CONFLUENT

eBPF: The next power tool of SRE's

Michael Kehoe
Sr Staff Security Engineer

Agenda

An introduction & history of BPF
What is all the fuss about?

How to get started with eBPF
Write your first program

Capability 1: Observability & Tracing
High performance, high fidelity tracing

Capability 2: Networking
Firewall, DDoS, Load-balancing

Capability 3: Security
Container & LSM controls

The future of eBPF & SRE
Where are we going

Introduction

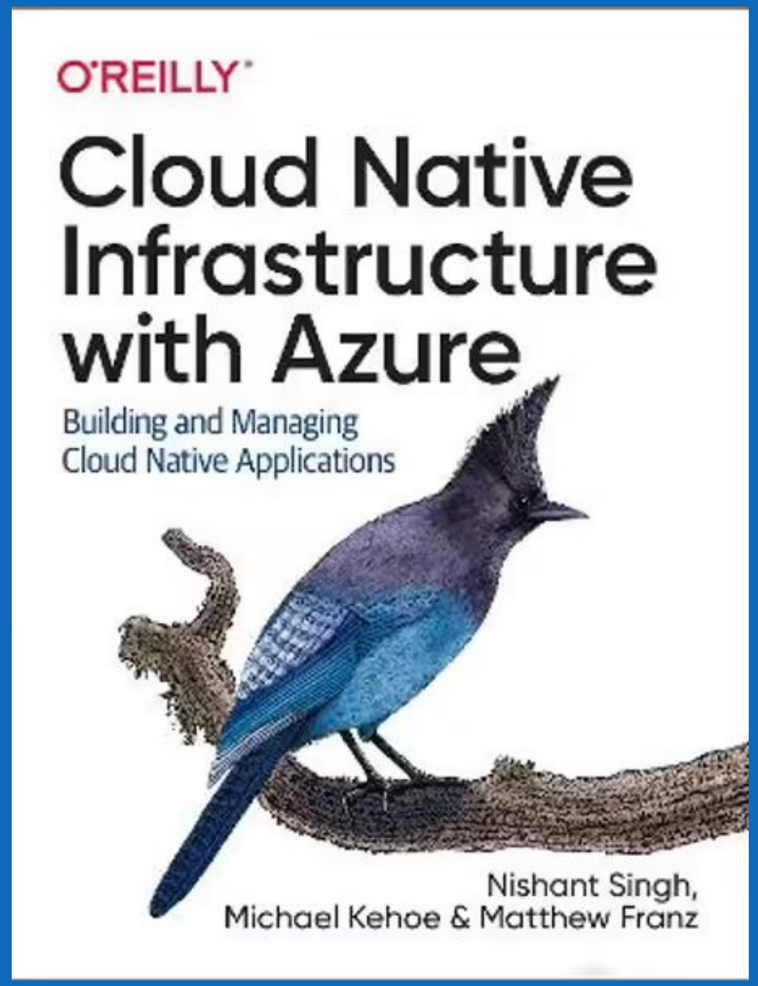
Introduction: Michael Kehoe

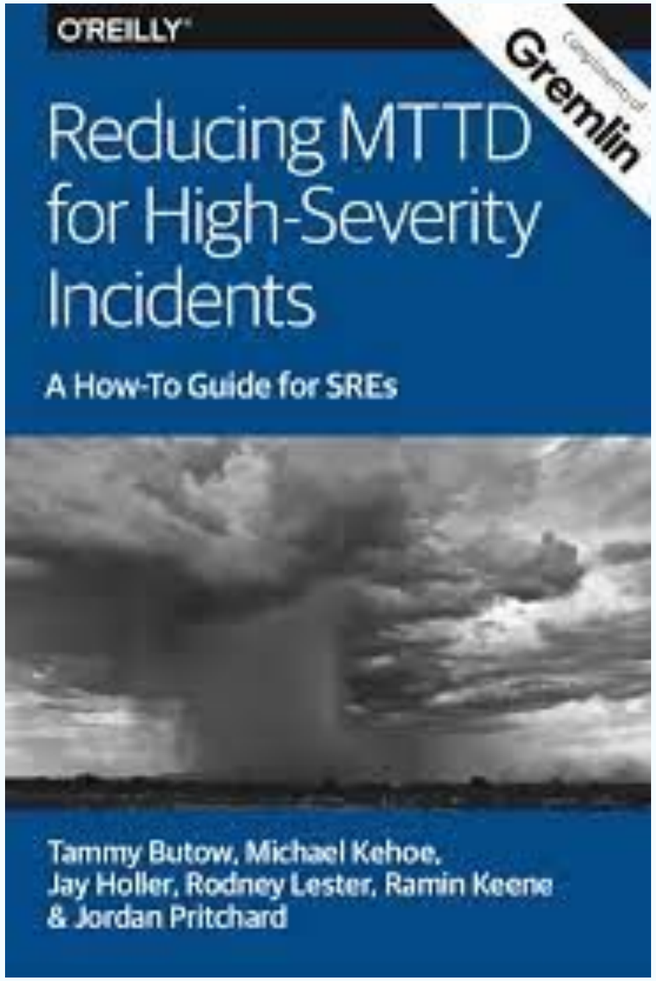
- Sr Staff Security Engineer - Confluent
  - InfraSec/ CloudSec team

- Previously:
  - Sr Staff SRE @ LinkedIn
