BPF packet capture

Linux Plumbers 2019

Alan Maguire
Linux Kernel Networking, Oracle
September 9, 2019

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, timing, and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

Program Agenda

1. The roots of BPF - tap + filter
2. How the tap works (libpcap)
3. Helping BPF programs to tap packets – bpf_pcap()?
4. Capturing packets with bpftool?
5. Future work

Program Agenda

The roots of BPF

- BPF was born in a time of crisis
- The genesis of this work by the LBNL (Lawrence Berkeley National Labs) Network Research Group was diagnosing and fixing congestion collapse of the ARPAnet
- The tools of the day performed poorly and didn't do much in the way of protocol decoding or filtering
- Some of the tools they built: tcpdump, libpcap, traceroute, pathchar…

  (libpcap: An Architecture and Optimization Methodology for Packet Capture   : Sharkfest '11)

The roots of BPF

https://www.tcpdump.org/papers/bpf-usenix93.pdf

- When observing traffic, we generally want a subset only (tcp port 80)
- A flexible way of describing filters was required
- Now termed “classic” BPF, filters were translated into a simple instruction/register set used to specify if packets are to be rejected, accepted (and if so how much of the packet)
- A simple VM ran the instructions in-kernel and filtered appropriately
- Critically, safety was the key criterion when injecting filter code. All programs must complete in a bounded time (no loops)

The roots of BPF

To see classic BPF, run tshark or tcpdump with -d option to see cBPF implementation of filter, e.g.:

```
# tcpdump -i wlp4s0 -d 'tcp'
(000) ldh      [12]
(001) jeq      #0x86dd          jt 2    jf 7
(002) ldb      [20]
(003) jeq      #0x6             jt 10   jf 4
(004) jeq      #0x2c            jt 5    jf 11
(005) ldb      [54]
(006) jeq      #0x6             jt 10   jf 11
(007) jeq      #0x800           jt 8    jf 11
(008) ldb      [23]
(009) jeq      #0x6             jt 10   jf 11
(010) ret      #65535
(011) ret      #0t
```

The roots of BPF

- But we need something to filter!
- The original BPF paper terms the packet source a “tap”
- The eBPF revolution was started by taking that filtering language, extending it and expanding the applications from filtering to fast packet processing (XDP), tracing (kprobes/tracing) and more!
- Similarly, here we're going to be looking at extending the concept of a tap from a fixed point in the kernel to something more dynamic, using BPF as the means to do so.

Program Agenda

How the tap works

- It's worth first describing how observability works today when using wireshark/tcpdump.
- Let's use libpcap as an example; to follow along look in the pcap-linux.c code in libpcap (the kernel side is described in Documentation/networking/packet_mmap.txt)
  - https://github.com/the-tcpdump-group/libpcap/tree/master/pcap
- It first opens a PF_PACKET/SOCK_RAW socket (activate_new() function)
- then binds the socket using a "struct sockaddr_ll" - a link-layer

  socket address containing family AF_PACKET, ifindex, and protocol

  (iface_bind())

How the tap works

- In the kernel this bind action results in a call to dev_add_pack() which adds the packet socket protocol handler to networking stack. this is the key step as with the handler in place the generic send and receive routines will deliver skbs to our packet socket for processing.
- switches on promiscuous mode if requested, pushes a filter if specified
- Once the socket is open, attempt to activate mmap access to packets (activate_mmap())
  - PACKET_MMAP uses a shared kernel/userspace ring buffer eliminating the need for system calls to read each packet. Previously libpcap required two system calls per-packet - one to read, another to get the capture time. Now a poll() can be used and multiple packets can be gathered at a time.
- Now we can start reading packets; see pcap_read_linux_mmap_*().

Where the tap operates

- Data is passed to the tap socket at device-agnostic layer on send, receive
  - see net/core/dev.c
- Problem is, a lot happens before and after these events (GSO, GRO, drops etc). With drops the packet may not even reach the tap.
- When debugging, there is no one right answer to the question "where is the best place to probe for packets?"
  - I only want drops at kfree_skb()!
  - I want to see packets before/after GSO segmentation/GRO reassembly!
  - …

Program Agenda

1. The roots of BPF – tap + filter
2. How the tap works (libpcap)
3. Helping BPF programs to tap packets – bpf_pcap()?
4. Capturing packets with bpftool?
5. Future work

