparison

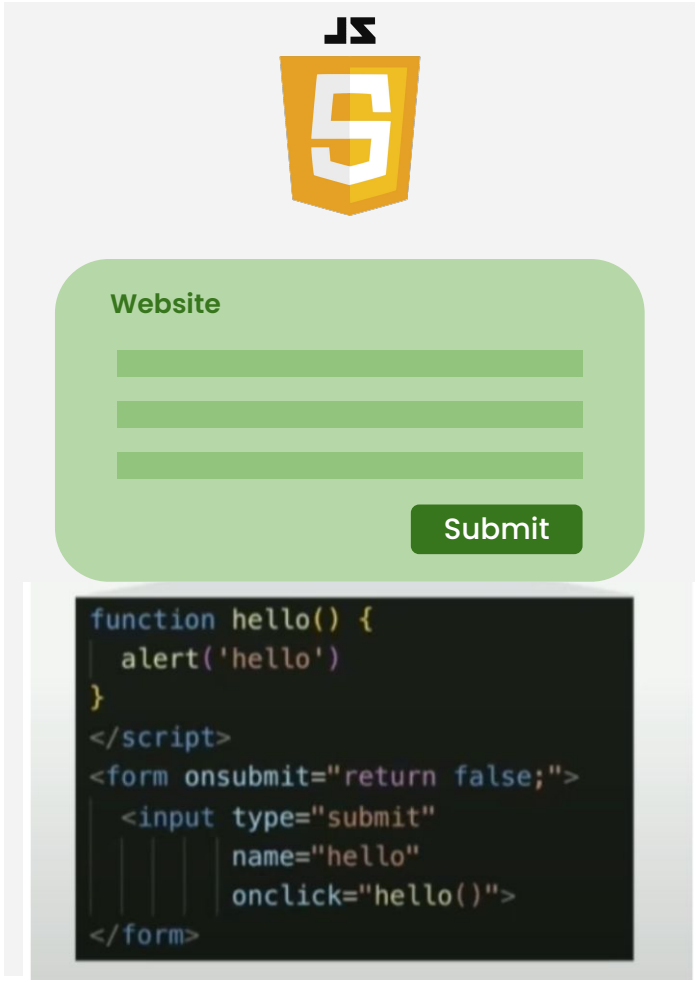
"eBPF is like JavaScript/Lua but for Kernel Developers"

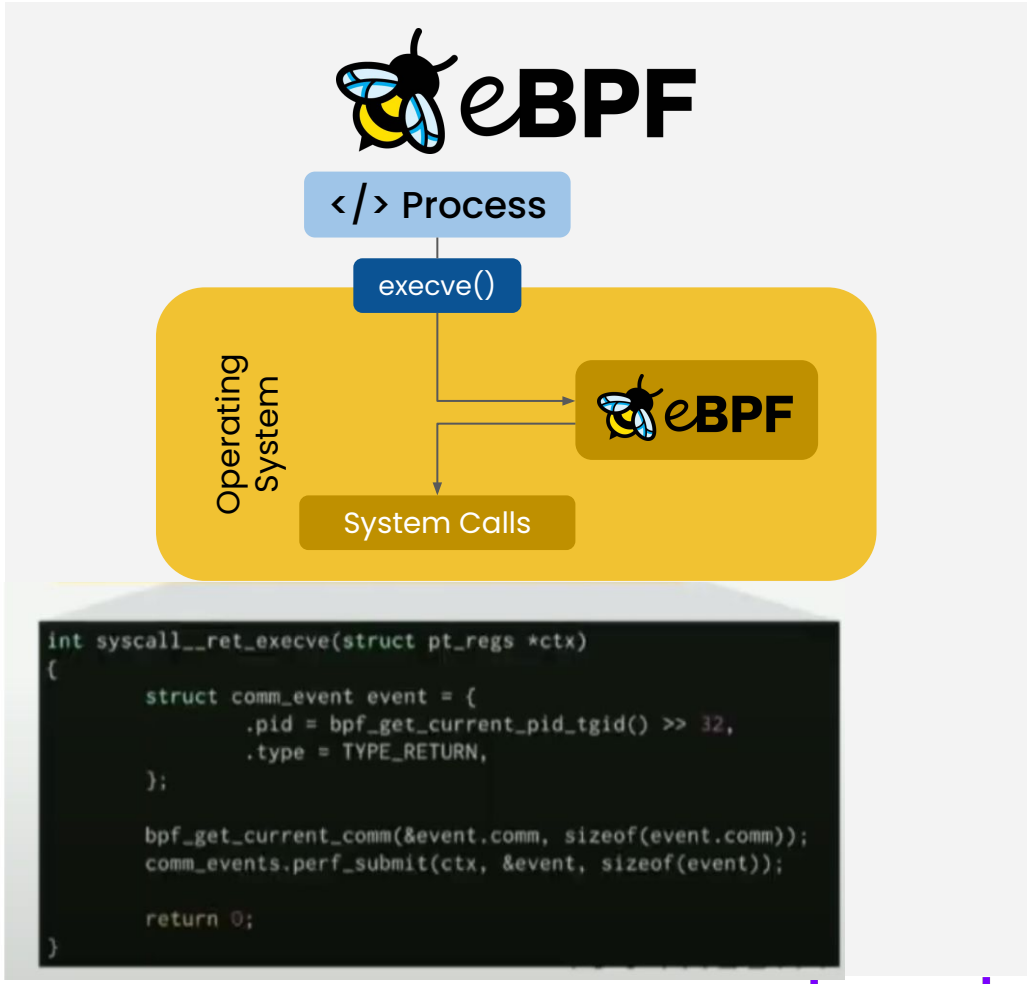
-Thomas Graf, CTO, Isovalent -

Reference:

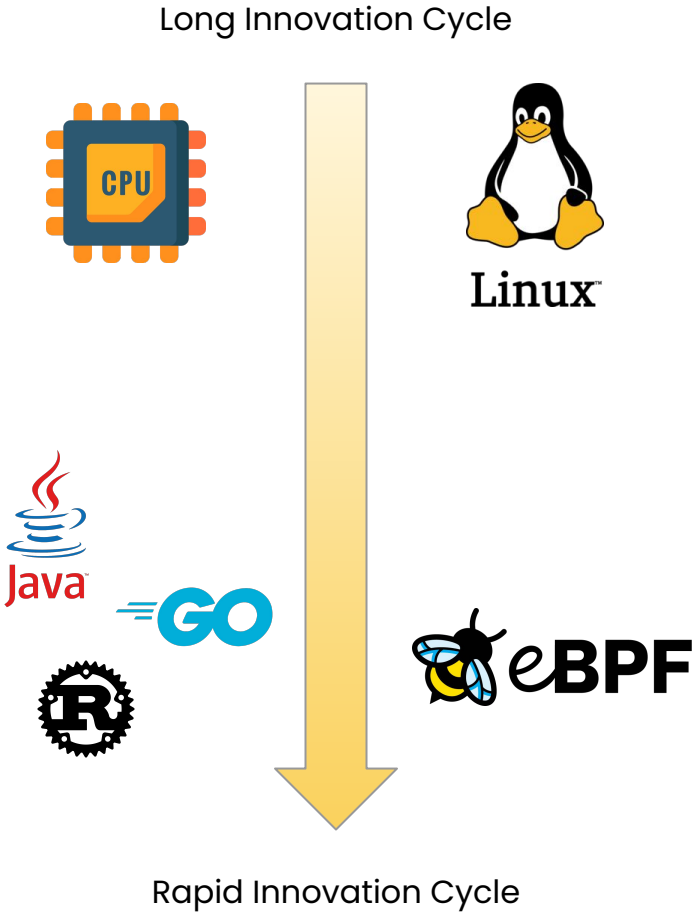
- Keynote: eBPF
- KubeCon2022 eBPF Day North America

Jumpbox®


Website
Submit
function hello() {
alert('hello')
}
</script>
<form onsubmit="return false;">
<input type="submit"
name="hello"
onclick="hello()">
</form>


eBPF
</> Process
execve()
Operating System
eBPF
System Calls
int syscall__ret_execve(struct pt_regs *ctx)
{
struct comm_event event = {
.pid = bpf_get_current_pid_tgid() >> 32,
.type = TYPE_RETURN,
};
bpf_get_current_comm(&event.comm, sizeof(event.comm));
comm_events.perf_submit(ctx, &event, sizeof(event));
return 0;
}

Operating Systems are like hardware,
hard to change and with long innovation cycles



Long innovation cycles result in the need to predict use cases or stick to providing building blocks

Programmability allows you to continuously adapt to changing requirements and innovate quickly

Jumpbox®

How is  eBPF different from  or  ?

- Designed specifically to be embedded into operating systems (Linux, Windows)
- Can interface with internal operating system APIs not available to user space
- Restricted to run safely in kernel context
- Aimed at kernel developers, hard to learn

- Designed to be embedded into applications
- General Purpose
- Aimed at application developers, easy to learn
- … **but** cannot run in an operating system context

How does  work?

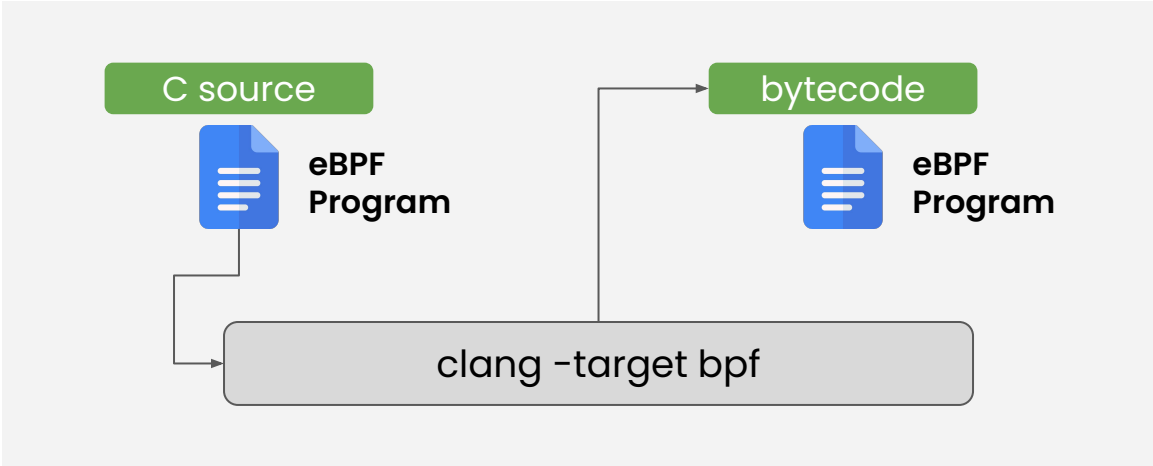
eBPF Language

+

eBPF Runtime

Jumpbox®

eBPF Language



cilium/ebpf

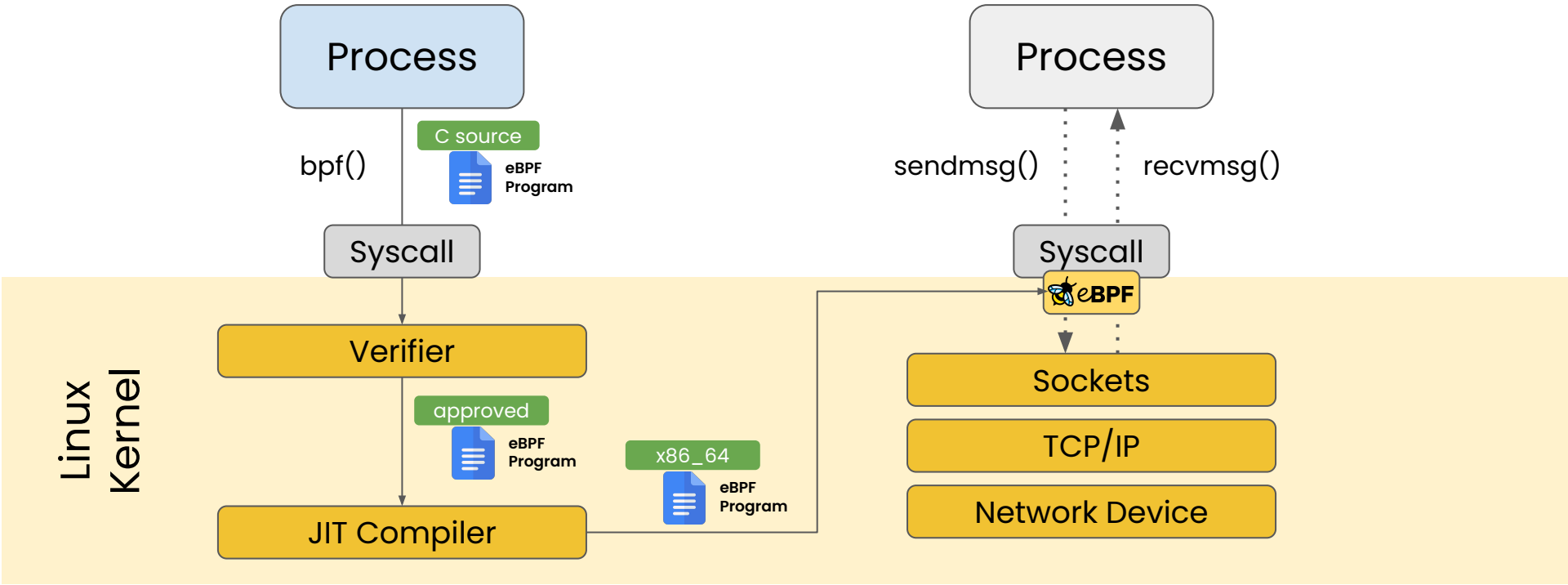
libbpf-rs | redbpf | Aya
Rust

libbpf
C++

bcc

Multiple SDKs and Compilers exist to get to eBPF bytecode

eBPF Runtime

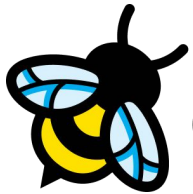


The runtime accepts bytecode, verifies it, just-in-time compiles it, and runs it at the requested hook point.

Jumpbox®

secure	efficient	portable
Runtime Verification Sandbox Concept Program Signing	JIT Compiler Embedded in OS Per-CPU data Structures	Generic Bytecode Data Type Discovery Stable API to OS

Jumpbox®

Who controls  eBPF ?

eBPF Foundation

Platinum

Meta

Google

ISOVALENT

Microsoft

TIGERA

HUAWEI

CROWDSTRIKE

Red Hat

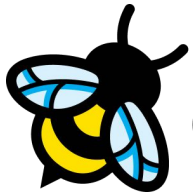
Silver

NETFLIX

DaoCloud

数悦铭金 DATOMS

FUTUREWEI

Where is  eBPF used today?

Cloud Native eBPF Landscape

Application Observability

- Pixie (CLOUD NATIVE COMPUTING FOUNDATION)
- bcc
- Inspektor Gadget
- Kubectl trace

Networking & Service Mesh

- Cilium (CLOUD NATIVE COMPUTING FOUNDATION)
- Calico

Security

- Tetragon (CLOUD NATIVE COMPUTING FOUNDATION)
- Tracee
- Falco (CLOUD NATIVE COMPUTING FOUNDATION)

Meta

https://github.com/facebookincubator/katran

Blazing fast, eBPF-based L4 load-balancer used at Facebook/Meta

All major providers have picked eBPF-based Networking & Security for their Kubernetes platforms

aws

Azure

Google Cloud

Smartphones

Android BPF Loader

During Android boot, all eBPF programs located at /system/etc/bpf/ are loaded. There programs are binary objects built by the Android build system from C programs and are accompanied by Android.bp files in the Android source tree. The build system stores the generated objects at /system/etc/bpf/ , and those objects become part of the system image.