  - PhoneSat intern @ NASA

- Background in:
  - Networks
  - Microservices
  - Traffic Engineering
  - KV Databases
  - Incident Management

- Twitter: @michaelkkehoe
- LinkedIn: linkedin.com/in/michaelkkkehoe
- Website: michael-kehoe.io


O'REILLY
Cloud Native Infrastructure with Azure
Building and Managing Cloud Native Applications
Nishant Singh, Michael Kehoe & Matthew Franz


O'REILLY
Compliments of Gremlin
Reducing MTTD for High-Severity Incidents
A How-To Guide for SREs
Tammy Butow, Michael Kehoe, Jay Holler, Rodney Lester, Ramin Keene & Jordan Pritchard

An Introduction to eBPF

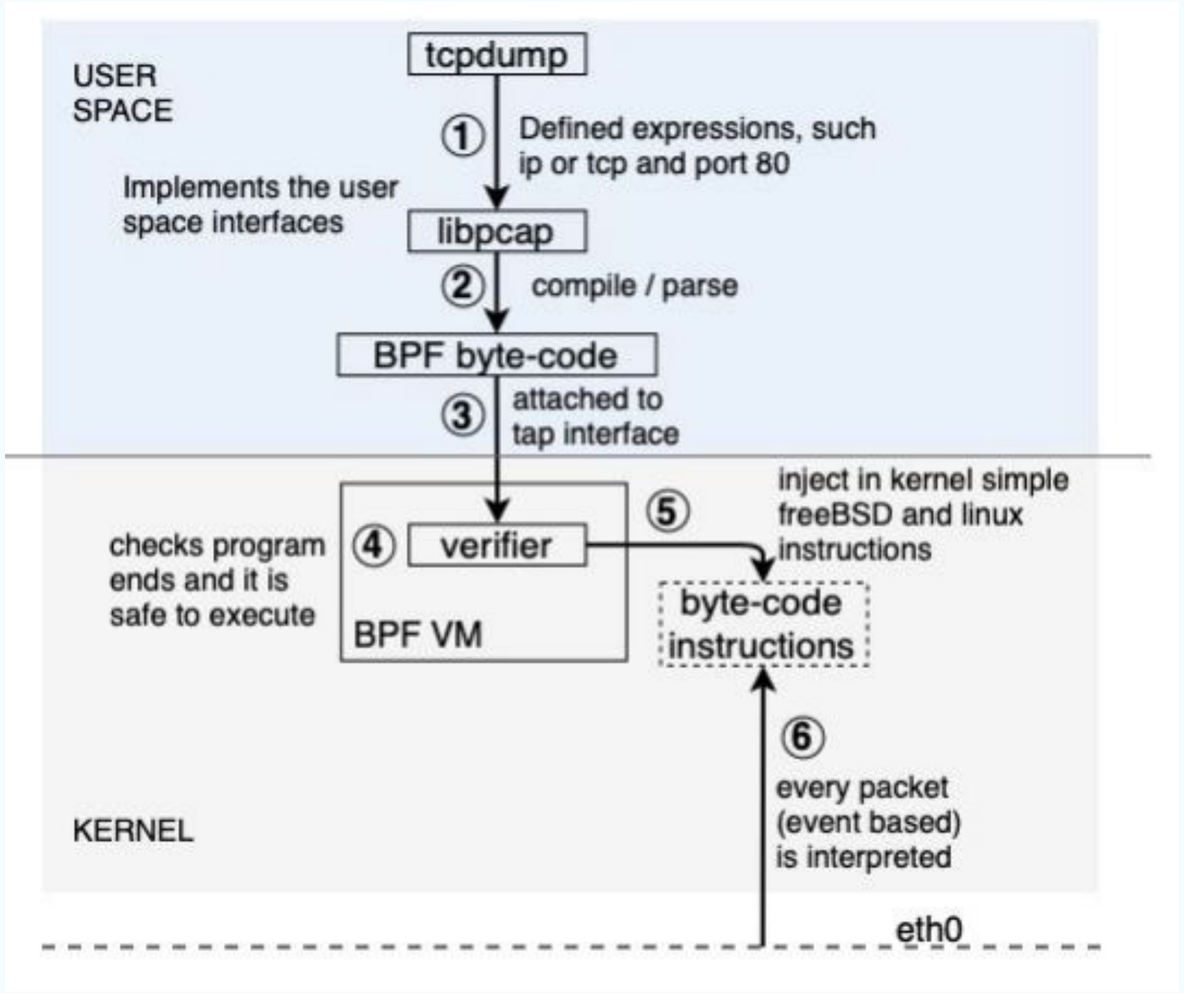
Put your hand up if you've used BPF before?