Talking to user-space from BPF programs

- BPF programs utilize helper functions to carry out tasks on their behalf
- Some helpers facilitate user-space communication
  - bpf_trace_printk() - printk for BPF programs
  - bpf_map_* APIs – lookup, update map data which is shared with user-space
  - **bpf_perf_event_output – record metadata + information from BPF programs**
- The latter helper works for
  - skb programs (tc, sk_skb etc)
  - xdp programs
  - Tracing programs (k[ret]probe, [raw_]tracepoint)
- xdpcap uses perf events to capture from xdp (https://blog.cloudflare.com/xdpcap/)

Proposing a bpf_pcap helper

- Perf events are written to an mmap'ed shared buffer with userspace
- We can poll for updates and read multiple events simultaneously without invoking multiple system calls (sound familiar?)
- Propose using the same mechanism for a bpf_pcap helper;
  - Metadata is information needed to describe captured packet
    - Protocol type
    - Packet length, capture length
    - Capture time
  - Data is packet itself
- As is the case for bpf_perf_event_output, support xdp, skb and tracing programs.

Proposing a bpf_pcap helper

*int bpf_pcap(void *data, __u32 max_len, void *map, int protocol, __u64 flags);*

*Send a perf event containing capture metadata and up to max_len bytes of the packet associated with **data**.*

- *data* : First argument is packet we want to trace
  - For skb programs, it is the context of the program (struct __sk_buff *)
  - Fox xdp programs, it is the xdp metadata (struct xdp_buff *)
  - For tracing programs, it is a pointer to a struct sk_buff derived from the kprobe context (struct pt_regs *), or a raw tracepoint argument, etc
    - bpf_pcap(PT_REGS_PARM1(ctx), ...) captures the skb data pointed to by the first argument to the function we've kprobe-d

Proposing a bpf_pcap helper

*int bpf_pcap(void *data, __u32 max_len, void *map, int protocol, __u64 flags);*

- *max_len* : Second argument limits the amount of data we capture
- *map* : Third argument is the perf_event_array map containing file descriptors indexed by CPU allowing the perf event to be stored in the appropriate mmap'ed area.
- *protocol* : The link type to be used in the capture – see pcap-linktype (7)
  - This is the starting protocol, and should match the protocol of the packet pointed to by skb→data (xdp→data)
  - 1 for ethernet, 12 for IP (v4 or v6), etc.

Proposing a bpf_pcap helper

*int bpf_pcap(void *data, __u32 max_len, void *map, int protocol, __u64 flags);*

- *flags* : Final argument
  - Provides 48 bits of identifier. Idea is this will allow us to associate other tracing data with the captured packet in userspace if needed
    - namespace ids
    - process ids
    - stackmap indices
  - Other bits can be used to identify the identifier

Proposing a bpf_pcap helper

struct bpf_pcap_hdr is used as metadata in the perf event triggered at capture time

- It contains
  - a magic number BPF_PCAP_MAGIC which identifies the perf event as a pcap-related event.
  - a starting protocol is the protocol associated with the header
  - the id value, derived from the flags value passed into bpf_pcap
  - packet length and amount of data captured
  - time in ns of capture

Proposing a bpf_pcap helper

- *struct bpf_pcap_hdr* can also be re-used to share user-space-driven configuration for capture with BPF programs, allowing us to configure capture dynamically.
- Recall it contains
  - desired starting protocol for capture (*protocol* argument)
  - Maximum length to be used for capture (cap_*len* argument)
  - A flags value which can be used to control program behaviour, specifically
    - specify incoming interface to restrict tracing to

Capture IP packets from skb arg1 of kernel functions

```
struct bpf_map_def SEC("maps") pcap_data_map = {
        .type = BPF_MAP_TYPE_PERF_EVENT_ARRAY,
        .key_size = sizeof(int),
        .value_size = sizeof(int),
        .max_entries = 1024,
};
SEC("kprobe/pcap_arg1")
int pcap_arg1(struct pt_regs *ctx) {
      bpf_pcap((void *)PT_REGS_PARM1(ctx), 2048,
                         &pcap_data_map, BPF_PCAP_TYPE_IP, 0);
}
```

Program Agenda

Proposing bpftool pcap

- Aim: give users a ready-made tool to capture packet data from BPF programs.
- bpftool (8) describes itself as a
  - "tool for inspection and simple manipulation of eBPF programs and maps"
- bpftool is designed to help debug BPF programs
- Packet capture can help debug (e.g. checksum errors); and it does so (in large part) by manipulating programs and maps.
- A new subcommand "pcap" for bpftool?
- Requires libpcap + headers (libpcap-devel package on some distros) at build time, libpcap at runtime.