Examples of eBPF in Andriod

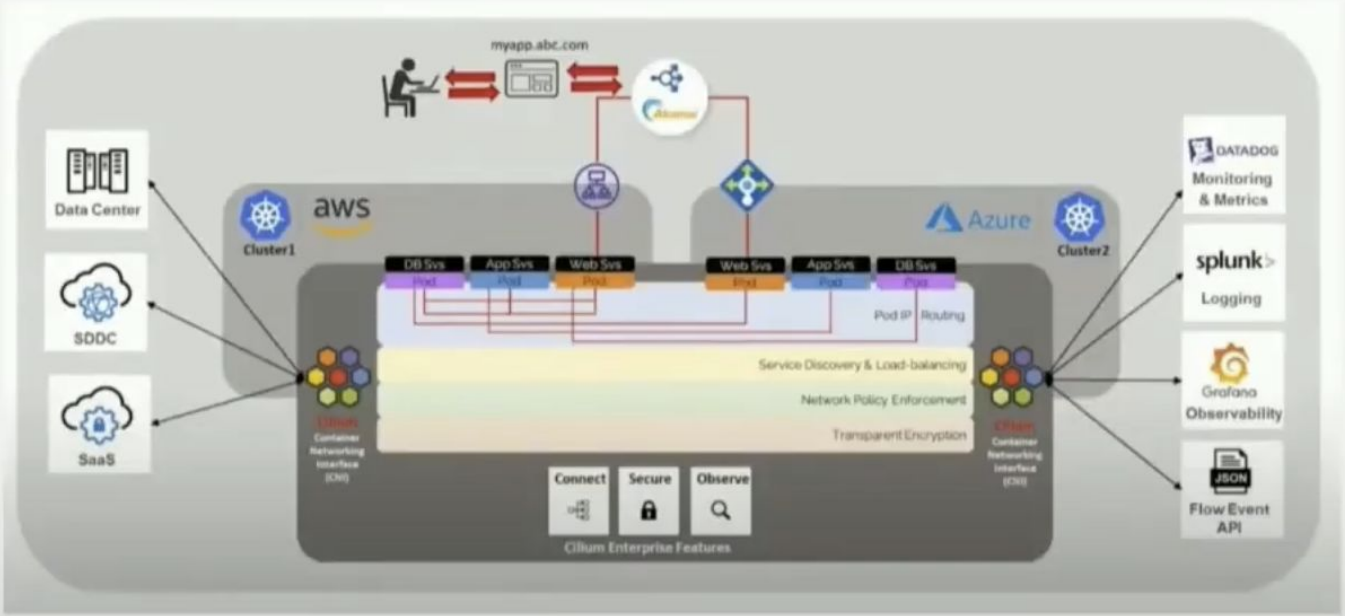
The following programs in AOSP provide additional examples of using eBPF:

- The netd eBPF C Program is used by the networking daemon (netd) in Android for various such as socket filtering and statistics gathering. To see how this program is used, check the eBPF traffic monitor sources.
- The time_in_state eBPF C program calculates the amount of time an Android app spends at different CPU frequencies, which is used to calculate power.
- In Android 12, the gpu_mem eBPF C program tracks total GPU memory usage for each process and for the entire system. The program is used for GPU memory profiling.

https://source.android.com/docs/core/architecture/kernel/bpf

Enterprises
Multi-Cloud Networking

S&P Global



[eBPF, a road to invisible network: S&P Global's Network Transformation Journey](#)
eBPF Summit 2022

ISOVALENT

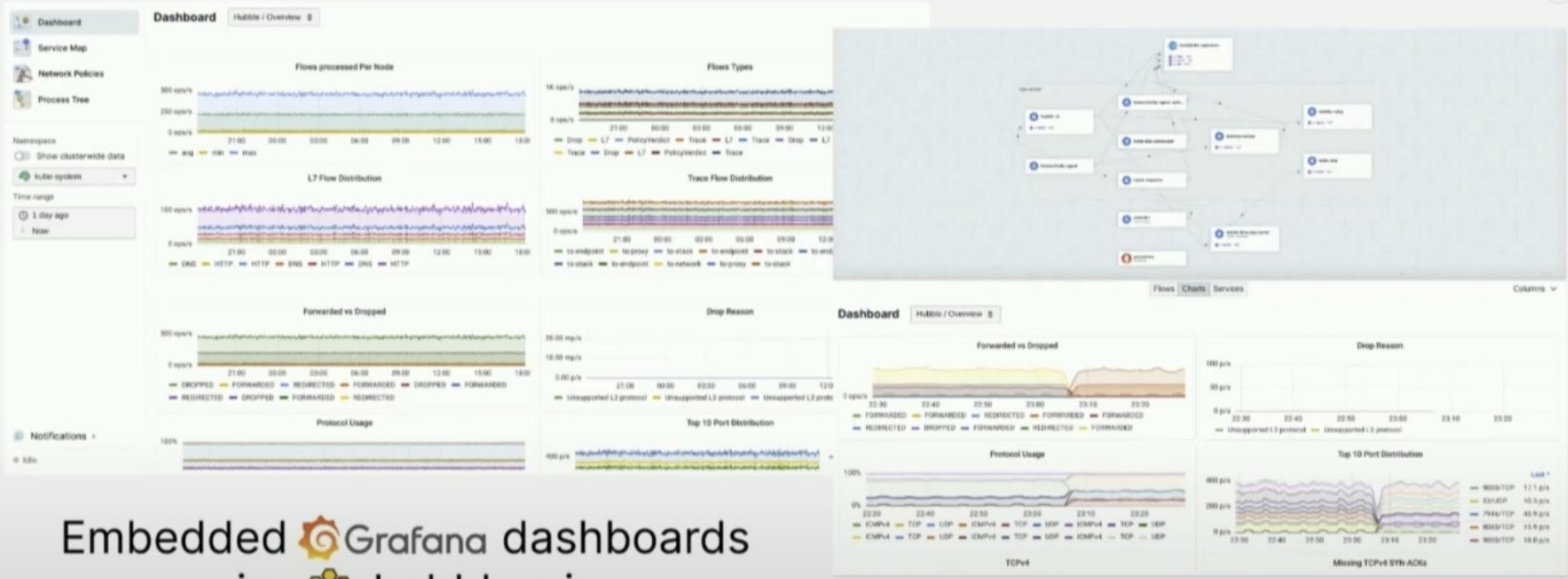
Jumpbox®

One more thing...

Jumpbox®

New Strategic Partnership to provide
eBPF -based Observability & Monitoring

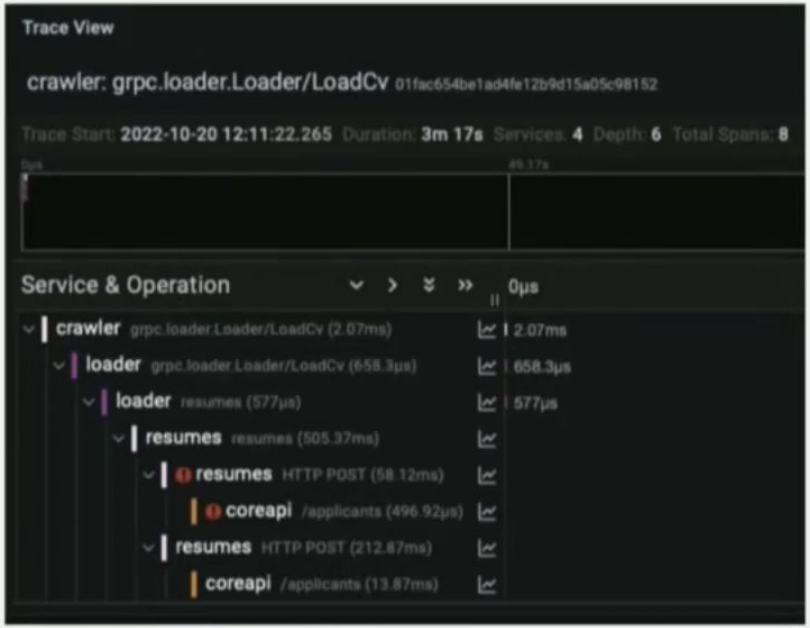
eBPF-based Network Observability



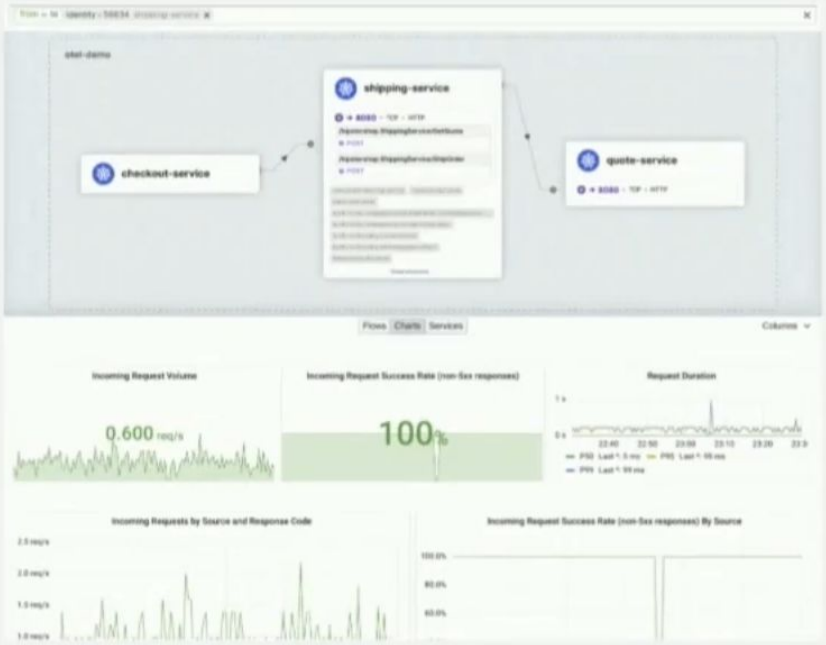
Embedded Grafana dashboards in hubble-ui

cilium

eBPF-based Tracing & HTTP Observability



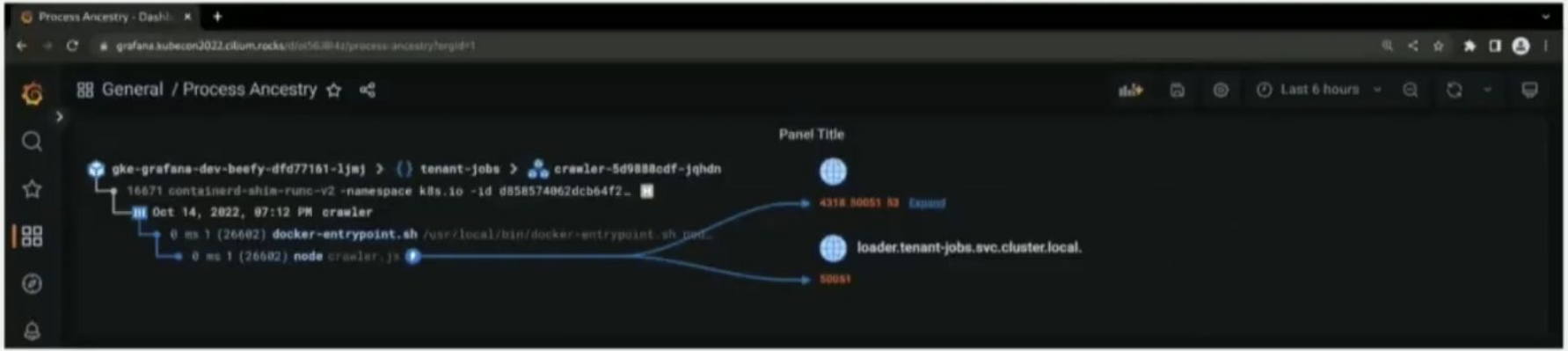
Grafana Tempo

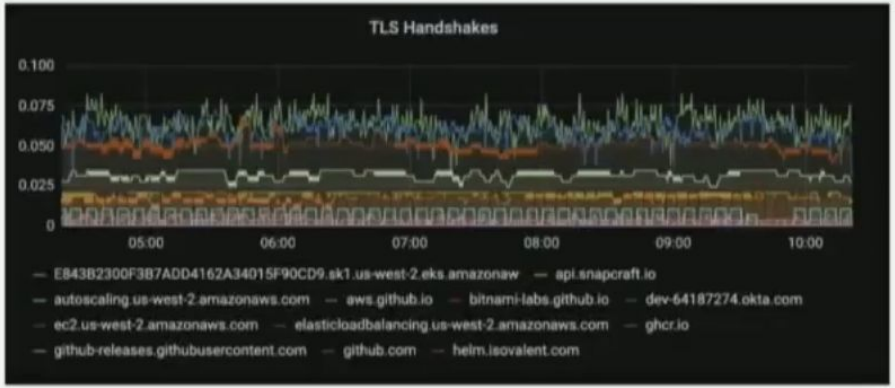


Golden Signal Dashboards

Jumpbox®

eBPF -based Security Observability





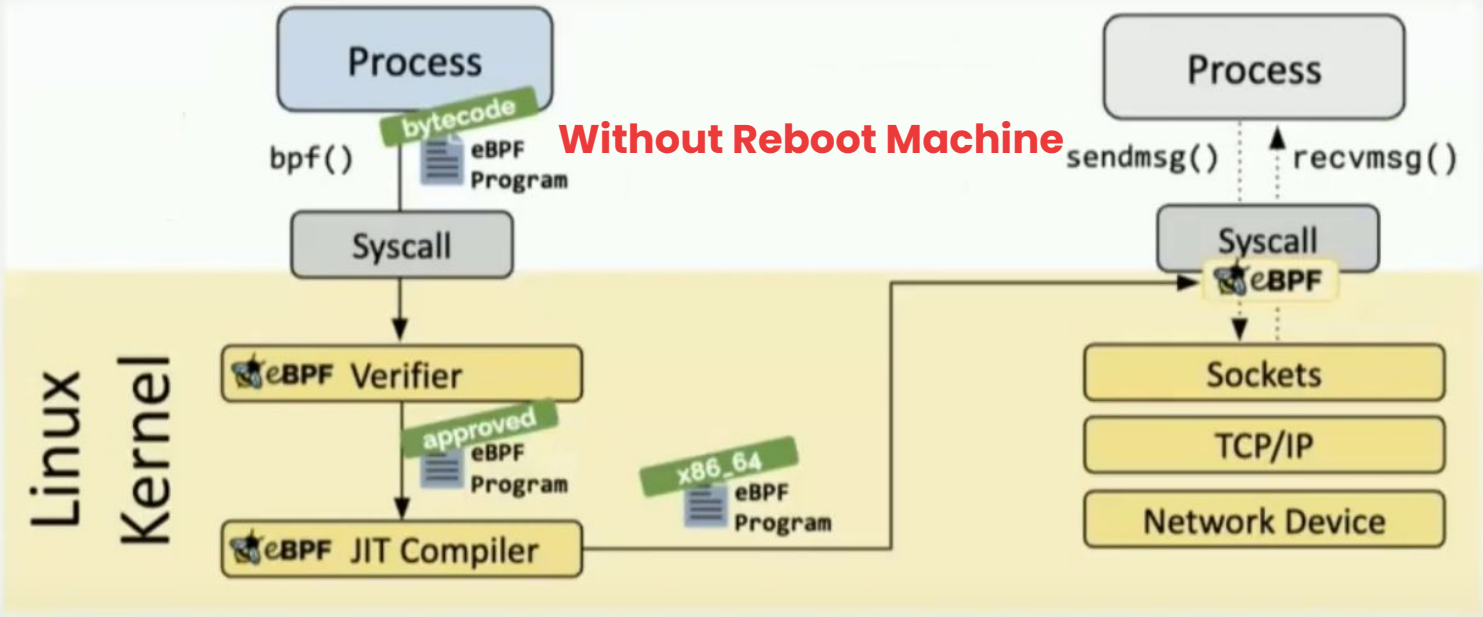
Run custom program of your choice

Jumpbox®

Run custom program of your choice
in the Kernel

Jumpbox®

eBPF Runtime



The runtime accepts bytecode, verifies it, just-in-time compiles it, and runs it at the requested hook point.

