eBPF on FreeBSD

Hiroki Sato <hrs@FreeBSD.org>

FreeBSD DevSummit 2019.9.19

What is eBPF?

- BPF (Berkeley Packet Filter)
  Steven McCanne, Van Jacobson: "The BSD Packet Filter: A New Architecture for User-level Packet Capture". USENIX, January 1993
- A virtual machine (software processor) designed for packet classification.
- Userland programs can use it via /dev/bpf.

BPF and Userland Applications

- Most of packet capturing software for BSD uses BPF.
  tcpdump (libpcap), dhclient, etc.

- Open /dev/bpf,
  bind an interface using ioctl (BIOCSETIF),
  and load a BPF program using ioctl (BIOCSETF).

- The userland program can read incoming/outgoing packets on the interface.

BPF in Kernel

- `struct ifnet` has `struct bpf_if *if_bpf`
- NIC device drivers call

  `BPF_MTAP(struct bpf_if *, struct mbuf *)`

  at the input and output routine.

- If a BPF file descriptor is attached, data in the mbuf will be evaluated by the registered BPF program.

- `bpf_filter()`

BPF in Kernel

```
hrs@phaleano % sudo tcpdump -d tcp and port 80
(000) ldh      [12]
(001) jeq      #0x86dd          jt 2    jf 8
(002) ldb      [20]
(003) jeq      #0x6             jt 4    jf 19
(004) ldh      [54]
(005) jeq      #0x50            jt 18   jf 6
(006) ldh      [56]
(007) jeq      #0x50            jt 18   jf 19
(008) jeq      #0x800           jt 9    jf 19
(009) ldb      [23]
(010) jeq      #0x6             jt 11   jf 19
(011) ldh      [20]
(012) jset     #0x1fff          jt 19   jf 13
(013) ldxb     4*([14]&0xf)
(014) ldh      [x + 14]
(015) jeq      #0x50            jt 18   jf 16
(016) ldh      [x + 16]
(017) jeq      #0x50            jt 18   jf 19
(018) ret      #262144
(019) ret      #0
```

12-byte offset, load 16 bits

20-byte offset, load 8 bits

54-byte offset, load 16 bits

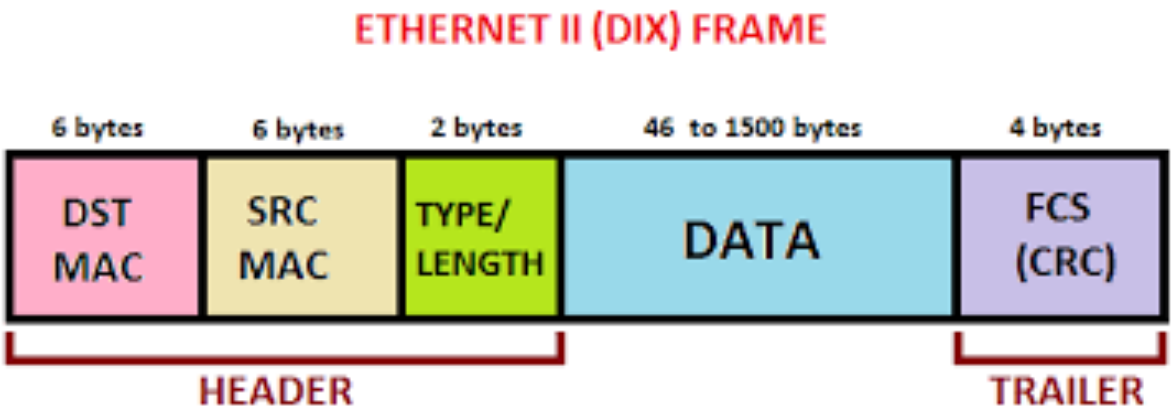
:

:

BPF in Kernel

```
hrs@phaleano % sudo tcpdump -d tcp and port 80
(000) ldh      [12]
(001) jeq      #0x86dd          jt 2    jf 8
(002) ldb      [20]
(003) j
(004) l
(005) j
(006) l
(007) j
(008) j
(009) l
(010) j
(011) l
(012) j
(013) l
(014) l
(015) jeq      #0x50            jt 18   jf 16
(016) ldh      [x + 16]
(017) jeq      #0x50            jt 18   jf 19
(018) ret      #262144
(019) ret      #0
```