Proposing bpftool pcap

*bpftool pcap **prog** {id ID | pinned PATH } [proto PROTOCOL] [len MAX_LEN]*
[pages NUMPAGES]

- Scan program for perf event map and optionally array map for config specification
- Create mmaps (of NUMPAGES pages per CPU) and set them in perf event array
- If config map found, set desired protocol, max length as BPF program may retrieve values from map to determine these values
- Open pcap/pcap_dumper
- Poll for perf events, dump packets

ORACLE

Proposing bpftool pcap

bpftool pcap ***trace*** *OBJECT [kprobe|tracepoint]:PROBE[:arg[1-4]]*

[proto PROTOCOL] [len MAX_LEN] [dev DEVNAME] [pages NUMPAGES]

- Load BPF object (or use default kprobe/tracepoint object supplied with bpftool if none is specified)
- Set configuration options
  - which argument to capture (arg1, arg2, arg3, arg4), kprobe:kfree_skb:arg1
  - what the starting protocol is; etc
- Attach to specified tracepoint/kprobes and trace from associated perf event map

Proposing bpftool pcap

- Propose supplying a kprobe and raw_tracepoint program with bpftool which can be used if no tracing program is specified
- Deliver bpftool_pcap_probe.o, bpftool_pcap_tracepoint.o objects with bpftool
- This gives users functionality out of the box. For example:

Examples: Observing Drops

Tracing dropped packets at kfree_skb(). We trigger a drop artificially by adding an iptables DROP rule and triggering it.

```
$ iptables -A INPUT -p tcp --dport 6666 -j DROP

$ bpftool pcap trace kprobe:kfree_skb proto ip data_out /tmp/cap &

$ nc 127.0.0.1 6666

Ncat: Connection timed out.

$ fg

^C

$ tshark -r /tmp/cap

Running as user "root" and group "root". This could be dangerous.

…

  3        7    127.0.0.1 -> 127.0.0.1    TCP 60 54732 > ircu [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=696475539 TSecr=0 WS=128

…
```

Examples: Observing WiFi

Tracing WiFi Scans

```
$ bpftool pcap trace kprobe:ieee80211_scan_rx:arg2 proto ieee802_11 |tcpdump -r -
reading from file -, link-type IEEE802_11 (802.11)
07:57:06.998832 Beacon (wifi_net1) [1.0* 2.0* 5.5* 11.0* 18.0 24.0 36.0 54.0 Mbit] ESS CH: 8, PRIVACY
07:57:15.376283 Probe Response (wifi_net1) [1.0* 2.0* 5.5* 11.0* 6.0 9.0 12.0 18.0 Mbit] CH: 1, PRIVACY
07:57:15.385763 Beacon (wifi_net2) [1.0* 2.0* 5.5* 11.0* 6.0 9.0 12.0 18.0 Mbit] ESS CH: 1, PRIVACY
07:57:15.712469 Probe Response (wifi_net2) [1.0* 2.0* 5.5* 11.0* 18.0 24.0 36.0 54.0 Mbit] CH: 8, PRIVACY
07:57:16.016965 Beacon (wifi_net3) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit] ESS, PRIVACY
07:57:16.134790 Probe Response (wifi_net3) [6.0* 9.0 12.0* 18.0 24.0* 36.0 48.0 54.0 Mbit], PRIVACY
```

Program Agenda

Future work

- Add packet capture support to BPF tracing programs (bpftrace, ply)
- pcap-ng packet capture format is interesting for this use case
  - Allows mixing of protocol types in a single capture file
  - Allows “annotation/comments” to packets in a capture
- We didn't even talk about filtering!
  - Mechanism to enable filtering in programs before capture?
    - bpf2bpf calls? Similar to bpf_tail_call() but calling a bpf filter program via helper function so we can evaluate the result and filter appropriately?
- Support tracing for more than SKBs? RDMA stack? USB subsystem?

References

McCanne, S. (2011) Keynote Presentation:The Architecture and Optimization Methodology of the libpcap Packet Capture Library. SharkFest '11

http://sharkfest.wireshark.org/sharkfest.11/presentations/McCanne-Sharkfest'11_Keynote_Address.pdf

https://www.youtube.com/watch?v=XHlqIqPvKw8

McCanne, S., Jacobson, V. (1993) The BSD Packet Filter: A New Architecture for User-level Packet Capture. Usenix https://www.tcpdump.org/papers/bpf-usenix93.pdf

Fabre, A. (2019) xdpcap: XDP Packet Capture. https://blog.cloudflare.com/xdpcap/

BPF packet capture RFC patchset https://lore.kernel.org/bpf/1567892444-16344-1-git-send-email-alan.maguire@oracle.com/

ORACLE®