Let’s get started with BPF

Jumpbox®

Before we go

Jumpbox®

Before Let’s talk about

Before Let's talk about

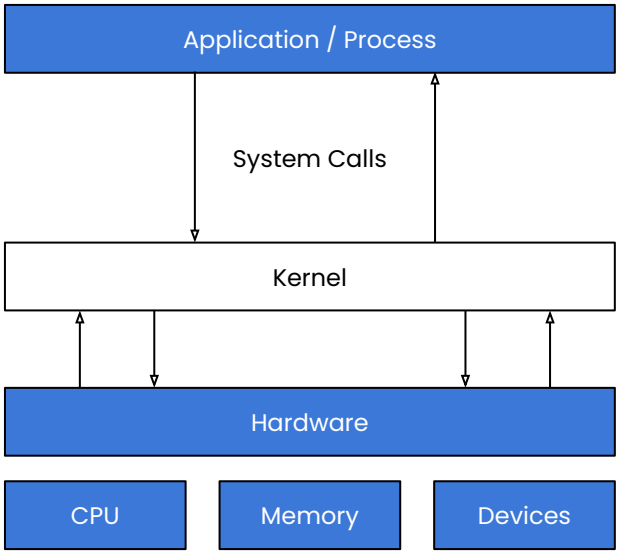
User Space and Kernel Space

Linux Kernel (Cont.)

- Kernel Space and User Space

Kernel Space

- Device Drivers
- Linux Network
- Scheduler



User Space

- Open a file
- Write to a file
- List Processes
- Defining a variable

Today,
We will focus on this area

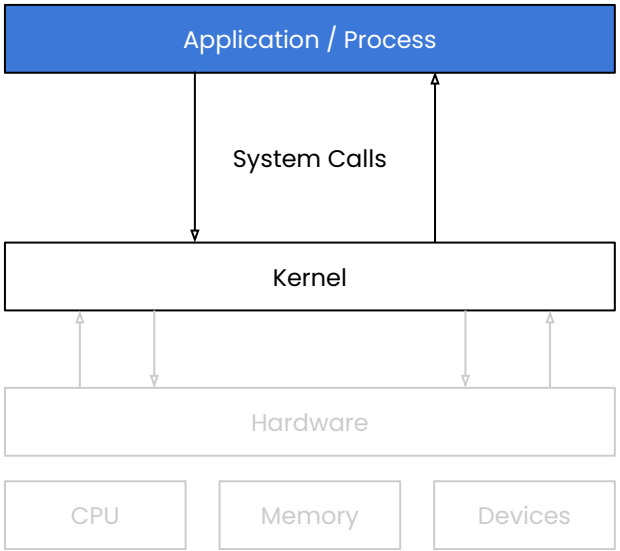
Jumpbox®

Linux Kernel (Cont.)

- Kernel Space and User Space

Kernel Space

- Device Drivers
- Linux Network
- Scheduler



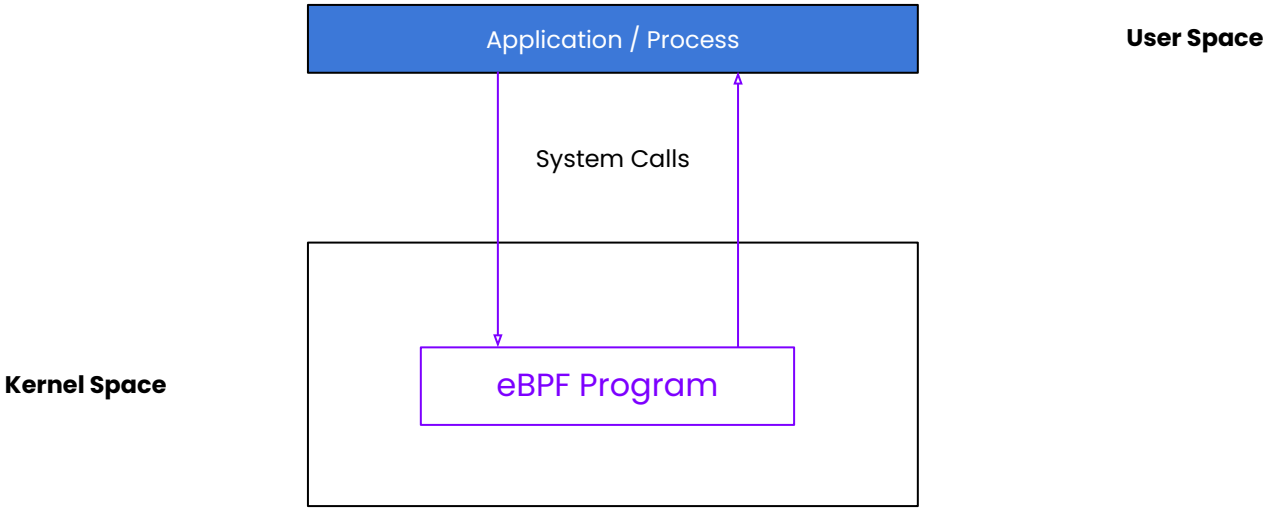
User Space

- Open a file
- Write to a file
- List Processes
- Defining a variable

Jumpbox®

Map to eBPF

eBPF with logical structure



Jumpbox®

What is BPF? (Cont.)

- The bpf() system call performs a range of operations related to extended Berkeley Packet Filters. Extended BPF (eBPF) is similar to the original (“classic”) BPF (cBPF) used to filter network packets.

- For both cBPF and eBPF programs, the kernel statically analyzes the programs (eBPF verifier) before loading them in order to ensure that they cannot harm the running system.

bpftrace

bpftrace is a high-level tracing language for Linux enhanced Berkeley Packet Filter (eBPF) available in recent Linux kernels (4.x). bpftrace uses LLVM as a backend to compile scripts to BPF-bytecode and makes use of BCC for interacting with the Linux BPF system, as well as existing Linux tracing capabilities: kernel dynamic tracing (kprobes), user-level dynamic tracing (uprobes), and tracepoints. The bpftrace language is inspired by awk and C, and predecessor tracers such as DTrace and SystemTap. bpftrace was created by Alastair Robertson.

To learn more about bpftrace, see the Reference Guide and One-Liner Tutorial.

One-Liners

The following one-liners demonstrate different capabilities:

```
# Files opened by process
bpftrace -e 'tracepoint:syscalls:sys_enter_open { printf("%s %s\n", comm, str(args->filename)); }'

# Syscall count by program
bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'

# Read bytes by process:
bpftrace -e 'tracepoint:syscalls:sys_exit_read /args->ret/ { @[comm] = sum(args->ret); }'

# Read size distribution by process:
bpftrace -e 'tracepoint:syscalls:sys_exit_read { @[comm] = hist(args->ret); }'

# Show per-second syscall rates:
bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @ = count(); } interval:s:1 { print(@); clear(@);
```

Explore bpf syscalls in bpftrace

Jumpbox®

```
vagrant@vagrant: ~/libbpfgo-beginners (ssh vagrant@127.0.0.1 -p 2222 -o LogLevel=FATAL -o Compression=yes -o DSAAuthentication=yes -o IdentitiesOnly=yes -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null -i /Users/liz/vagrant/ubuntu-
vagrant@vagrant:~/libbpfgo-beginners$ sudo bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'
Attaching 1 probe...
^C

@[vmstats]: 2
@[memballoon]: 2
@[timesync]: 4
@[sudo]: 7
@[multipathd]: 22
@[true]: 29
@[cpptools-srv]: 35
@[irqbalance]: 38
@[sshd]: 44
@[gmain]: 44
@[bpftrace]: 47
@[gopls]: 65
@[docker-init]: 96
@[cpptools]: 97
@[dbus-daemon]: 97
@[vminfo]: 100
@[systemd-logind]: 108
@[containerd]: 220
@[(kubelet)]: 235
@[rs:main Q:Reg]: 252
@[grep]: 324
@[in:imuxsock]: 465
@[systemd-run]: 477
@[systemd-journal]: 710
@[sh]: 903
@[systemd]: 1052
@[node]: 1956
@[ls]: 2271
@[dockerd]: 2639
@[kubelet]: 2881
@[runc]: 4610
```

```
vagrant@vagrant:~/libbpfgo-beginners$ sudo strace -e bpf bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_ARRAY, key_size=4, value_size=4, max_entries=1, map_flags=0, inner_map_fd=0, map_name="
, map_ifindex=0, btf_fd=0, btf_key_type_id=0, btf_value_type_id=0}, 120) = 3
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERCPU_HASH, key_size=16, value_size=8, max_entries=4096, map_flags=0, inner_map_fd=0,
map_name="@", map_ifindex=0, btf_fd=0, btf_key_type_id=0, btf_value_type_id=0}, 120) = -1 EINVAL (Invalid argument)
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERCPU_HASH, key_size=16, value_size=8, max_entries=4096, map_flags=0, inner_map_fd=0,
map_name="", map_ifindex=0, btf_fd=0, btf_key_type_id=0, btf_value_type_id=0}, 120) = 3
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERF_EVENT_ARRAY, key_size=4, value_size=4, max_entries=2, map_flags=0, inner_map_fd=0,
 map_name="printf", map_ifindex=0, btf_fd=0, btf_key_type_id=0, btf_value_type_id=0}, 120) = 4
Attaching 1 probe...
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=4, key=0x7ffe26e88d4c, value=0x7ffe26e88d50, flags=BPF_ANY}, 120) = 0
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=4, key=0x7ffe26e88d4c, value=0x7ffe26e88d50, flags=BPF_ANY}, 120) = 0
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_TRACEPOINT, insn_cnt=27, insns=0x7efeb74ce000, license="GPL", log_level=0, log_size=0,
 log_buf=NULL, kern_version=KERNEL_VERSION(5, 8, 17), prog_flags=0, prog_name="sys_enter", prog_ifindex=0, expected_attach_type=BP
F_CGROUP_INET_INGRESS, prog_btf_fd=0, func_info_rec_size=0, func_info=NULL, func_info_cnt=0, line_info_rec_size=0, line_info=NULL,
 line_info_cnt=0, attach_btf_id=0, attach_prog_fd=0}, 120) = 9
^Cstrace: Process 1688560 detached

vagrant@vagrant:~/libbpfgo-beginners$

@[vmstats]: 2
@[memballoon]: 2
@[gmain]: 5
@[systemd-journal]: 5
@[sudo]: 7
@[multipathd]: 8
@[cpptools-srv]: 16
@[dockerd]: 24
@[containerd]: 28
@[bpftrace]: 28
@[gopls]: 35
@[irqbalance]: 38
@[cpptools]: 48
@[sshd]: 53
@[grep]: 108
@[strace]: 191
```

```
Attaching 1 probe ...
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=4, key=0x7ffe26e88d4c, value=0x7ffe26e88d50, flags=BPF_ANY}, 120) = 0
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=4, key=0x7ffe26e88d4c, value=0x7ffe26e88d50, flags=BPF_ANY}, 120) = 0
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_TRACEPOINT, insn_cnt=27, insns=0x7efeb74ce000, license="GPL", log_level=0, log_size=0,
 log_buf=NULL, kern_version=KERNEL_VERSION(5, 8, 17), prog_flags=0, prog_name="sys_enter", prog_ifindex=0, expected_attach_type=BP
F_CGROUP_INET_INGRESS, prog_btf_fd=0, func_info_rec_size=0, func_info=NULL, func_info_cnt=0, line_info_rec_size=0, line_info=NULL,
 line_info_cnt=0, attach_btf_id=0, attach_prog_fd=0}, 120) = 9
^Cstrace: Process 1688560 detached
```

```
Attaching 1 probe ...
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=4, key=0x7ffe26e88d4c, value=0x7ffe26e88d50, flags=BPF_ANY}, 120) = 0
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=4, key=0x7ffe26e88d4c, value=0x7ffe26e88d50, flags=BPF_ANY}, 120) = 0
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_TRACEPOINT, insn_cnt=27, insns=0x7efeb74ce000, license="GPL", log_level=0, log_size=0,
 log_buf=NULL, kern_version=KERNEL_VERSION(5, 8, 17), prog_flags=0, prog_name="sys_enter", prog_ifindex=0, expected_attach_type=BP
F_CGROUP_INET_INGRESS, prog_btf_fd=0, func_info_rec_size=0, func_info=NULL, func_info_cnt=0, line_info_rec_size=0, line_info=NULL,
 line_info_cnt=0, attach_btf_id=0, attach_prog_fd=0}, 120) = 9
^Cstrace: Process 1688560 detached
```

eBPF programs & maps

```
bpf(BPF_PROG_LOAD, …)
bpf(BPF_MAP_CREATE, …)
```

eBPF Programs (Cont.)

- eBPF programs can be written in a restricted C that is compiled (using the clang compiler) into eBPF bytecode.

- Various features are omitted from this restricted C, such as loops, global variables, variadic functions, floating-point numbers, and passing structures as function arguments.

Reference:
- restricted C

eBPF Helper function (Cont.)

[eBPF Helper functions] are used by eBPF programs to interact with the system, or with the context in which they work.

For instance, they can be used to print debugging messages:

```
bpf_trace_printk()

bpf_get_current_comm()

bpf_perf_event_output()

...
```

Reference:

- eBPF Helper function

Jumpbox®

eBPF Map (Cont.)

Maps are a generic data structure for the storage of different types of data.

They allow **sharing of data** between eBPF kernel programs and also between **kernel and user-space applications**.

eBPF Map

Reference:

- eBPF Helper function

Attaching eBPF to events

eBPF programs are event-driven and are run when the kernel or an application passes a certain hook point.

Pre-defined hooks include system calls, function entry/exit, kernel tracepoints, network events, and several others.