12-byte offset, load 16 bits



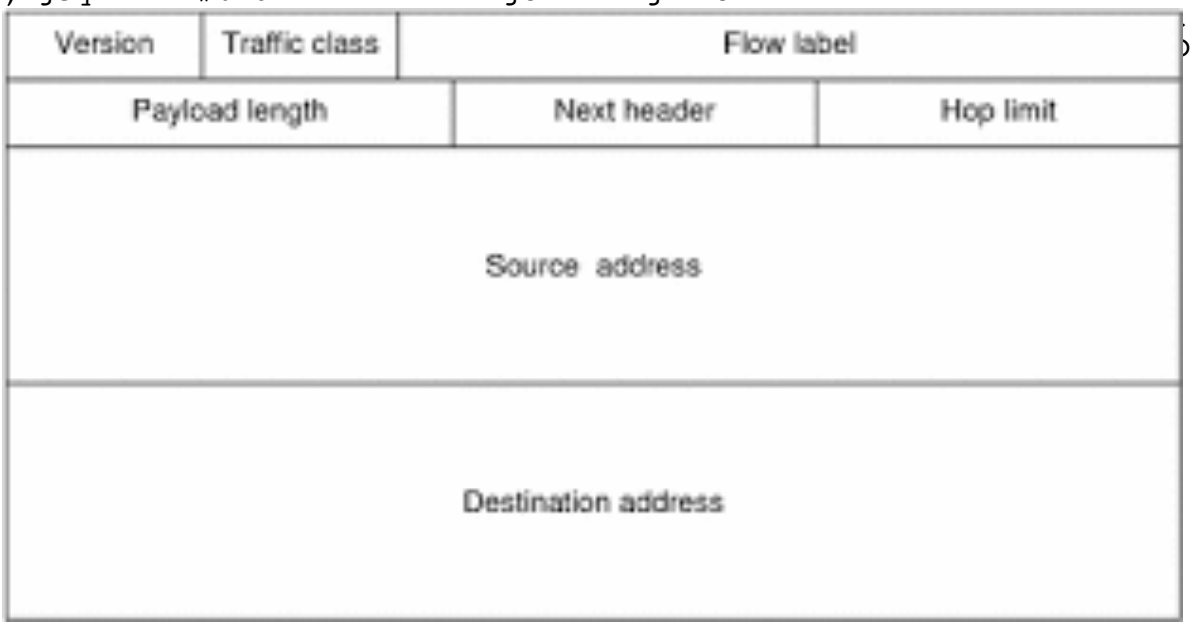
BPF in Kernel

```
hrs@phaleano % sudo tcpdump -d tcp and port 80
(000) ldh      [12]
(001) jeq      #0x86dd          jt 2    jf 8
(002) ldb      [20]
(003) jeq      #0x6             jt 4    jf 19
(004) ldh      [54]
(005) jeq      #0x50            jt 18   jf 6
(006) ldh      [56]
(007) jeq      #0x50            jt 18   jf 10
(008) jeq      #0x1bb           jt 18   jf 9
(009) ldb      [62]
(010) jeq      #0x6             jt 11   jf 12
(011) ldh      [60]
(012) jset     #0x1fff          jt 18   jf 13
(013) ldxb     4*([62]&0xf)
(014) ldh      [x + 54]
(015) jeq      #0x50            jt 17   jf 16
(016) ldh      [x + 56]
(017) jeq      #0x1bb           jt 17   jf 18
(018) ret      #262144
(019) ret      #0
```