Put your hand up if you've used* tcpdump *before?

What is cBPF?

- cBPF - Classic BPF
  - Also known as “Linux Packet Filtering”
- BPF was first introduced in 1992 by Steven McCanne and Van Jacobson in BSD
  - Implemented in Linux kernel 2.2 (Linux Socket Filtering)
- Originally used for network packet filtering & later, seccomp
- Works by: Filter expressions → byte code → interpreter
- Uses: Small, in-kernel VM, Register based, limited instructions

What is cBPF?


USER
SPACE
tcpdump
1
Defined expressions, such
ip or tcp and port 80
Implements the user
space interfaces
libpcap
2
compile / parse
BPF byte-code
3
attached to
tap interface
inject in kernel simple
freeBSD and linux
instructions
5
checks program
ends and it is
safe to execute
4
verifier
BPF VM
byte-code
instructions
6
every packet
(event based)
is interpreted
KERNEL
eth0

What is eBPF?

“eBPF does to Linux what JavaScript does to HTML”

Brendan Gregg
Sr Performance Engineer, Netflix

“eBPF is Linux’s new superpower”

Gaurav Gupta
SAP Labs

“BPF is a highly flexible and efficient virtual machine-like construct in the Linux kernel allowing to execute bytecode at various hook points in a safe manner. It is used in a number of Linux kernel subsystems, most prominently networking, tracing and security (e.g. sandboxing).”

