solo.io

eBPF: A Top-Down View

Lawrence Gadban
Field Engineer

Agenda

01 eBPF Overview

02 Top-Down View

03 Demo

logo credit: https://github.com/ebpf-io/ebpf.io/blob/master/src/assets/logo-big.png

What is eBPF?

- “Extended” Berkeley Packet Filter, evolution of BPF – think `tcpdump(8)`
- Linux technology which enables custom logic to be “attached” to various “hook points” in the kernel (and elsewhere!)
- Event-based (think JavaScript)
- eBPF programs are verified to be “safe” – won’t crash the kernel, guaranteed to return, only accesses specific sections of memory, etc.

What is an eBPF program?

- “An eBPF program is a sequence of 64-bit instructions.” ~ https://github.com/iovisor/bpf-docs/blob/master/eBPF.md
- eBPF runs as a VM, has its own instruction set, i.e. “bytecode”
- eBPF bytecode can then be JIT compiled to underlying architecture – i.e. the code ultimately runs natively (fast!)

```
$ llvm-objdump-13 --section=kprobe/tcp_v4_connect -d examples/tcpconnect/tcpconnect.o

examples/tcpconnect/tcpconnect.o:       file format elf64-bpf

Disassembly of section kprobe/tcp_v4_connect:

0000000000000000 <tcp_v4_connect>:
       0:     79 11 70 00 00 00 00 00 r1 = *(u64 *)(r1 + 112)
       1:     7b 1a f8 ff 00 00 00 00 *(u64 *)(r10 - 8) = r1
       2:     85 00 00 00 0e 00 00 00 call 14
       3:     bf 06 00 00 00 00 00 00 r6 = r0
       4:     63 6a f4 ff 00 00 00 00 *(u32 *)(r10 - 12) = r6
       5:     b7 01 00 00 6c 6c 65 64 r1 = 1684368492
       6:     63 1a d8 ff 00 00 00 00 *(u32 *)(r10 - 40) = r1
       ...
```

eBPF Workflow

- Write some eBPF program code in “C”
- Compile via clang to eBPF assembly/bytecode (stored as ELF file)
- Load programs into kernel, create necessary maps
- Attach programs to desired hookpoints
- Read/analyze data

Loading and Attaching eBPF programs

- Load BPF program into kernel with `BPF(2)` syscall, get back a file descriptor
- Hookpoints depend on the “type” of BPF program
  - (Tracepoints, Kprobes, Socket Filter, XDP, etc.)
- Different system calls/tools used to attach different types of BPF programs
- For example, kprobes can be attached via `perf_event_open(2)` followed by `ioctl(2)`
- “Kprobes enables you to dynamically break into any kernel routine and collect debugging and performance information non-disruptively. You can trap at almost any kernel code address specifying a handler routine to be invoked when the breakpoint is hit.” ~ Kernel documentation

eBPF Maps

- Data structures used to store data gathered from BPF program, share with other programs, etc.
- Various types of maps and data structures available (HashMap, RingBuffer, etc.)
- Can be accessed by BPF programs as well as user space programs
- Create via `BPF(2)` syscall – i.e. can create from userspace

Build an eBPF program

```
104
105  SEC("kprobe/tcp_v4_connect")
106  int BPF_KPROBE(tcp_v4_connect, struct sock *sk)
107  {
108      return enter_tcp_connect(ctx, sk);
109  }
110
111  SEC("kretprobe/tcp_v4_connect")
112  int BPF_KRETPROBE(tcp_v4_connect_ret, int ret)
113  {
114      return exit_tcp_connect(ctx, ret);
115  }
116
```

```
$ clang -target bpf -c tcpconnect.c -o tcpconnect.o
```

```
$ file tcpconnect.o
tcpconnect.o: ELF 64-bit LSB relocatable, eBPF, ...
```