12-byte offset, load 16 bits

20-byte offset, load 8 bits

54-byte offset, load 16 bits



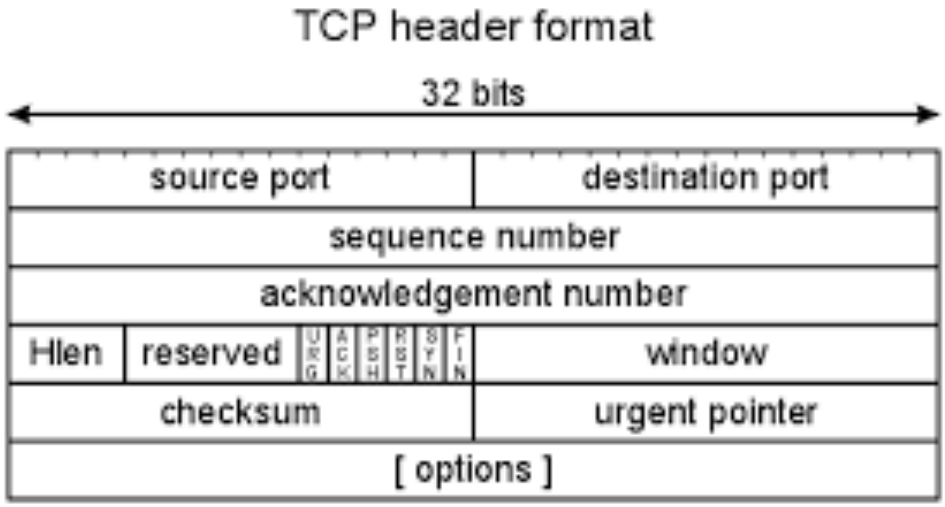
BPF in Kernel

```
hrs@phaleano % sudo tcpdump -d tcp and port 80
(000) ldh      [12]
(001) jeq      #0x86dd          jt 2    jf 8
(002) ldb      [20]
(003) jeq      #0x6             jt 4    jf 19
(004) ldh      [54]
(005) jeq      #0x50            jt 18   jf 6
(006) ldh      [56]
(007) jeq
(008) jeq
(009) ldb
(010) jeq
(011) ldh
(012) jset
(013) ldxb
(014) ldh
(015) jeq
(016) ldh
(017) jeq
(018) ret
(019) ret
```

12-byte offset, load 16 bits

20-byte offset, load 8 bits

54-byte offset, load 16 bits



BPF in Kernel

```
hrs@phaleano % sudo tcpdump -dd tcp and port 80
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 6, 0x000086dd },
{ 0x30, 0, 0, 0x00000014 },
{ 0x15, 0, 15, 0x00000006 },
{ 0x28, 0, 0, 0x00000036 },
{ 0x15, 12, 0, 0x00000050 },
{ 0x28, 0, 0, 0x00000038 },
{ 0x15, 10, 11, 0x00000050 },
{ 0x15, 0, 10, 0x00000800 },
{ 0x30, 0, 0, 0x00000017 },
{ 0x15, 0, 8, 0x00000006 },
{ 0x28, 0, 0, 0x00000014 },
{ 0x45, 6, 0, 0x00001fff },
{ 0xb1, 0, 0, 0x0000000e },
{ 0x48, 0, 0, 0x0000000e },
{ 0x15, 2, 0, 0x00000050 },
{ 0x48, 0, 0, 0x00000010 },
{ 0x15, 0, 1, 0x00000050 },
{ 0x6, 0, 0, 0x00040000 },
{ 0x6, 0, 0, 0x00000000 },
```


BPF in Kernel

- `bpf_filter()` is an interpreter of the BPF instructions. 32-bit register x 16 + ACC + Index reg.
- The entry points are limited: BPF_MTAP() calls bpf_filter() to classify data in the mbufs.
- There is no API to call a BPF program, or to call a kernel function from a BPF program.

eBPF

- An extended version of BPF virtual machine (Linux 4.16 or later).
- 64-bit registers x 10, 512 bytes stack, and key-value store. A function call instruction.
- General purpose, not limited to as a packet classifier---it can communicate with other subsystem or userland via the key-value store or function call.
- Addresses for the function calls and key-value store are resolved at loading time.

eBPF function call

- Function pointers are relocated at loading time.
- Although "call" instruction has a jump target address, eBPF interpreter does not allow an arbitrary address.
- Limited to "pre-registered kernel helper functions".