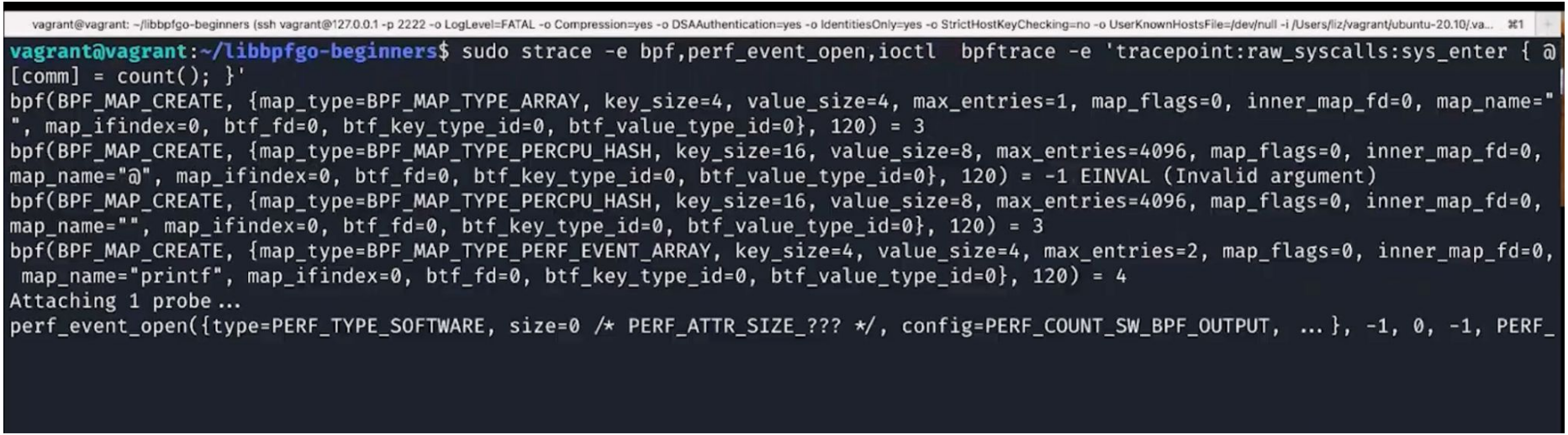
If a predefined hook does not exist for a particular need, it is possible to create a kernel probe (kprobe) or user probe (uprobe) to attach eBPF program almost anywhere in kernel or user applications.

Reference:

- Program type BPF_PROG_TYPE_KPROBE

Attach custom code to an event

```
bpf(BPF_PROG_LOAD, …) = x
perf_event_open(…) = y
ioctl(y, PERF_EVENT_IOC_SET_BPF, x)
```


vagrant@vagrant: ~/libbpfgo-beginners (ssh vagrant@127.0.0.1 -p 2222 -o LogLevel=FATAL -o Compression=yes -o DSAAuthentication=yes -o IdentitiesOnly=yes -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null -i /Users/liz/vagrant/ubuntu-20.10/.va... ⌘1
vagrant@vagrant:~/libbpfgo-beginners$ sudo strace -e bpf,perf_event_open,ioctl bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_ARRAY, key_size=4, value_size=4, max_entries=1, map_flags=0, inner_map_fd=0, map_name="", map_ifindex=0, btf_fd=0, btf_key_type_id=0, btf_value_type_id=0}, 120) = 3
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERCPU_HASH, key_size=16, value_size=8, max_entries=4096, map_flags=0, inner_map_fd=0, map_name="@", map_ifindex=0, btf_fd=0, btf_key_type_id=0, btf_value_type_id=0}, 120) = -1 EINVAL (Invalid argument)
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERCPU_HASH, key_size=16, value_size=8, max_entries=4096, map_flags=0, inner_map_fd=0, map_name="", map_ifindex=0, btf_fd=0, btf_key_type_id=0, btf_value_type_id=0}, 120) = 3
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERF_EVENT_ARRAY, key_size=4, value_size=4, max_entries=2, map_flags=0, inner_map_fd=0, map_name="printf", map_ifindex=0, btf_fd=0, btf_key_type_id=0, btf_value_type_id=0}, 120) = 4
Attaching 1 probe...
perf_event_open({type=PERF_TYPE_SOFTWARE, size=0 /* PERF_ATTR_SIZE_??? */, config=PERF_COUNT_SW_BPF_OUTPUT, ...}, -1, 0, -1, PERF_

```
 map_name="printf", map_ifindex=0, btf_fd=0, btf_key_type_id=0, btf_value_type_id=0}, 120) = 4
Attaching 1 probe...
perf_event_open({type=PERF_TYPE_SOFTWARE, size=0 /* PERF_ATTR_SIZE_??? */, config=PERF_COUNT_SW_BPF_OUTPUT, ...}, -1, 0, -1, PER
FLAG_FD_CLOEXEC) = 6
ioctl(6, PERF_EVENT_IOC_ENABLE, 0)      = 0
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=4, key=0x7ffdf9ccbc0c, value=0x7ffdf9ccbc10, flags=BPF_ANY}, 120) = 0
perf_event_open({type=PERF_TYPE_SOFTWARE, size=0 /* PERF_ATTR_SIZE_??? */, config=PERF_COUNT_SW_BPF_OUTPUT, ...}, -1, 1, -1, PER
FLAG_FD_CLOEXEC) = 7
ioctl(7, PERF_EVENT_IOC_ENABLE, 0)      = 0
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=4, key=0x7ffdf9ccbc0c, value=0x7ffdf9ccbc10, flags=BPF_ANY}, 120) = 0
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_TRACEPOINT, insn_cnt=27, insns=0x7f88a6521000, license="GPL", log_level=0, log_size=
 log_buf=NULL, kern_version=KERNEL_VERSION(5, 8, 17), prog_flags=0, prog_name="sys_enter", prog_ifindex=0, expected_attach_type=
F_CGROUP_INET_INGRESS, prog_btf_fd=0, func_info_rec_size=0, func_info=NULL, func_info_cnt=0, line_info_rec_size=0, line_info=NUL
 line_info_cnt=0, attach_btf_id=0, attach_prog_fd=0}, 120) = 9
perf_event_open({type=PERF_TYPE_TRACEPOINT, size=0 /* PERF_ATTR_SIZE_??? */, config=22, ...}, -1, 0, -1, PERF_FLAG_FD_CLOEXEC) =
ioctl(8, PERF_EVENT_IOC_SET_BPF, 9)     = 0
ioctl(8, PERF_EVENT_IOC_ENABLE, 0)      = 0
^Cstrace: Process 1690732 detached

vagrant@vagrant:~/libbpfgo-beginners$

@[multipathd]: 4
@[systemd-journal]: 5
@[sudo]: 7
@[cpptools-srv]: 10
@[gopls]: 11
@[bpftrace]: 16
@[containerd]: 20
```

```
 map_name="printf", map_ifindex=0, btf_fd=0, btf_key_type_id=0, btf_value_type_id=0}, 120) = 4
Attaching 1 probe...
perf_event_open({type=PERF_TYPE_SOFTWARE, size=0 /* PERF_ATTR_SIZE_??? */, config=PERF_COUNT_SW_BPF_OUTPUT, ...}, -1, 0, -1, PERF_
FLAG_FD_CLOEXEC) = 6
ioctl(6, PERF_EVENT_IOC_ENABLE, 0)      = 0
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=4, key=0x7ffdf9ccbc0c, value=0x7ffdf9ccbc10, flags=BPF_ANY}, 120) = 0
perf_event_open({type=PERF_TYPE_SOFTWARE, size=0 /* PERF_ATTR_SIZE_??? */, config=PERF_COUNT_SW_BPF_OUTPUT, ...}, -1, 1, -1, PERF_
FLAG_FD_CLOEXEC) = 7
ioctl(7, PERF_EVENT_IOC_ENABLE, 0)      = 0
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=4, key=0x7ffdf9ccbc0c, value=0x7ffdf9ccbc10, flags=BPF_ANY}, 120) = 0
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_TRACEPOINT, insn_cnt=27, insns=0x7f88a6521000, license="GPL", log_level=0, log_size=0,
 log_buf=NULL, kern_version=KERNEL_VERSION(5, 8, 17), prog_flags=0, prog_name="sys_enter", prog_ifindex=0, expected_attach_type=BP
F_CGROUP_INET_INGRESS, prog_btf_fd=0, func_info_rec_size=0, func_info=NULL, func_info_cnt=0, line_info_rec_size=0, line_info=NULL,
 line_info_cnt=0, attach_btf_id=0, attach_prog_fd=0}, 120) = 9
perf_event_open({type=PERF_TYPE_TRACEPOINT, size=0 /* PERF_ATTR_SIZE_??? */, config=22, ...}, -1, 0, -1, PERF_FLAG_FD_CLOEXEC) = 8
ioctl(8, PERF_EVENT_IOC_SET_BPF, 9)     = 0
ioctl(8, PERF_EVENT_IOC_ENABLE, 0)      = 0
^Cstrace: Process 1690732 detached
```

```
ioctl(7, PERF_EVENT_IOC_ENABLE, 0)      = 0
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=4, key=0x7ffdf9ccbc0c, value=0x7ffdf9ccbc10, flags=BPF_ANY}, 120) = 0
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_TRACEPOINT, insn_cnt=27, insns=0x7f88a6521000, license="GPL", log_level=0, log_size=0,
 log_buf=NULL, kern_version=KERNEL_VERSION(5, 8, 17), prog_flags=0, prog_name="sys_enter", prog_ifindex=0, expected_attach_type=BP
F_CGROUP_INET_INGRESS, prog_btf_fd=0, func_info_rec_size=0, func_info=NULL, func_info_cnt=0, line_info_rec_size=0, line_info=NULL,
 line_info_cnt=0, attach_btf_id=0, attach_prog_fd=0}, 120) = 9
perf_event_open({type=PERF_TYPE_TRACEPOINT, size=0 /* PERF_ATTR_SIZE_??? */, config=22, ...}, -1, 0, -1, PERF_FLAG_FD_CLOEXEC) = 8
ioctl(8, PERF_EVENT_IOC_SET_BPF, 9)     = 0
ioctl(8, PERF_EVENT_IOC_ENABLE, 0)      = 0
^Cstrace: Process 1690732 detached
```

How to write eBPF hello world?

Jumpbox®

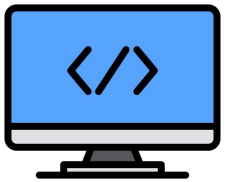
Back to Basics

Jumpbox®

Back to Basics
with Application Building

Jumpbox®

Application Building (Cont.)



Source Code

Code Translation

(Compile / Interpret / Transpilers / JIT / AOT)

Executable file

(Application)

- OS Executable
- Need Application Runtime:
  → Application Package

Jumpbox®

eBPF with logical structure flow (Cont.)

User Space

eBPF with logical structure flow (Cont.)