Build an eBPF program

```
$ file tcpconnect.o
tcpconnect.o: ELF 64-bit LSB relocatable, eBPF, ...
```

```
$ llvm-objdump-13 --headers tcpconnect.o

tcpconnect.o:       file format elf64-bpf

Sections:
Idx Name                          Size   VMA          Type
  0                               00000000 0000000000000000
  1 .strtab                       00000166 0000000000000000
  2 .text                         00000000 0000000000000000 TEXT
  3 kprobe/tcp_v4_connect         00000160 0000000000000000 TEXT
  4 .relkprobe/tcp_v4_connect     00000010 0000000000000000
  5 kretprobe/tcp_v4_connect      00000628 0000000000000000 TEXT
  6 .relkretprobe/tcp_v4_connect 00000050 0000000000000000
  7 license                       0000000d 0000000000000000 DATA
  8 .maps                         00000028 0000000000000000 DATA
  9 .maps.counter                 00000038 0000000000000000 DATA
```

Build an eBPF program

```
$ file tcpconnect.o
tcpconnect.o: ELF 64-bit LSB relocatable, eBPF, ...
```

```
$ llvm-objdump-13 --headers tcpconnect.o

tcpconnect.o:       file format elf64-bpf

Sections:
Idx Name                          Size   VMA          Type
  0                               00000000 0000000000000000
  1 .strtab                       00000166 0000000000000000
  2 .text                         00000000 0000000000000000 TEXT
  3 kprobe/tcp_v4_connect         00000160 0000000000000000 TEXT
  4 .relkprobe/tcp_v4_connect     00000010 0000000000000000
  5 kretprobe/tcp_v4_connect      00000628 0000000000000000 TEXT
  6 .relkretprobe/tcp_v4_connect 00000050 0000000000000000
  7 license                       0000000d 0000000000000000 DATA
  8 .maps                         00000028 0000000000000000 DATA
  9 .maps.counter                 00000038 0000000000000000 DATA
```

Load an eBPF program into kernel

```
$ llvm-objdump-13 --headers tcpconnect.o

tcpconnect.o:       file format elf64-bpf

Sections:

  3 kprobe/tcp_v4_connect        00000160 0000000000000000 TEXT
```

```
bpf(BPF_PROG_LOAD, ...program details...)
```

```
bpf_prog_load(BPF_PROG_TYPE_KPROBE, eBPF bytecode..., ...)
```

Verifier

Writing eBPF Programs

- Write instructions directly in eBPF assembly

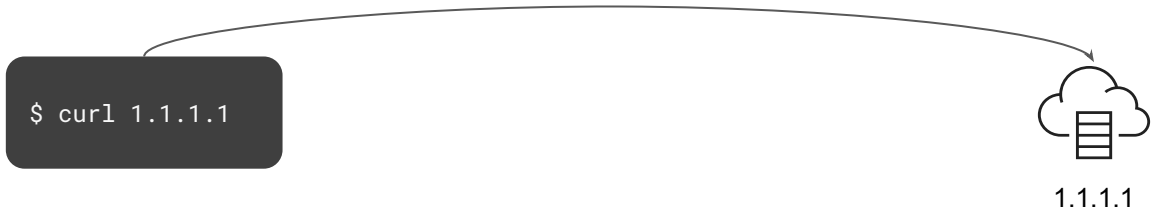
- BCC – https://github.com/iovisor/bcc
  - “BCC makes BPF programs easier to write, with kernel instrumentation in C (and includes a C wrapper around LLVM), and front-ends in Python and lua. It is suited for many tasks, including performance analysis and network traffic control.”
  - One of the original “frameworks” for getting started with eBPF
  - Lots and lots of existing tools and examples

- libbpf – https://github.com/libbpf/libbpf
  - “BPF program code is normally written in the C language with some code organization conventions added to let libbpf make sense of BPF code structure and [...] properly hand everything into the kernel.” ~ https://nakryiko.com/posts/libbpf-bootstrap/
  - BPF portability via BTF & CO-RE

- Other higher-level tools like bpftrace (https://github.com/iovisor/bpftrace)