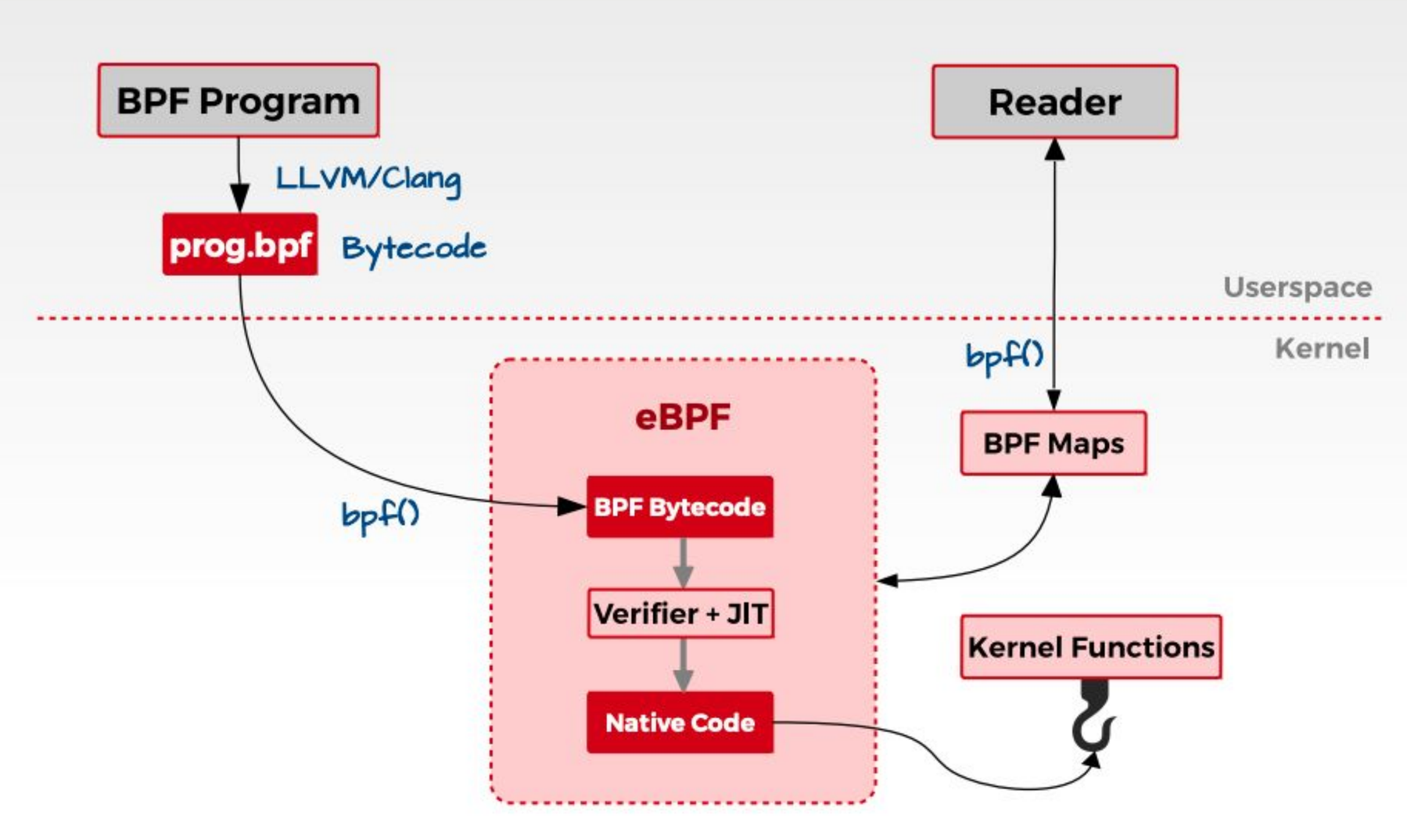
Cilium

What is eBPF?

- eBPF - extended Berkeley Packet Filter
- User defined, sandboxed bytecode executed by the kernel
- VM that implements a RISC-like assembly language in kernel space
- Multiple verification layers to ensure kernel safety
- Interactions between kernel/ user space are done through eBPF “maps”
    - And blocking trace pipes
- eBPF does not allow loops*
- Kernel-like functionality without the FUD