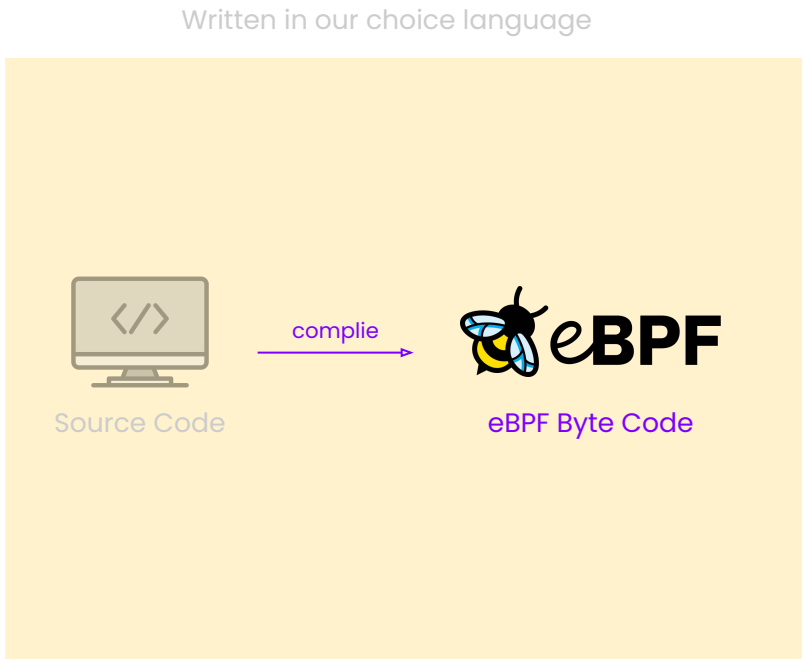
Written in our choice language

Source Code

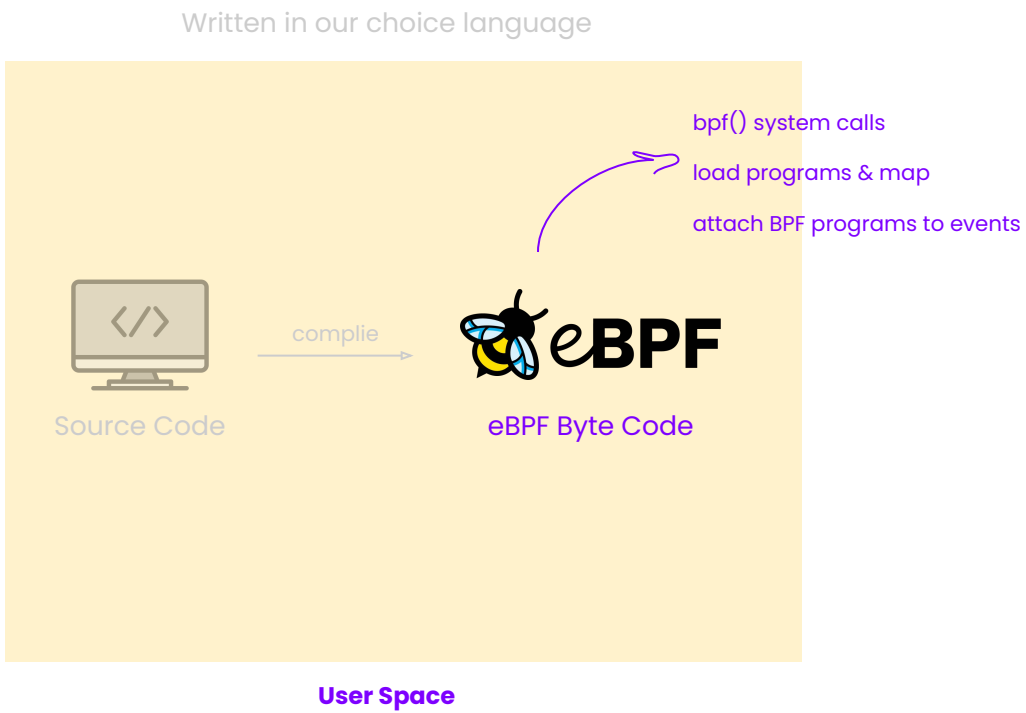
User Space

Jumpbox®

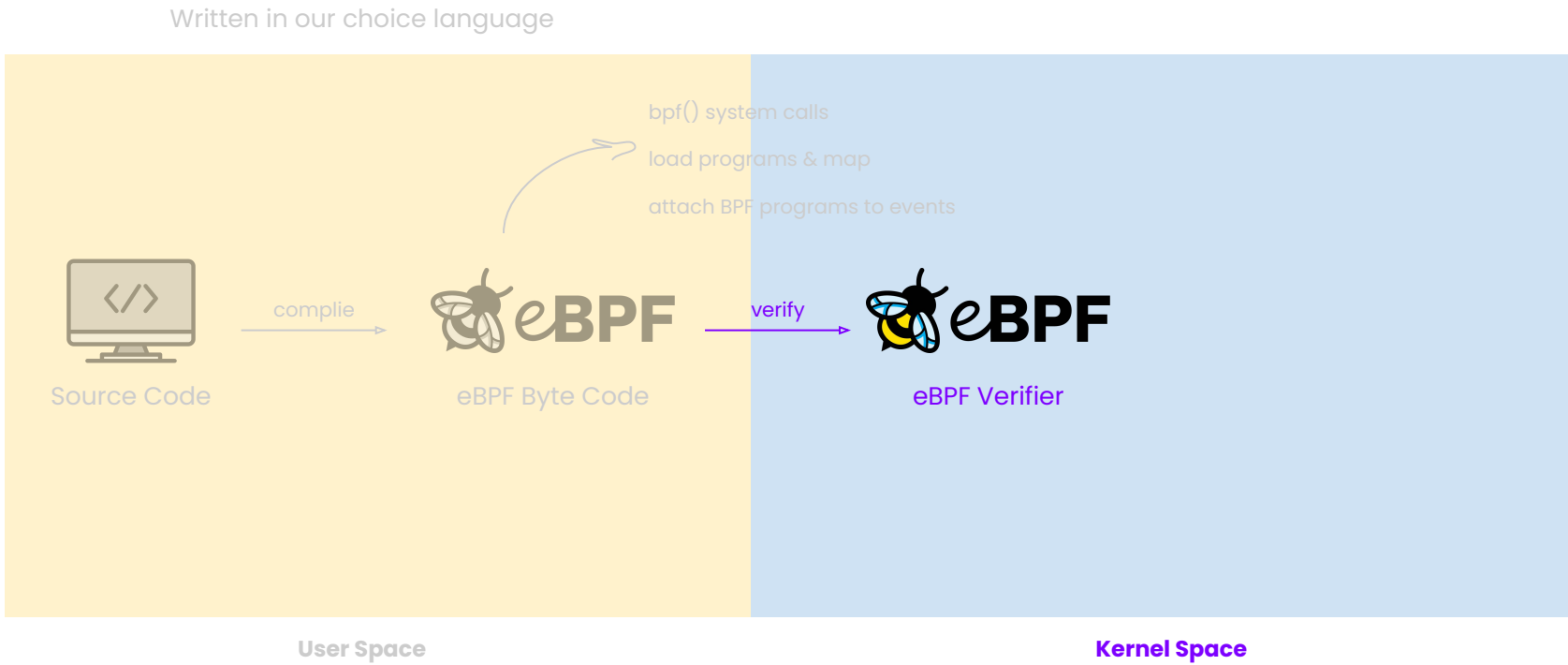
eBPF with logical structure flow (Cont.)



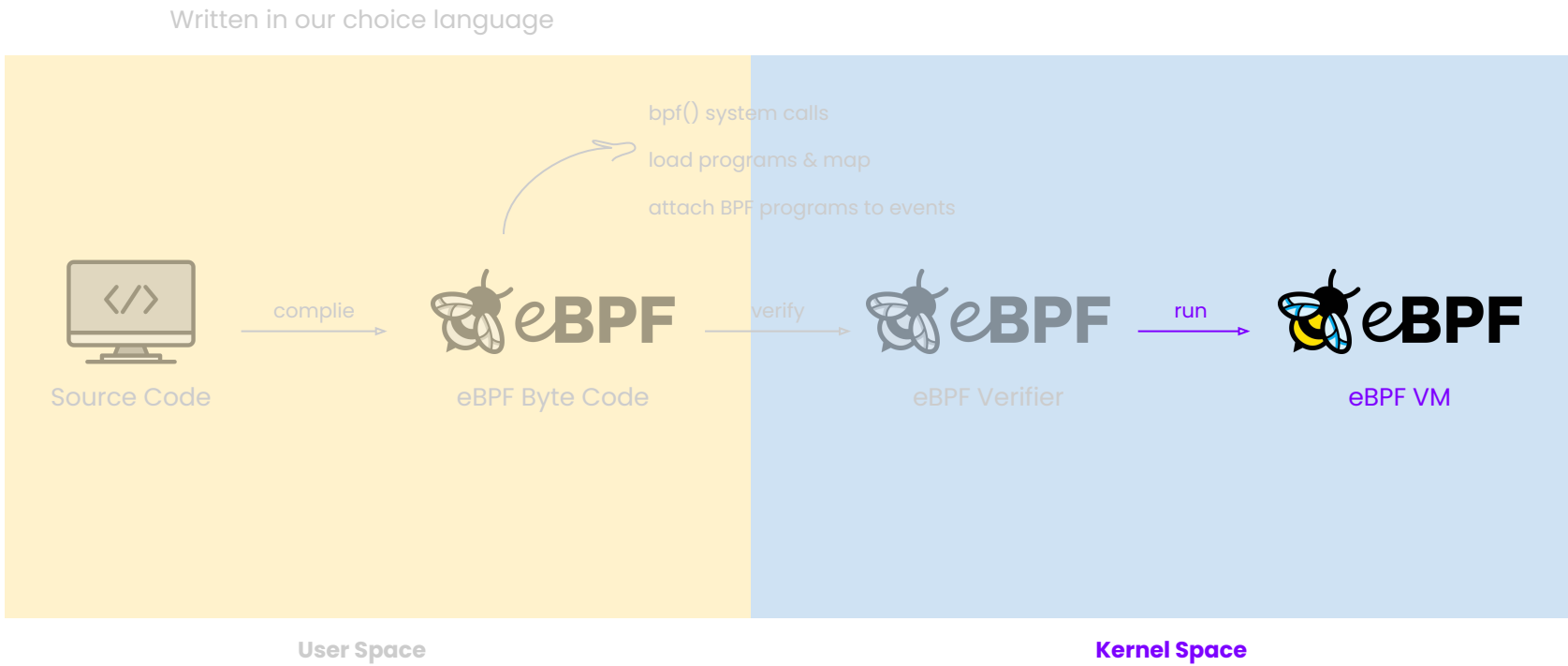
eBPF with logical structure flow (Cont.)



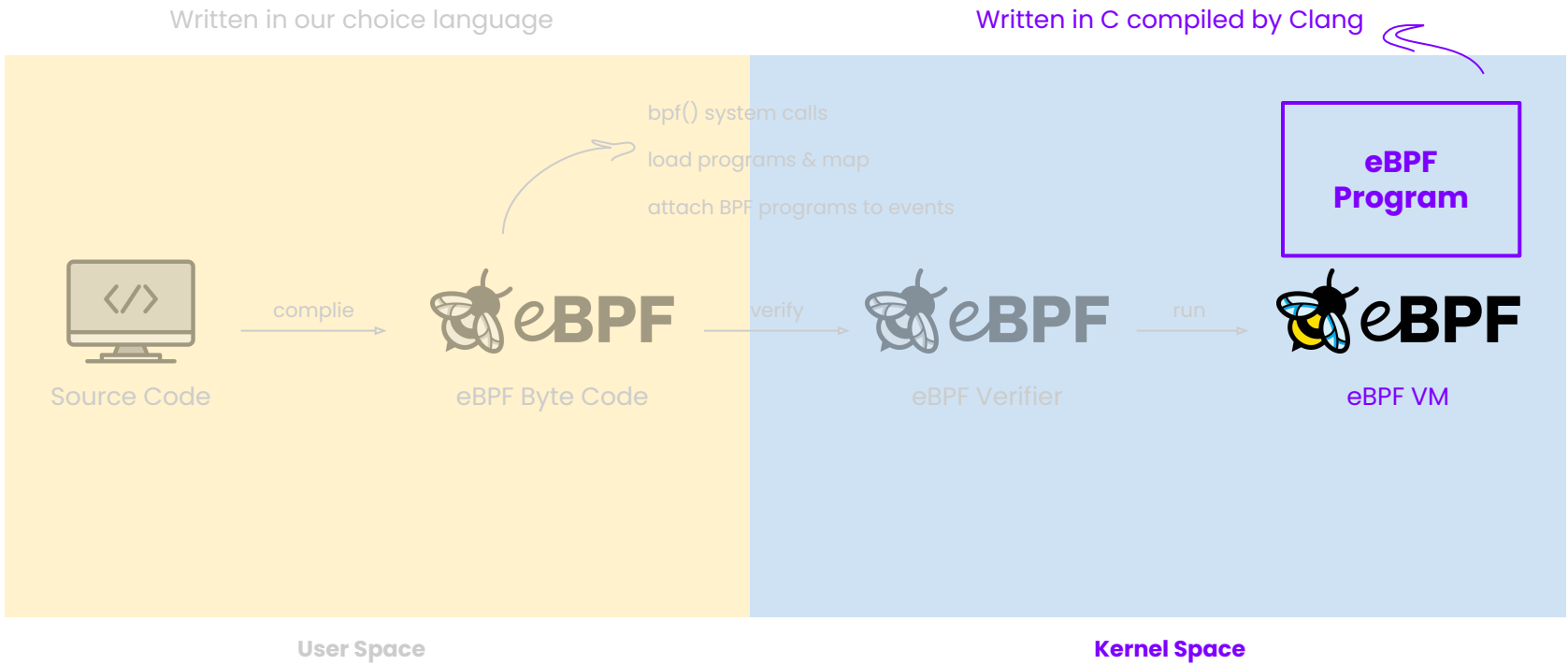
eBPF with logical structure flow (Cont.)



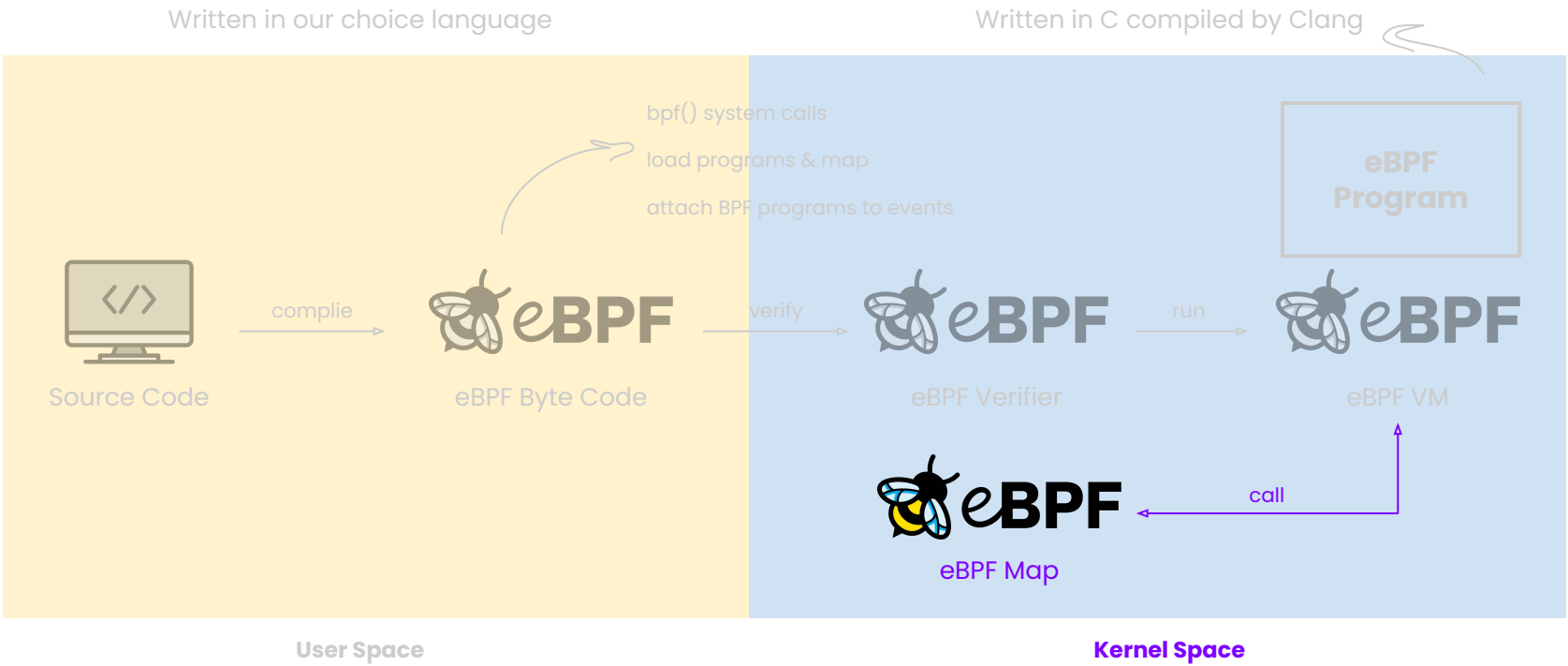
eBPF with logical structure flow (Cont.)



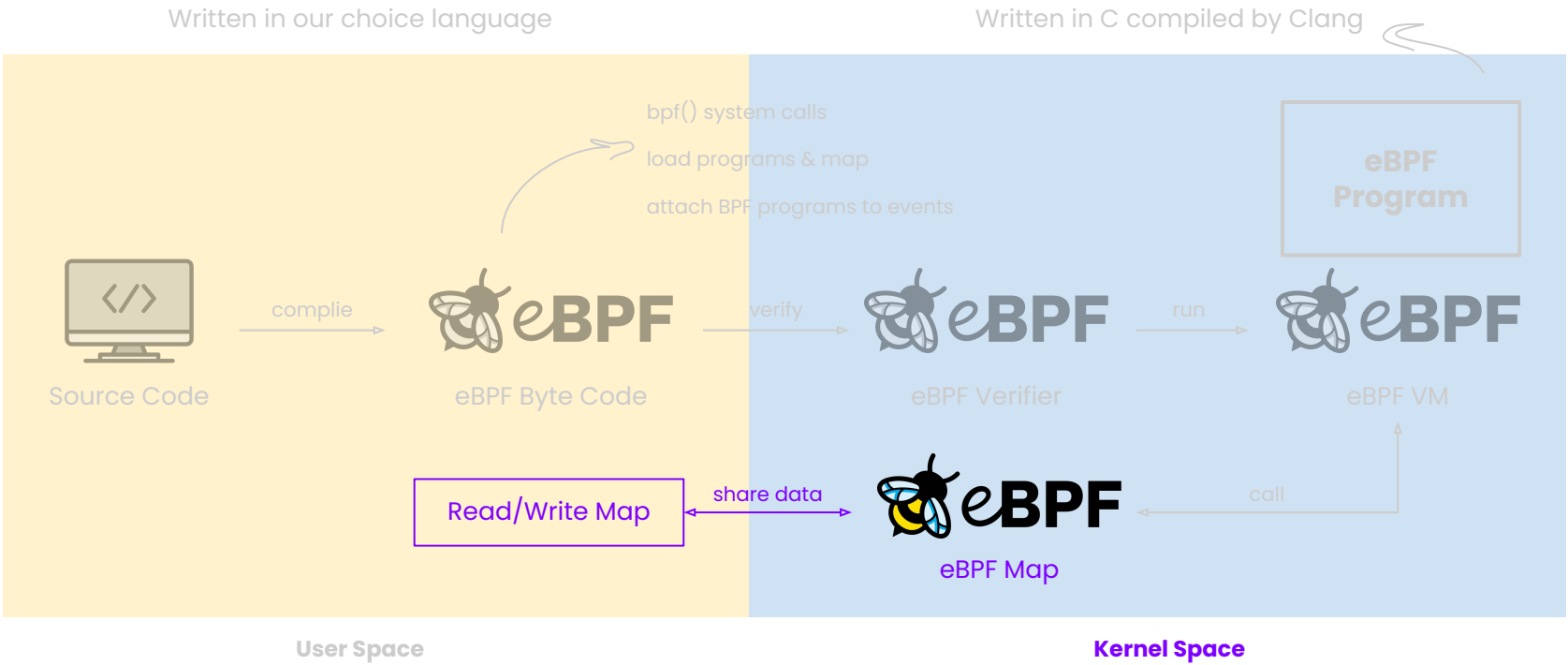
eBPF with logical structure flow (Cont.)



eBPF with logical structure flow (Cont.)