eBPF Map

- Key-value store data structure which can be accessed by eBPF program, kernel subsystem, and userland program. Statically pre-allocated before running a program.
- Array, Hash, address table, etc
- eBPF programs use kernel helper functions to access them. Other kernel subsystems can read/write data by using the same function set.
- Userland program can read/write the data structure by ioctl(2), which eventually calls the same function set.

eBPF on FreeBSD

- I and Yutaro Hayakawa, GSoC student in 2018, have maintained an implementation for FreeBSD at https://github.com/YutaroHayakawa/generic-ebpf
  - [Current status]
    - eBPF interpreter and JIT compiler
    - Map helper functions
    - Userland interface (/dev/ebpf and ioctls)
- Still lacking of...
  - hooks in kernel.
  - helper functions other than map access.

Use of eBPF

Development

- eBPF in assembly language is pain. An eBPF program can be written in a C-like language:
  make WITH_LLVM_TARGET_BPF=yes buildworld (?)
  https://reviews.freebsd.org/D16033

- `% clang80 -target bpf -c -o foo.o foo.c`

- ELF sections for eBPF maps and hints for relocation.


```
hrs@phaleano % cat foo.c

#include <stdint.h>

uint64_t glob64;

int
main(void)
{
        int a;

        a = 2;

        return (a + 1);
}
```

```
hrs@phaleano % llvm-objdump80 -x --disassemble foo.o

foo.o:  file format ELF64-BPF

architecture: bpfel
start address: 0x0000000000000000

Disassembly of section .text:
0000000000000000 main:
       0:       b7 01 00 00 00 00 00 00         r1 = 0
       1:       63 1a fc ff 00 00 00 00         *(u32 *)(r10 - 4) = r1
       2:       b7 01 00 00 02 00 00 00         r1 = 2
       3:       63 1a f8 ff 00 00 00 00         *(u32 *)(r10 - 8) = r1
       4:       61 a1 f8 ff 00 00 00 00         r1 = *(u32 *)(r10 - 8)
       5:       07 01 00 00 01 00 00 00         r1 += 1
       6:       bf 10 00 00 00 00 00 00         r0 = r1
       7:       95 00 00 00 00 00 00 00         exit
Sections:
Idx Name          Size      Address          Type
  0               00000000 0000000000000000
  1 .strtab       00000045 0000000000000000
  2 .text         00000040 0000000000000000 TEXT
  3 .BTF          00000019 0000000000000000
  4 .BTF.ext      00000020 0000000000000000
  5 .llvm_addrsig 00000000 0000000000000000
  6 .symtab       00000060 0000000000000000
SYMBOL TABLE:
```


Use of eBPF

Entry Points

- An eBPF program can be invoked in kernel: ebpf_vm_alloc(), ebpf_load_prog(), and ebpf_vm_run()
- With helper functions, a userland program can inject an eBPF program, make it do something in kernel, and get a result via map data structure.
- It is possible to add a hook into the existing execution path just like DTrace's probe.

Benefits

- **Q:** Why a new vm/language? Developing a kernel module in C language looks easier.

- **A:** It works well where flexibility in kernel depending on data is required:
  - Packet filtering based on rules,
  - Patching a syscall from userland for security purpose
  - L2 packet forwarding (learning bridge) w/o entering the kernel network stack,
  - cf. Netgraph subsystem: collections of klds to process mbufs.

Discussions (1)

- Actual use cases? Linux uses eBPF as
  - a packet processing state machine.
  - a tool for tracing and filtering syscalls.
    - bpftrace: ebpf-backed DTrace clone

- Entry points and helper functions for eBPF programs in kernel depends on the application:
  - Possible to attach to socket, if_input/output, syscall entry, etc.
  - Processing involving "traversing a tree and compare"--- e.g. compiling IPsec SPD/SAD into eBPF code.

Discussions (2)

- Safety of an eBPF program. Linux implements a code verification based on simulating execution paths and pointer arithmetics.

- Program compatibility with Linux eBPF?
  - Same kernel helper functions and map structures.
  - It uses (struct sk_buff)-specific helper functions.