* Bounded loops in kernel 5.3

What is eBPF



http://hsdm.dorsal.polymtl.ca/system/files/eBPF-5May2017%20%281%29.pdf

bpf() system call

```
enum bpf_cmd {
        BPF_MAP_CREATE,
        BPF_MAP_LOOKUP_ELEM,
        BPF_MAP_UPDATE_ELEM,
        BPF_MAP_DELETE_ELEM,
        BPF_MAP_GET_NEXT_KEY,
        BPF_PROG_LOAD,
        BPF_OBJ_PIN,
        BPF_OBJ_GET,
        BPF_PROG_ATTACH,
        BPF_PROG_DETACH,
        BPF_PROG_TEST_RUN,
        BPF_PROG_RUN = BPF_PROG_TEST_RUN,
        BPF_PROG_GET_NEXT_ID,
        BPF_MAP_GET_NEXT_ID,
        BPF_PROG_GET_FD_BY_ID,
        BPF_MAP_GET_FD_BY_ID,
        BPF_OBJ_GET_INFO_BY_FD,
        BPF_PROG_QUERY,
        BPF_RAW_TRACEPOINT_OPEN,
        BPF_BTF_LOAD,
        BPF_BTF_GET_FD_BY_ID,
        BPF_TASK_FD_QUERY,
        BPF_MAP_LOOKUP_AND_DELETE_ELEM,
        BPF_MAP_FREEZE,
        BPF_BTF_GET_NEXT_ID,
        BPF_MAP_LOOKUP_BATCH,
        BPF_MAP_LOOKUP_AND_DELETE_BATCH,
        BPF_MAP_UPDATE_BATCH,
        BPF_MAP_DELETE_BATCH,
        BPF_LINK_CREATE,
        BPF_LINK_UPDATE,
        BPF_LINK_GET_FD_BY_ID,
        BPF_LINK_GET_NEXT_ID,
        BPF_ENABLE_STATS,
        BPF_ITER_CREATE,
        BPF_LINK_DETACH,
        BPF_PROG_BIND_MAP,
};
```

bpf_cmd

Interface between user-space & eBPF VM

eBPF Program Types

```
enum bpf_prog_type {
        BPF_PROG_TYPE_UNSPEC,
        BPF_PROG_TYPE_SOCKET_FILTER,
        BPF_PROG_TYPE_KPROBE,
        BPF_PROG_TYPE_SCHED_CLS,
        BPF_PROG_TYPE_SCHED_ACT,
        BPF_PROG_TYPE_TRACEPOINT,
        BPF_PROG_TYPE_XDP,
        BPF_PROG_TYPE_PERF_EVENT,
        BPF_PROG_TYPE_CGROUP_SKB,
        BPF_PROG_TYPE_CGROUP_SOCK,
        BPF_PROG_TYPE_LWT_IN,
        BPF_PROG_TYPE_LWT_OUT,
        BPF_PROG_TYPE_LWT_XMIT,
        BPF_PROG_TYPE_SOCK_OPS,
        BPF_PROG_TYPE_SK_SKB,
        BPF_PROG_TYPE_CGROUP_DEVICE,
        BPF_PROG_TYPE_SK_MSG,
        BPF_PROG_TYPE_RAW_TRACEPOINT,
        BPF_PROG_TYPE_CGROUP_SOCK_ADDR,
        BPF_PROG_TYPE_LWT_SEG6LOCAL,
        BPF_PROG_TYPE_LIRC_MODE2,
        BPF_PROG_TYPE_SK_REUSEPORT,
        BPF_PROG_TYPE_FLOW_DISSECTOR,
        BPF_PROG_TYPE_CGROUP_SYSCTL,
        BPF_PROG_TYPE_RAW_TRACEPOINT_WRITABLE,
        BPF_PROG_TYPE_CGROUP_SOCKOPT,
        BPF_PROG_TYPE_TRACING,
        BPF_PROG_TYPE_STRUCT_OPS,
        BPF_PROG_TYPE_EXT,
        BPF_PROG_TYPE_LSM,
        BPF_PROG_TYPE_SK_LOOKUP,
        BPF_PROG_TYPE_SYSCALL, /* a program that can execute syscalls */
};
```

bpf_prog_type

Determines the subset of kernel helper functions the program may call

bpf_context

The program type will help determine the set of arguments given to a eBPF program