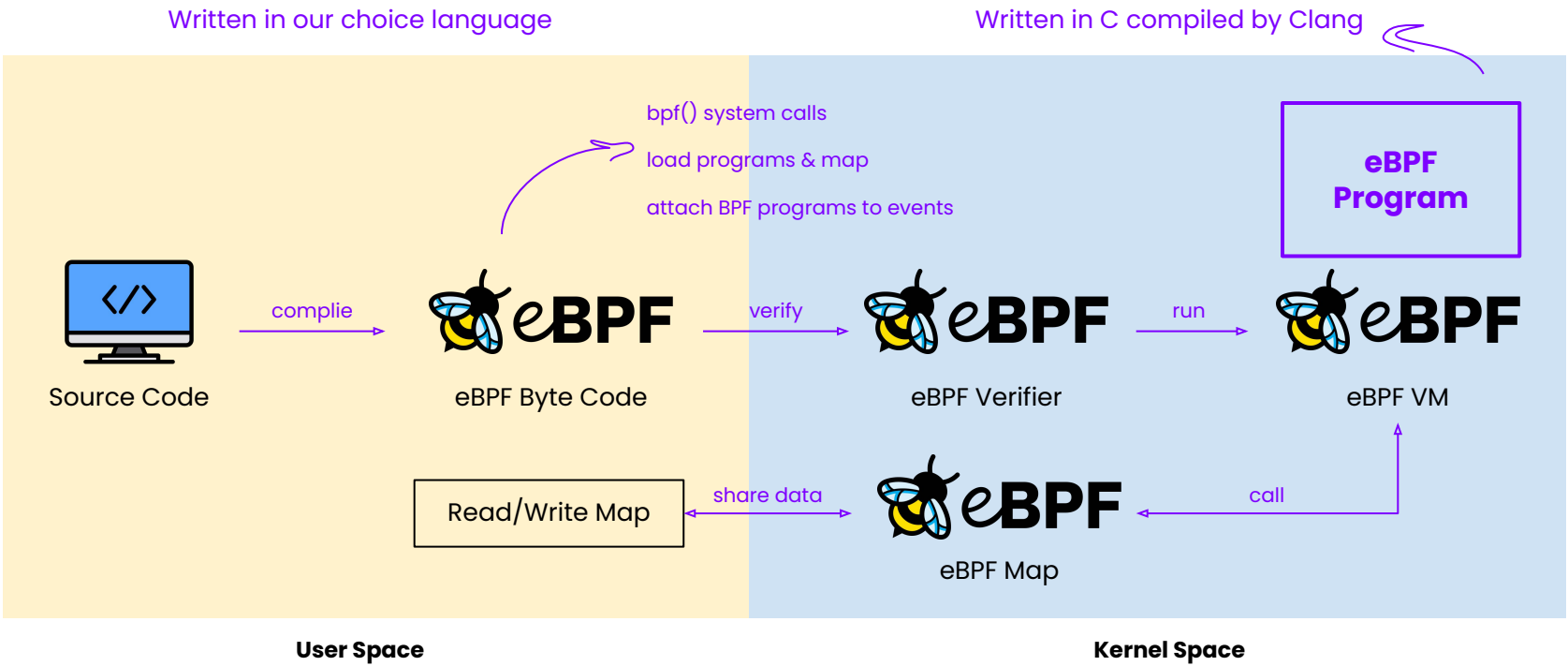


eBPF with logical structure flow (Cont.)

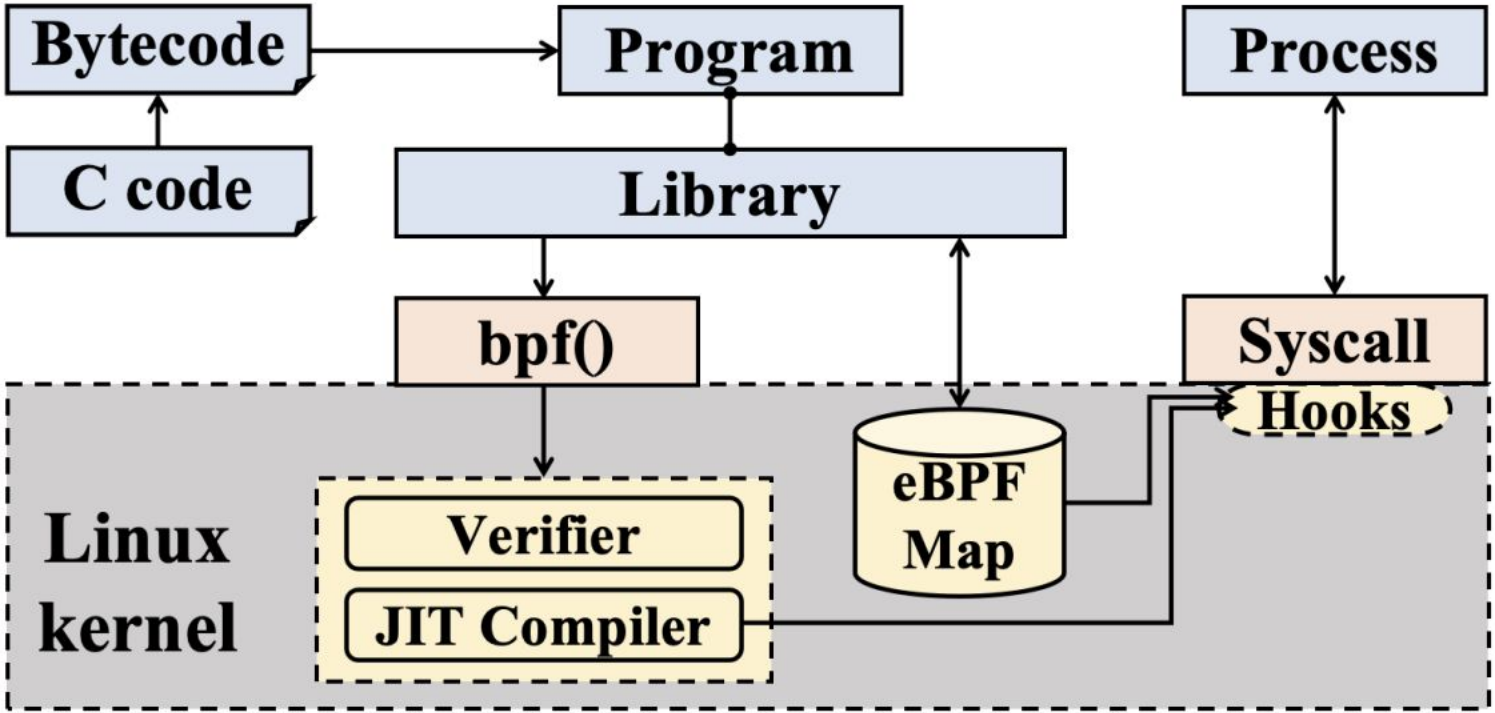


Jumpbox®

eBPF with logical structure flow (Cont.)



eBPF with logical structure flow (Cont.)



Reference:

- Full Logical Structure

Review Keywords

- tracepoint: Trace Syscalls
- uprobe: Userspace Probe
- kprobe: Kernel Function Probe
- xdp: Packet Filtering
- lsm: Linux Security Module

Can MacOS users develop eBPF?

Jumpbox®

[🌎Web site] [📖Documentation] [👤Slack (`#lima`)]

Lima: Linux Machines

openssf best practices passing

Lima launches Linux virtual machines with automatic file sharing and port forwarding (similar to WSL2).

The original goal of Lima was to promote containerd including nerdctl (contaiNERD ctl) to Mac users, but Lima can be used for non-container applications as well.

Lima also supports other container engines (Docker, Podman, Kubernetes, etc.) and non-macOS hosts (Linux, NetBSD, etc.).

Reference:

- Lima

QEMU README

QEMU is a generic and open source machine & userspace emulator and virtualizer.

QEMU is capable of emulating a complete machine in software without any need for hardware virtualization support. By using dynamic translation, it achieves very good performance. QEMU can also integrate with the Xen and KVM hypervisors to provide emulated hardware while allowing the hypervisor to manage the CPU. With hypervisor support, QEMU can achieve near native performance for CPUs. When QEMU emulates CPUs directly it is capable of running operating systems made for one machine (e.g. an ARMv7 board) on a different machine (e.g. an x86_64 PC board).

QEMU is also capable of providing userspace API virtualization for Linux and BSD kernel interfaces. This allows binaries compiled against one architecture ABI (e.g. the Linux PPC64 ABI) to be run on a host using a different architecture ABI (e.g. the Linux x86_64 ABI). This does not involve any hardware emulation, simply CPU and syscall emulation.

QEMU aims to fit into a variety of use cases. It can be invoked directly by users wishing to have full control over its behaviour and settings. It also aims to facilitate integration into higher level management layers, by providing a stable command line interface and monitor API. It is commonly invoked indirectly via the libvirt library when using open source applications such as oVirt, OpenStack and virt-manager.

QEMU as a whole is released under the GNU General Public License, version 2. For full licensing details, consult the LICENSE file.

Reference:

- Qemu

Lima & Qemu Stack (Cont.)

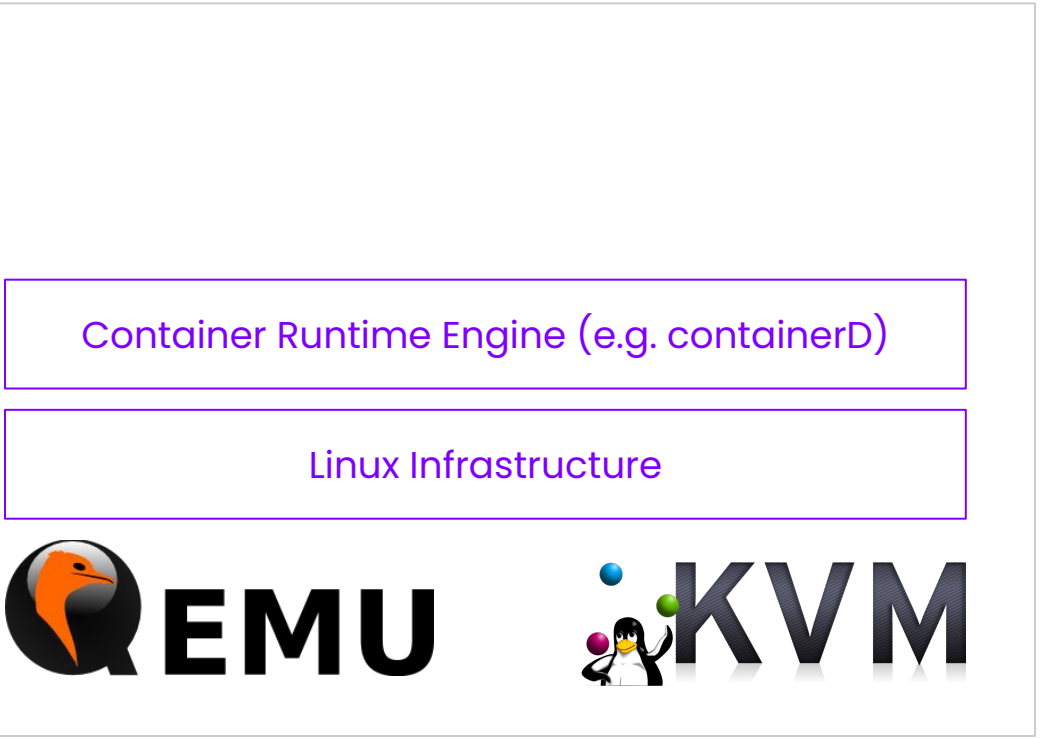
Lima

Linux Virtual Machine

Lima & Qemu Stack (Cont.)

Lima

Linux Virtual Machine



Lima & Qemu Stack (Cont.)

Linux Virtual Machine

Shell

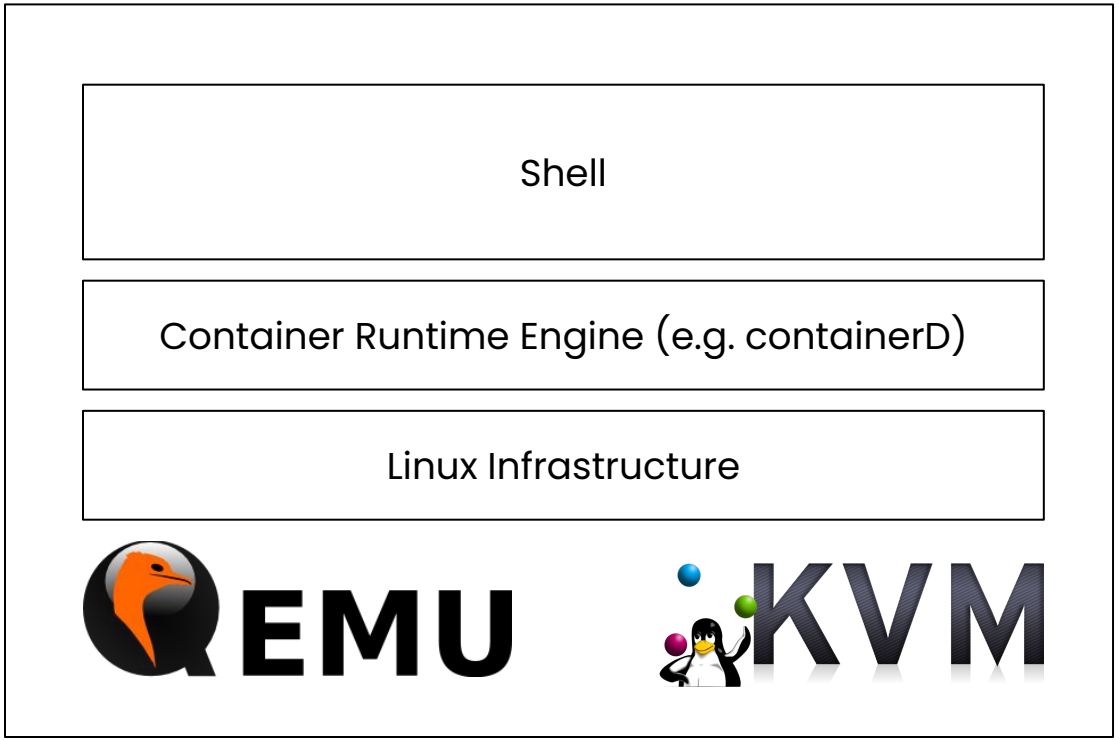
Container Runtime Engine (e.g. containerD)

Linux Infrastructure

Lima & Qemu Stack (Cont.)

