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/michaelkkehoe
- Website: michael-kehoe.io
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?

1. tcpdump
   - Defined expressions, such as ip or tcp and port 80

2. libpcap
   - Compile / parse

3. BPF byte-code
   - Attached to tap interface

4. Verifier
   - Checks program ends and it is safe to execute

5. BPF VM
   - Inject in kernel simple FreeBSD and Linux instructions

6. byte-code instructions
   - Every packet (event based) is interpreted

USER SPACE

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

bpf() system call

```c
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_STMT_GET_FD_BY_ID,
    BPF_TASK_FD_QUERY,
    BPF_MAP_LOOKUP_AND_DELETE_ELEM,
    BPF_MAP_FREEZE,
    BPF_STMT_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_ITER_DETACH,
    BPF_PROG_BIND_MAP,
};
```
eBPF Program Types

```c
enum bpf_prog_type {
  BPF_PROG_TYPE_UNSPEC,
  BPF_PROG_TYPE_SOCKET_FILTER,
  BPF_PROG_TYPE_UPROBE,
  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_LUT_IN,
  BPF_PROG_TYPE_LUT_OUT,
  BPF_PROG_TYPE_LUT_XMIT,
  BPF_PROG_TYPE_SOCK_OPS,
  BPF_PROG_TYPE_SKB_SKB,
  BPF_PROG_TYPE_CGROUP_DEVICE,
  BPF_PROG_TYPE_SKB_MSG,
  BPF_PROG_TYPE_RAW_TRACEPOINT,
  BPF_PROG_TYPE_CGROUP_SOCK_ADDR,
  BPF_PROG_TYPE_LUT_SEDLOCAL,
  BPF_PROG_TYPE_LIRL_MODE2,
  BPF_PROG_TYPE_SKB_REUSEPORT,
  BPF_PROG_TYPE_FLOW_DISSECTOR,
  BPF_PROG_TYPE_CGROUP_SYSCTL,
  BPF_PROG_TYPE_RAW_TRACEPOINT_WRITEABLE,
  BPF_PROG_TYPE_CGROUP_SOCKOPT,
  BPF_PROG_TYPE_TRACING,
  BPF_PROG_TYPE_STRUCT_OPS,
  BPF_PROG_TYPE_EXT,
  BPF_PROG_TYPE_LSN,
  BPF_PROG_TYPE_SKB_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**

```c
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_GROUPTHREADARRAY,
    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_CMPMAP,
    BPF_MAP_TYPE_XDMPMAP,
    BPF_MAP_TYPE_SOCKHASH,
    BPF_MAP_TYPE_GROUPTHREAD_STORAGE,
    BPF_MAP_TYPE_REUSEPORT_SOCKARRAY,
    BPF_MAP_TYPE_PERCPU_GROUPTHREAD_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

- 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

```
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/trace/tracing where DebugFS is available. It can take up
    to three additional arguments (as an eBPF helpers, the total number of
    arguments is limited to four).

    Each time the helper is called, it appends a line to the trace. Lines are
    discarded while /sys/kernel/debug/tracing/tracing is open, use
    /sys/kernel/debug/tracing/tracing.trace 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/tracing.options (see also the README file
    under the same directory). However, it usually defaults to something like:

    telnet:478 [001] ... 439421:045004: 0x00000001: <formatted msg>

    In the above:
    - telnet is the name of the current task.
    - 478 is the PID of the current task.
    - 001 is the CPU number on which the task is running.
    - In ... each character refers to a set of options (whether irqs are
    enabled, scheduling options, whether hard/softirqs are running, levels
    of preempt_disabled respectively). A means that T Kim_NEEDED and
    PREEMPT_NEEDED are set.
    - 439421:045004 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
```
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

```bash
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
```
# 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
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
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()
$ ./disksnoop.py

<table>
<thead>
<tr>
<th>TIME(s)</th>
<th>T</th>
<th>BYTES</th>
<th>LAT(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16458043.435457</td>
<td>W</td>
<td>4096</td>
<td>2.73</td>
</tr>
<tr>
<td>16458043.435981</td>
<td>W</td>
<td>4096</td>
<td>3.24</td>
</tr>
<tr>
<td>16458043.436012</td>
<td>W</td>
<td>4096</td>
<td>3.13</td>
</tr>
<tr>
<td>16458043.437326</td>
<td>W</td>
<td>4096</td>
<td>4.44</td>
</tr>
<tr>
<td>16458044.126545</td>
<td>R</td>
<td>4096</td>
<td>42.82</td>
</tr>
<tr>
<td>16458044.129872</td>
<td>R</td>
<td>4096</td>
<td>3.24</td>
</tr>
<tr>
<td>16458044.130705</td>
<td>R</td>
<td>4096</td>
<td>0.73</td>
</tr>
<tr>
<td>16458044.142813</td>
<td>R</td>
<td>4096</td>
<td>12.01</td>
</tr>
<tr>
<td>16458044.147302</td>
<td>R</td>
<td>4096</td>
<td>4.33</td>
</tr>
<tr>
<td>16458044.148117</td>
<td>R</td>
<td>4096</td>
<td>0.71</td>
</tr>
</tbody>
</table>
Capability 2: Networking
eBPF Networking

Load balancing
Easily load-balance/forward millions of packets per second

Network Filters/ DDoS protection
Easily firewall/filter millions of packets per second

Traffic Control (tc)
Prioritize/monitor flows

Control of sockets
Additional controls for sockets after they have been created

Flow dissection
Write custom programs to perform network flow dissection for monitoring & accounting
- 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://github.com/iovisor/bcc/tree/master/examples/networking)
- [https://blog.cloudflare.com/tag/ebpf/](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.
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;
}"

Security: LSM example

```python
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()
```

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)