eBPF Map Types

```
enum bpf_map_type {
        BPF_MAP_TYPE_UNSPEC,
        BPF_MAP_TYPE_HASH,
        BPF_MAP_TYPE_ARRAY,
        BPF_MAP_TYPE_PROG_ARRAY,
        BPF_MAP_TYPE_PERF_EVENT_ARRAY,
        BPF_MAP_TYPE_PERCPU_HASH,
        BPF_MAP_TYPE_PERCPU_ARRAY,
        BPF_MAP_TYPE_STACK_TRACE,
        BPF_MAP_TYPE_CGROUP_ARRAY,
        BPF_MAP_TYPE_LRU_HASH,
        BPF_MAP_TYPE_LRU_PERCPU_HASH,
        BPF_MAP_TYPE_LPM_TRIE,
        BPF_MAP_TYPE_ARRAY_OF_MAPS,
        BPF_MAP_TYPE_HASH_OF_MAPS,
        BPF_MAP_TYPE_DEVMAP,
        BPF_MAP_TYPE_SOCKMAP,
        BPF_MAP_TYPE_CPUMAP,
        BPF_MAP_TYPE_XSKMAP,
        BPF_MAP_TYPE_SOCKHASH,
        BPF_MAP_TYPE_CGROUP_STORAGE,
        BPF_MAP_TYPE_REUSEPORT_SOCKARRAY,
        BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE,
        BPF_MAP_TYPE_QUEUE,
        BPF_MAP_TYPE_STACK,
        BPF_MAP_TYPE_SK_STORAGE,
        BPF_MAP_TYPE_DEVMAP_HASH,
        BPF_MAP_TYPE_STRUCT_OPS,
        BPF_MAP_TYPE_RINGBUF,
        BPF_MAP_TYPE_INODE_STORAGE,
        BPF_MAP_TYPE_TASK_STORAGE,
        BPF_MAP_TYPE_BLOOM_FILTER,
};
```

eBPF Maps

- Generic structure for storage of different data types
- Allows sharing of data:
  - Within an eBPF program
  - Between kernel & user space

eBPF Helpers

int bpf_trace_printk(const char *fmt, u32 fmt_size, ...)

Description

This helper is a "printk()-like" facility for debugging. It prints a message defined by format fmt (of size fmt_size) to file /sys/kernel/debug/tracing/trace from DebugFS, if available. It can take up to three additional **u64** arguments (as an eBPF helpers, the total number of arguments is limited to five).

Each time the helper is called, it appends a line to the trace. Lines are discarded while /sys/kernel/debug/tracing/trace is open, use /sys/kernel/debug/tracing/trace_pipe to avoid this. The format of the trace is customizable, and the exact output one will get depends on the options set in /sys/kernel/debug/tracing/trace_options (see also the README file under the same directory). However, it usually defaults to something like:

```
telnet-470   [001] .N.. 419421.045894: 0x00000001: <formatted msg>
```

In the above:

- **telnet** is the name of the current task.
- **470** is the PID of the current task.
- **001** is the CPU number on which the task is running.
- In **.N..**, each character refers to a set of options (whether irqs are enabled, scheduling options, whether hard/softirqs are running, level of preempt_disabled respectively). **N** means that **TIF_NEED_RESCHED** and **PREEMPT_NEED_RESCHED** are set.
- **419421.045894** is a timestamp.
- **0x00000001** is a fake value used by BPF for the instruction pointer register.
- **<formatted msg>** is the message formatted with fmt.

https://manpages.ubuntu.com/manpages/focal/man7/bpf-helpers.7.html

eBPF Helpers

- Specific functions to be run within an eBPF program
- Various functionality
  - Manipulating maps
  - Debug functions
  - Load data from packets
  - ....and more
- Check your kernel for compatibility

How to get started with eBPF

Where to get started with eBPF

1. Run the most recent kernel possible
2. Ensure that eBPF kernel configuration options are set to ‘y’
3. Install bcctools (https://github.com/iovisor/bcc/)
4. Start coding