Lima

Linux Virtual Machine



Demo

Demo with Python (Cont.)

```
#!/usr/bin/python
from bcc import BPF

program = """
int hello_world(void *ctx) {
   bpf_trace_printk("Hello World! NongKai is here. Enjoy!\\n");
   return 0;
}
"""
b = BPF(text=program)
syscall = b.get_syscall_fnname("execve")
b.attach_kprobe(event=syscall, fn_name="hello_world")

b.trace_print()
```

Reference:

- Python: bcc Reference Guide

Jumpbox®

Demo with Python (Cont.)

```
#!/usr/bin/python
from bcc import BPF
```

bcc: python framework easy to write eBPF Program

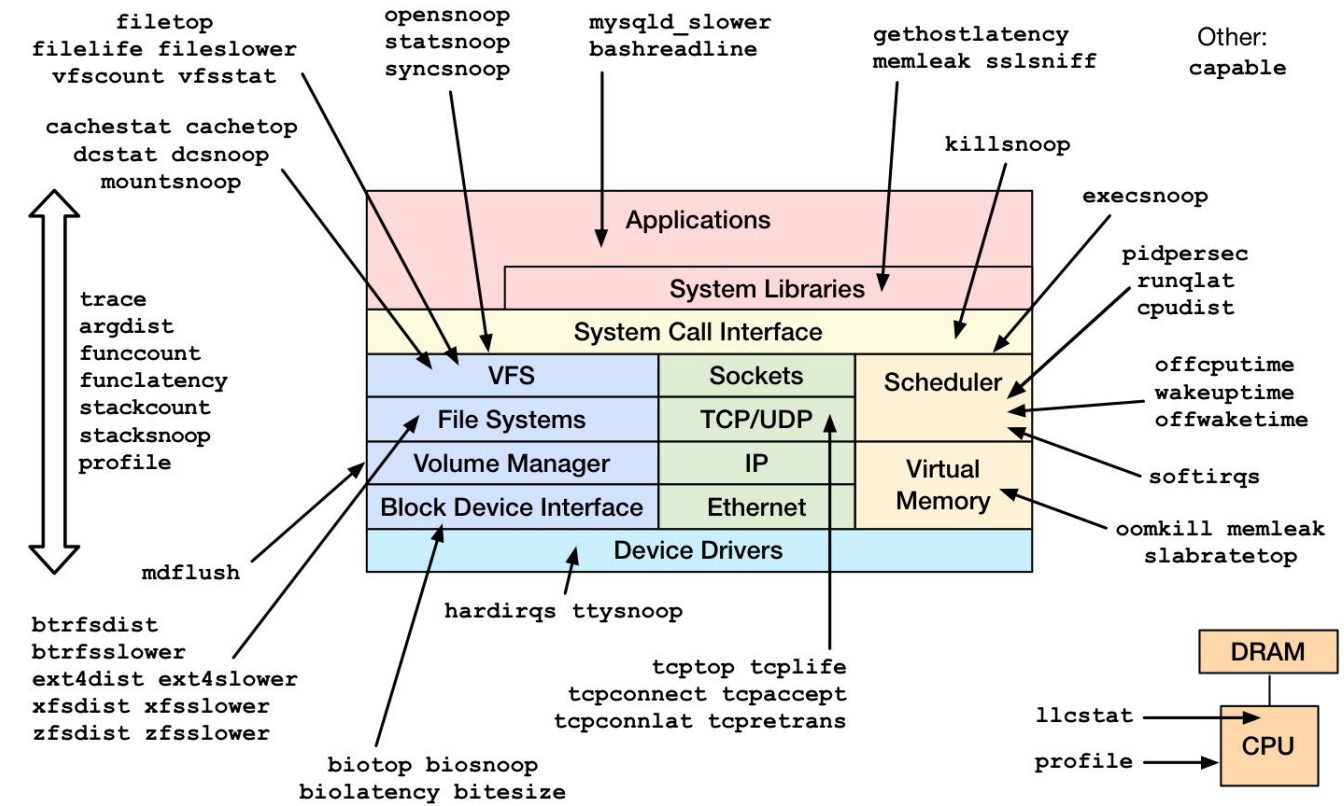
```
program = """
int hello_world(void *ctx) {
   bpf_trace_printk("Hello World! NongKai is here. Enjoy!\\n");
   return 0;
}
"""
b = BPF(text=program)
syscall = b.get_syscall_fnname("execve")
b.attach_kprobe(event=syscall, fn_name="hello_world")

b.trace_print()
```

Reference:

- Python: bcc Reference Guide

Jumpbox®

Linux bcc/BPF Tracing Tools



https://github.com/iovisor/bcc#tools 2016

Reference:

- Linux BCC

Demo with Python (Cont.)

```
#!/usr/bin/python
from bcc import BPF

program = """
int hello_world(void *ctx) {
   bpf_trace_printk("Hello World! NongKai is here. Enjoy!\\n");
   return 0;
}
"""
b = BPF(text=program)
syscall = b.get_syscall_fnname("execve")
b.attach_kprobe(event=syscall, fn_name="hello_world")

b.trace_print()
```

C function (hello_world):
when event involked by system call

Reference:

- Python: bcc Reference Guide

Jumpbox®

Demo with Python (Cont.)

```
#!/usr/bin/python
from bcc import BPF

program = """
int hello_world(void *ctx) {
   bpf_trace_printk("Hello World! NongKai is here. Enjoy!\\n");
   return 0
}
"""
b = BPF(text=program)
syscall = b.get_syscall_fnname("execve")
b.attach_kprobe(event=syscall, fn_name="hello_world")

b.trace_print()
```

Catch event:
execute program (execve)

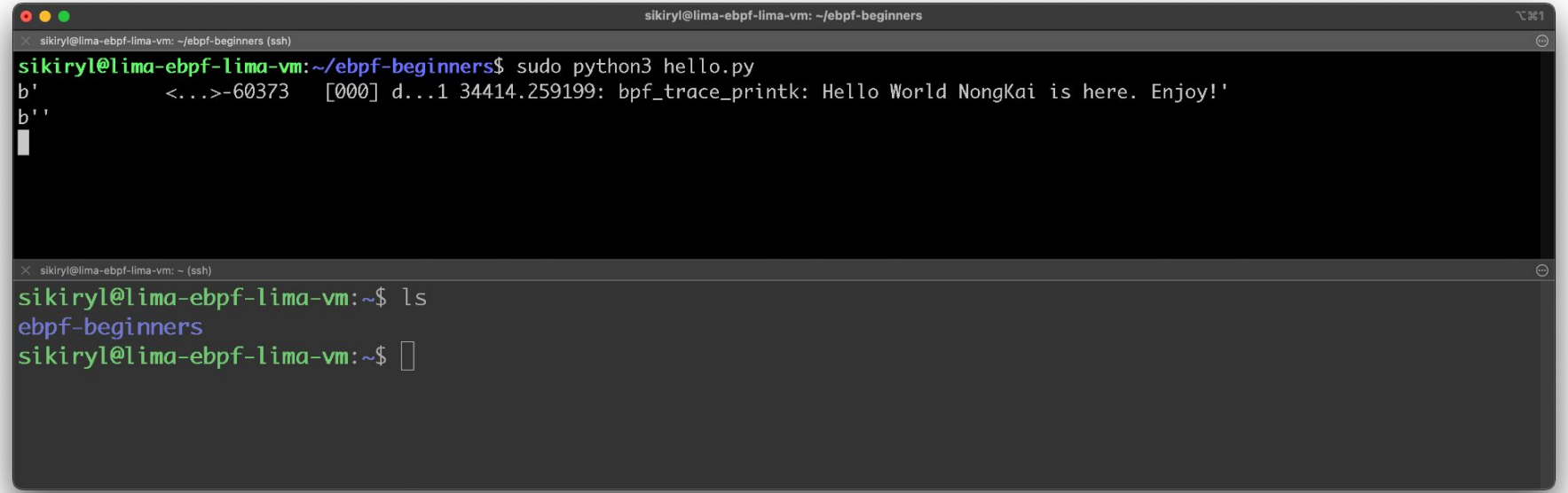
Reference:

- Linux System Call: execve

Jumpbox®

Demo with Python (Cont.)

Output:



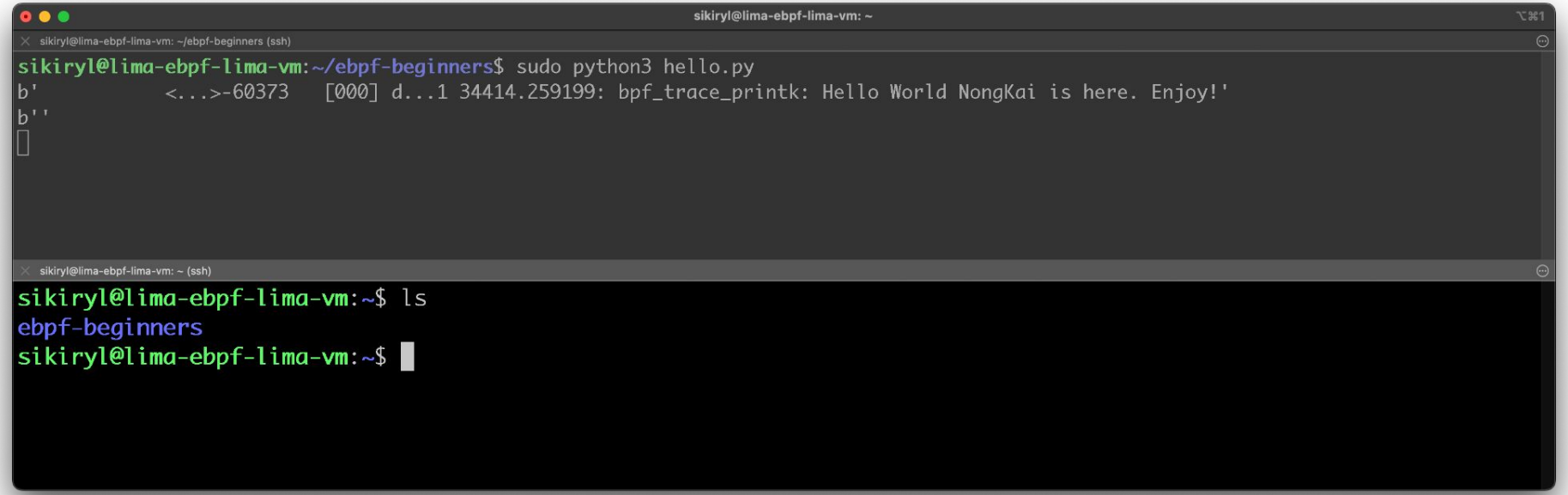
Reference:

- Linux System Call: execve

Jumpbox®

Demo with Python (Cont.)

Output:

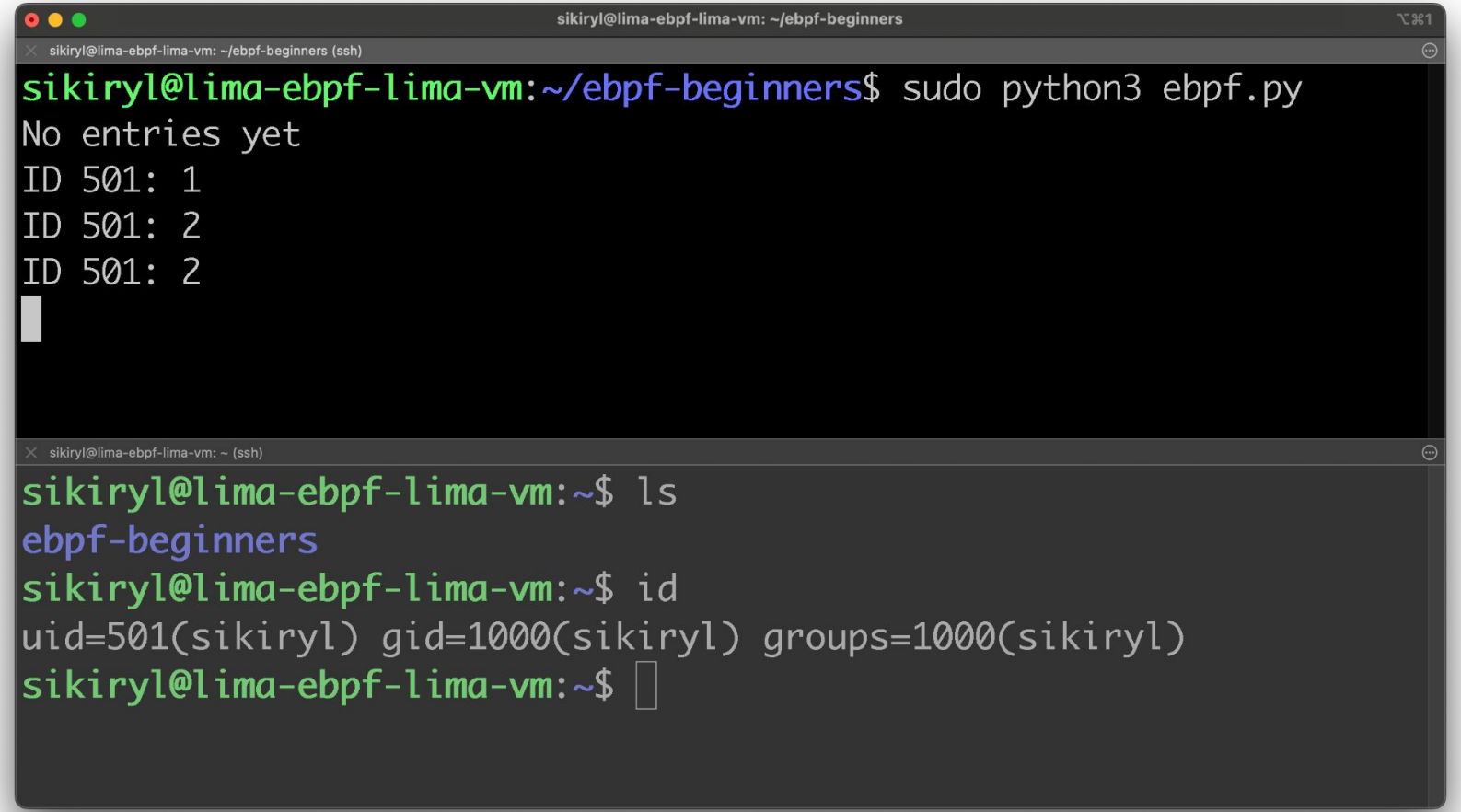


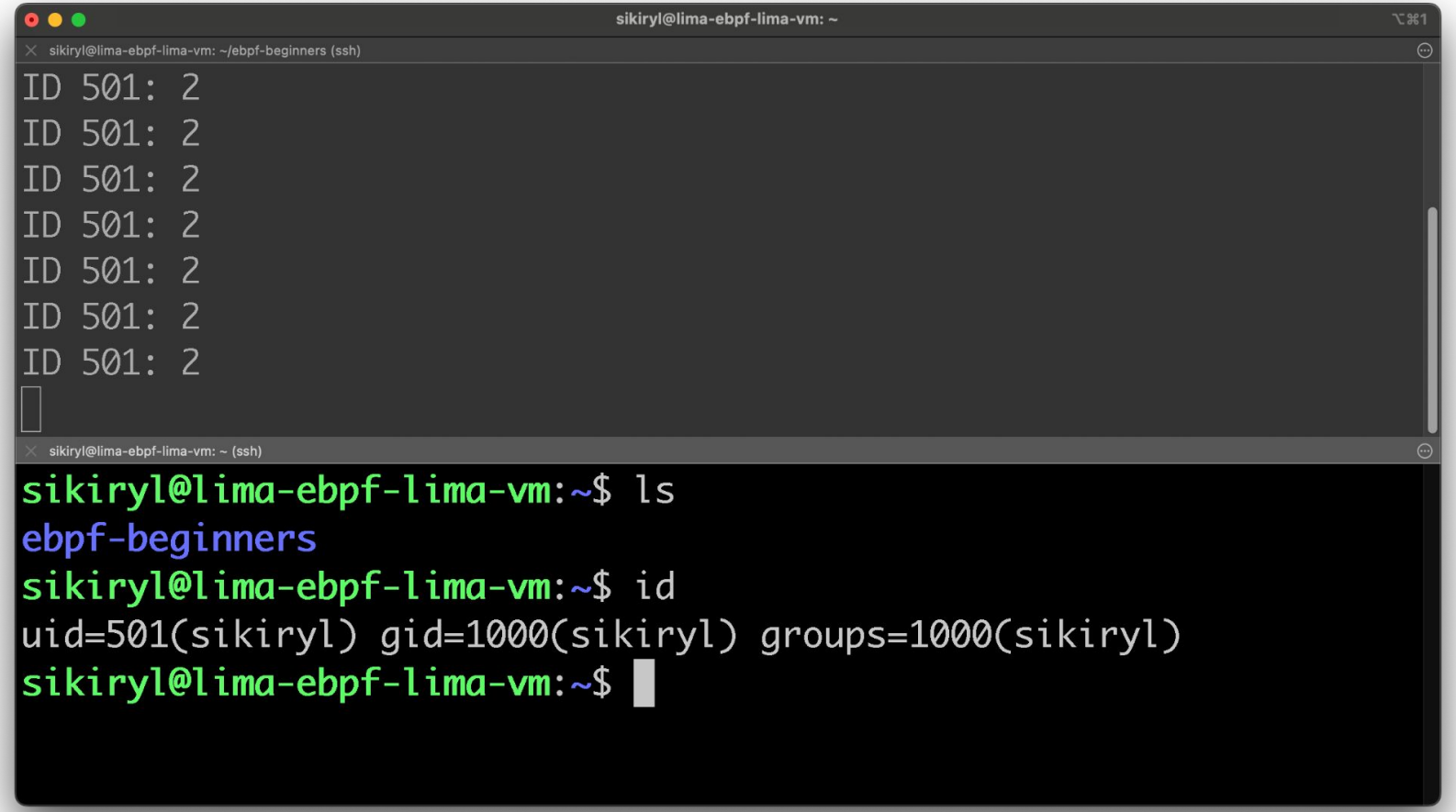
Reference:

- Linux System Call: execve

Jumpbox®

Catch UserID execution event


sikiryl@lima-ebpf-lima-vm: ~/ebpf-beginners
sikiryl@lima-ebpf-lima-vm: ~/ebpf-beginners (ssh)
sikiryl@lima-ebpf-lima-vm:~/ebpf-beginners$ sudo python3 ebpf.py
No entries yet
ID 501: 1
ID 501: 2
ID 501: 2
sikiryl@lima-ebpf-lima-vm: ~ (ssh)
sikiryl@lima-ebpf-lima-vm:~$ ls
ebpf-beginners
sikiryl@lima-ebpf-lima-vm:~$ id
uid=501(sikiryl) gid=1000(sikiryl) groups=1000(sikiryl)
sikiryl@lima-ebpf-lima-vm:~$


sikiryl@lima-ebpf-lima-vm: ~
sikiryl@lima-ebpf-lima-vm: ~/ebpf-beginners (ssh)
ID 501: 2
ID 501: 2
ID 501: 2
ID 501: 2
ID 501: 2
ID 501: 2
ID 501: 2
sikiryl@lima-ebpf-lima-vm: ~ (ssh)
sikiryl@lima-ebpf-lima-vm:~$ ls
ebpf-beginners
sikiryl@lima-ebpf-lima-vm:~$ id
uid=501(sikiryl) gid=1000(sikiryl) groups=1000(sikiryl)
sikiryl@lima-ebpf-lima-vm:~$

eBPF mentioned
at the latest KubeCon2024

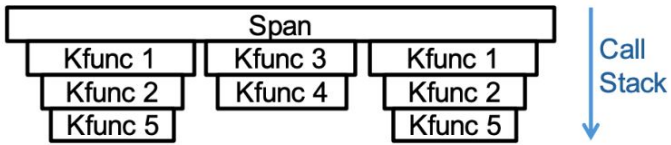
Jumpbox®

OpenTelemetry Amplified- Full Observability with EBPF-Enabled Distributed Tracing

Jumpbox®

Fine-grained Traces

- Fine-grained Traces
  - Collect telemetry data in Kernel
  - Context Propagation between kernel functions
  - Merge kernel spans with request-level spans



Example Demo

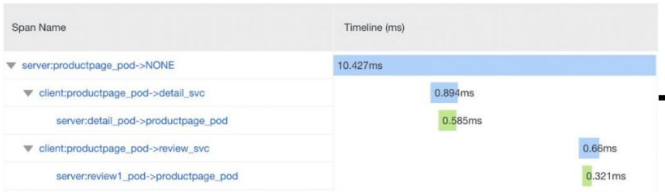
KubeCon CloudNativeCon China 2024 | THE LINUX FOUNDATION OPEN SOURCE SUMMIT | AI_dev

- Trace Visualization

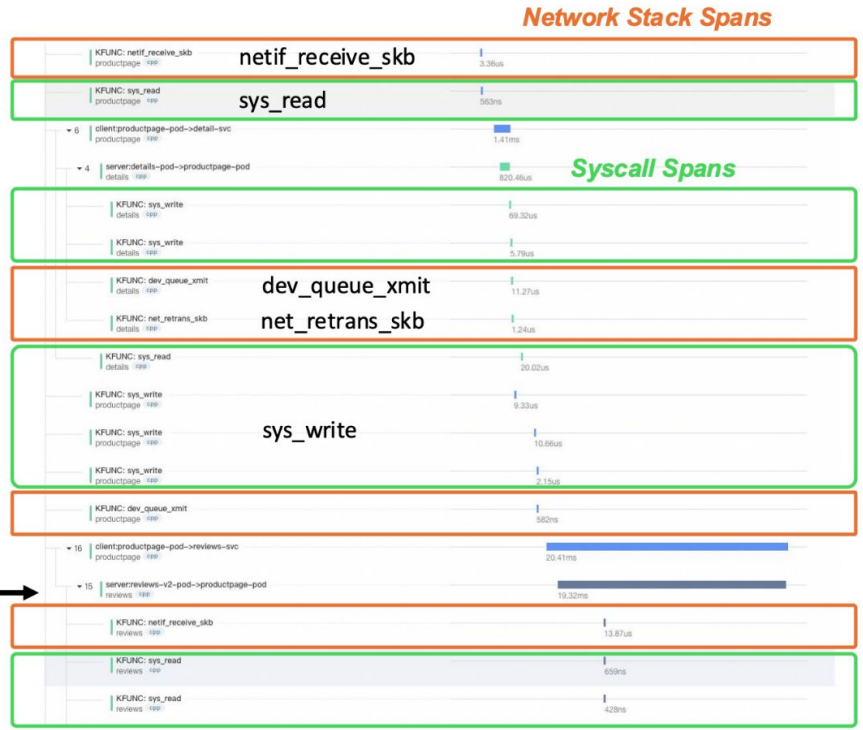
Request-level insights

Network insights

Security insights



Before



After

Jumpbox®

eBPF Topic in KubeCon2024

- OpenTelemetry Amplified- Full Observability with EBPF-Enabled Distributed Tracing
- Redefining Service Mesh: Leveraging EBPF to Optimize Istio Ambient Architecture and Performance
- KubeSkoop- Deal with the Complexity of Network Issues and Monitoring with eBPF
- Unlocking LLM Performance with EBPF: Optimizing Training and Inference Pipelines

Incoming Event

Reference:

- Incoming event: Cilium+eBPF Day North America November 12, 2024 Salt Lake City, Utah

Learn more about eBPF

- eBPF Everything You Need to Know in 5 Minutes
- Getting Started with eBPF
- Beginner’s Guide to eBPF Programming
- Beginner's Guide to eBPF Programming with Go
- Run Fast! Catch Performance Regressions in eBPF with Rust
- What is eBPF? Brightboard Lesson
- Brendan Gregg • BPF Performance Tools

ISOVALENT

x

Jumpbox®

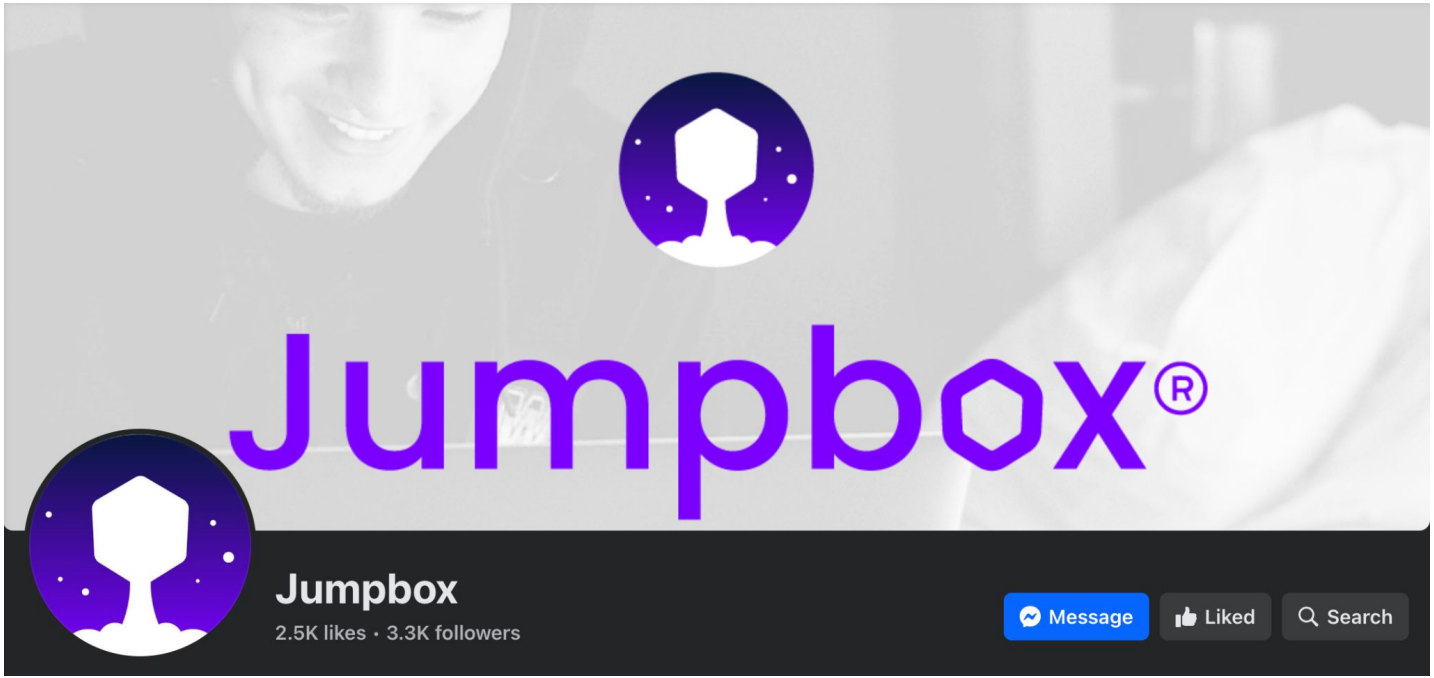
We are now partners

Jumpbox®

https://github.com/jumpbox-academy/learn-ebpf


Jumpbox®
THE CLOUD CAMP
2024

facebook



https://www.facebook.com/jumpbox.academy

Jumpbox®

YouTube

Jumpbox

@jumpbox.academy · 1.92K subscribers · 30 videos

More about this channel **...more**

https://www.youtube.com/@jumpbox.academy

Contact Us

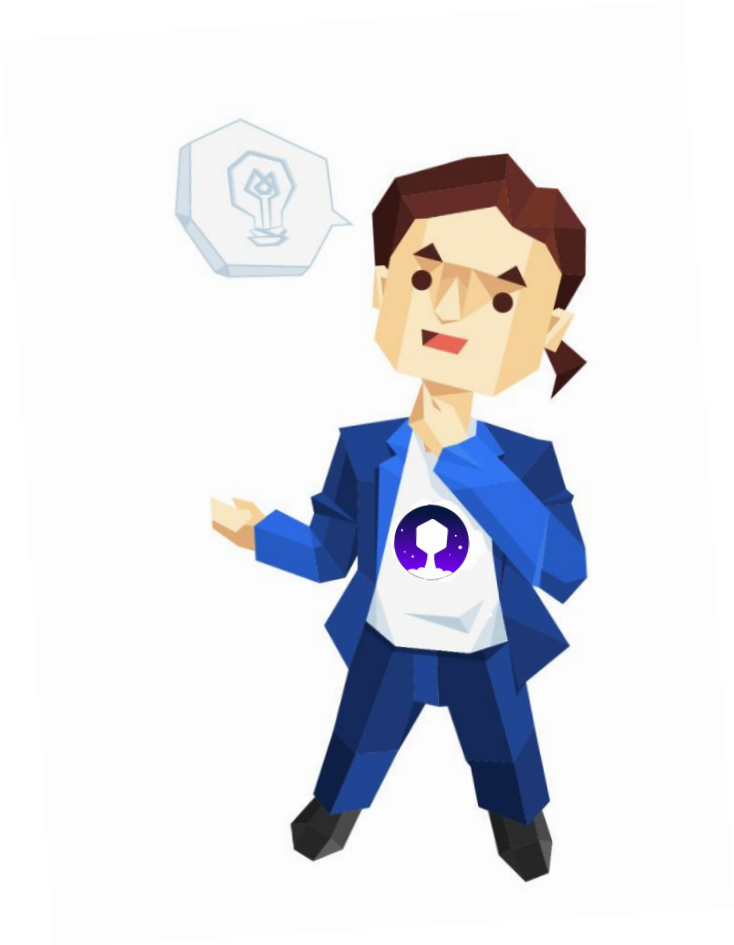
Jumpbox

@jumpbox

admin@jumpbox.co

063-245-2168 (JoJo)

062-796-1559 (Beau)



Jumpbox®

"เราเชื่อว่าการเรียนรู้ทำให้ชีวิตคุณดีขึ้น"

-Jumpbox Team-

Jumpbox®

Jumpbox®