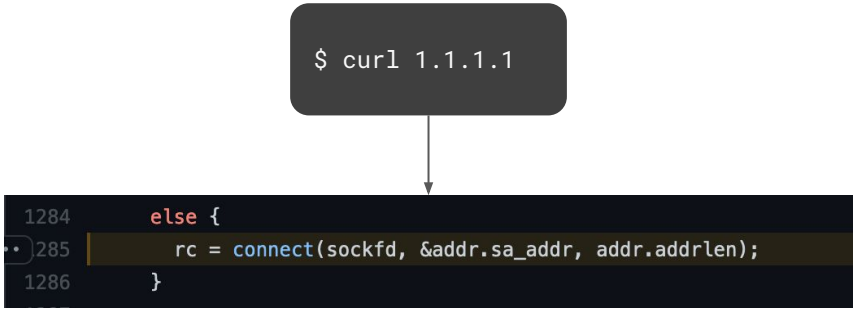
Top-Down View

- Trace a network connection initiated via `curl`



Top-Down View

`curl/lib/connect.c`
Use syscall `connect(2)` on an open socket



Top-Down View

```
$ curl 1.1.1.1
```

`curl/lib/connect.c`
Use syscall `connect(2)` on an open socket

```
1284      else {
1285        rc = connect(sockfd, &addr.sa_addr, addr.addrlen);
1286      }
```

`linux/net/socket.c`
Internally call `connect` function for TCP IPv4

```
1896        err = sock->ops->connect(sock, (struct sockaddr *)address, addrlen,
1897                                 sock->file->f_flags | file_flags);
```

Top-Down View

`curl/lib/connect.c`
Use syscall `connect(2)` on an open socket

```
$ curl 1.1.1.1
```

```
1284      else {
1285        rc = connect(sockfd, &addr.sa_addr, addr.addrlen);
1286      }
```

`linux/net/socket.c`
Internally call `connect` function for TCP IPv4

```
1896        err = sock->ops->connect(sock, (struct sockaddr *)address, addrlen,
1897                                 sock->file->f_flags | file_flags);
```

`linux/net/ipv4/tcp_ipv4.c`
For TCP IPv4, point `connect` to `tcp_v4_connect` function

```
3052  struct proto tcp_prot = {
3053          .name                   = "TCP",
3054          .owner                  = THIS_MODULE,
3055          .close                  = tcp_close,
3056          .pre_connect            = tcp_v4_pre_connect,
3057          .connect                = tcp_v4_connect,
```

Top-Down View

```
$ curl 1.1.1.1
```

`curl/lib/connect.c`
Use syscall `connect(2)` on an open socket

```
1284      else {
1285        rc = connect(sockfd, &addr.sa_addr, addr.addrlen);
1286      }
```

`linux/net/socket.c`
Internally call `connect` function for TCP IPv4

```
1896        err = sock->ops->connect(sock, (struct sockaddr *)address, addrlen,
1897                                 sock->file->f_flags | file_flags);
```

`linux/net/ipv4/tcp_ipv4.c`
For TCP IPv4, point `connect` to `tcp_v4_connect` function

```
3052  struct proto tcp_prot = {
3053          .name                   = "TCP",
3054          .owner                  = THIS_MODULE,
3055          .close                  = tcp_close,
3056          .pre_connect            = tcp_v4_pre_connect,
3057          .connect                = tcp_v4_connect,
```

`linux/net/ipv4/tcp_ipv4.c`
Actual kernel function which will establish the TCP connection

```
197   /* This will initiate an outgoing connection. */
198   int tcp_v4_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)
```

Top-Down View

- Trace a network connection initiated via `curl`


```c
104
105  SEC("kprobe/tcp_v4_connect")
106  int BPF_KPROBE(tcp_v4_connect, struct sock *sk)
107  {
108      return enter_tcp_connect(ctx, sk);
109  }
110
111  SEC("kretprobe/tcp_v4_connect")
112  int BPF_KRETPROBE(tcp_v4_connect_ret, int ret)
113  {
114      return exit_tcp_connect(ctx, ret);
115  }
116
```

Add a kprobe (and kretprobe) to `tcp_v4_connect` to monitor network connections

No application-level instrumentation needed, transparent attachment, extremely performant

Demo with BumbleBee

https://bumblebee.io/
https://github.com/solo-io/bumblebee

solo.io