Where to get started with eBPF

```
CONFIG_BPF=y
CONFIG_BPF_SYSCALL=y

# [optional, for tc filters/ actions]
CONFIG_NET_CLS_BPF=m
CONFIG_NET_ACT_BPF=m

CONFIG_BPF_JIT=y

# [for Linux kernel versions 5.7 and later]
CONFIG_BPF_LSM=y

# [for Linux kernel versions 4.7 and later]
CONFIG_HAVE_EBPF_JIT=y

# [optional, for kprobes]
CONFIG_BPF_EVENTS=y

# Need kernel headers through /sys/kernel/kheaders.tar.xz
CONFIG_IKHEADERS=y
```

How to get started with eBPF

```
# CentOS/ Redhat
$ sudo yum install bcc bcc-doc bcc-tools

# Debian/ Ubuntu
$ sudo apt-get install bpfcc-tools linux-headers-$(uname -r)
```

Where to get started with eBPF: Hello World

```
from bcc import BPF

# Kernel-Space
prog = “““
    int kprobe__sys_clone(void *ctx) {
        bpf_trace_printk("Hello, World!\\n");
        return 0;
    }
”””

# User-Space
BPF(text=prog).trace_print()
```

https://github.com/iovisor/bcc/blob/master/docs/tutorial_bcc_python_developer.md

Where to get started with eBPF: Hello World

```
michael@laptop:~$ sudo python ebpf_demo.py
b' Privileged Cont-3480 [005] d... 78819.733331: bpf_trace_printk: Hello, World!'
b''
b'   WebExtensions-3801    [001] d... 78819.816553: bpf_trace_printk: Hello, World!'
b''
b'   WebExtensions-3801    [001] d... 78819.822080: bpf_trace_printk: Hello, World!'
b''
b'   WebExtensions-3801    [001] d... 78819.822308: bpf_trace_printk: Hello, World!'
b''
b'   WebExtensions-3801    [001] d... 78819.822495: bpf_trace_printk: Hello, World!'
```

Capability 1: Observability

eBPF Observability

K(ret)probes/ U(ret)probes

- Captures the entering (or exiting) of a kprobe or uprobe
- Exceptionally useful for capturing:
  - Disk operations
  - Network connections
  - Execution of programs

USDT’s

- Captures user statically defined tracepoints (USDT’s) in a program
- You can add tracepoints to your own program and then debug it with eBPF

eBPF Observability

Tracepoints

- Allows you to instrument (pre-defined) tracepoints in kernel code.
- Can have higher performance than kprobes

Perf Events

- Allows you instrument software and hardware performance events otherwise known as perf-events

Observability: disksnoop.py

```
from bcc import BPF
from bcc.utils import printb

b = BPF(text="""
#include <uapi/linux/ptrace.h>
#include <linux/blk-mq.h>

BPF_HASH(start, struct request *);

void trace_start(struct pt_regs *ctx, struct request *req) {
        // stash start timestamp by request ptr
        u64 ts = bpf_ktime_get_ns();
        start.update(&req, &ts);
}

void trace_completion(struct pt_regs *ctx, struct request *req) {
        u64 *tsp, delta;
        tsp = start.lookup(&req);
        if (tsp != 0) {
                delta = bpf_ktime_get_ns() - *tsp;
                bpf_trace_printk("%d %x %d\\n", req->__data_len,
                    req->cmd_flags, delta / 1000);
                start.delete(&req);
        }
}
""")
```

https://github.com/iovisor/bcc/blob/master/examples/tracing/disksnoop.py

Observability: disksnoop.py

```
b.attach_kprobe(event="blk_mq_start_request", fn_name="trace_start")
b.attach_kprobe(event="blk_account_io_done", fn_name="trace_completion")

while 1:
    try:
        (task, pid, cpu, flags, ts, msg) = b.trace_fields()
        (bytes_s, bflags_s, us_s) = msg.split()

        if int(bflags_s, 16):
            type_s = b"W"
        elif bytes_s == "0":   # see blk_fill_rwbs() for logic
            type_s = b"M"
        else:
            type_s = b"R"
        ms = float(int(us_s, 10)) / 1000

        printb(b"%-18.9f %-2s %-7s %8.2f" % (ts, type_s, bytes_s, ms))
    except KeyboardInterrupt:
        exit()
```

https://github.com/iovisor/bcc/blob/master/examples/tracing/disksnoop.py

Observability: disksnoop.py

```
$ ./disksnoop.py

TIME(s)              T  BYTES    LAT(ms)
16458043.435457    W  4096       2.73
16458043.435981    W  4096       3.24
16458043.436012    W  4096       3.13
16458043.437326    W  4096       4.44
16458044.126545    R  4096      42.82
16458044.129872    R  4096       3.24
16458044.130705    R  4096       0.73
16458044.142813    R  4096      12.01
16458044.147302    R  4096       4.33
16458044.148117    R  4096       0.71
```

https://github.com/iovisor/bcc/blob/master/examples/tracing/disksnoop_example.txt

Capability 2: Networking

eBPF Networking

Load balancing

Easily load-balance/ forward millions of packets per second

Control of sockets

Additional controls for sockets after they have been created

Network Filters/ DDoS protection

Easily firewall/ filter millions of packets per second

Flow dissection

Write custom programs to perform network flow dissection for monitoring & accounting

Traffic Control (tc)

Prioritize/ monitor flows

eBPF Networking

- Katran (Facebook load balancer)
- Cilium/ Hubble (Kubernetes network load-balancing/ firewall & more)
- Calico (Kubernetes CNI)
- Cloudflare edge infra (read their blog)
- https://github.com/iovisor/bcc/tree/master/examples/networking
- https://blog.cloudflare.com/tag/ebpf/

Capability 3: Security

eBPF Security

cgroup device

- Control/ monitor usage of host’s devices by a cgroup

cgroup sysctl

- Control/ monitor usage of host’s sysctl’s by a cgroup

eBPF Security

cgroup skb

- Firewall/ network-filters for cgroups

LSM

- Instruments an LSM hook as a BPF program.
- It can be used to audit security events and implement MAC security policies in BPF.

Security: LSM example

```
import os
import sys
import time

from bcc import BPF, libbcc

src = """
#include <linux/fs.h>
#include <uapi/asm-generic/errno-base.h>

LSM_PROBE(file_open, struct file *file) {
    bpf_trace_printk("LSM hook: file_open\\n");

    u32 pid = bpf_get_current_pid_tgid();
    if (pid != 1) {
        bpf_trace_printk("LSM hook: file_open: Denied\\n");
        return -EPERM;
    }
    bpf_trace_printk("LSM hook: file_open: Allowed\\n");
    return 0;
}
"""
```

Ref: https://www.kernel.org/doc/html/v5.2/security/LSM.html

Security: LSM example

```
b = BPF(text=src)
fn = b.load_func("file_open", BPF.LSM)

try:
    while 1:
        time.sleep(0.5)
        print(b.trace_fields())
       # Extra logging logic
except KeyboardInterrupt:
    sys.exit()
```

Ref: https://www.kernel.org/doc/html/v5.2/security/LSM.html

The future of eBPF & SRE

The future of eBPF & SRE

Observability

- Allows you to troubleshoot low-level issues without worrying about performance
    - Never have to use *strace* again
- Opens up new possibilities to optimize user-owned software and locate bugs

Networking

- Real-life examples in Kubernetes/ Cilium
- Hyperscale for everyone:
    - Firewalls
    - Load-balancing
    - WAFs

Security

- Deep integration with LSM's for rich runtime security data
- Cgroup protections:
    - Devices
    - sysctl's
    - Network Traffic

The future of eBPF & SRE: Words of caution

- Despite the performance of eBPF, you can still harm your system
    - Know your performance boundaries/ limitations
- Be wary of OS/ kernel compatibility
    - CentOS/ Redhat often backport to older kernels
- You will need to think about your deployment strategies (hint: look at CO-RE)
    - Running programs via systemd is an option
- While eBPF is kernel-safe, you still need to thoroughly test before production

Resources

- https://github.com/michael-kehoe/bpf-workshop
- https://ebpf.io/
- https://docs.cilium.io/en/stable/bpf/
- https://github.com/iovisor/bcc
- https://github.com/aquasecurity/tracee
- Linux Observability with BPF (Book)
- BPF Performance Tools (Book)

Q & A


CONFLUENT