XDP – eXpress Data Path

Used for DDoS protection

Linux Kernel self protection

Learn writing eBPF code

Jesper Dangaard Brouer
Principal Engineer, Red Hat

OpenSourceDays
March, 2017

Audience: Prepare yourself!

- Git clone or fork
  - https://github.com/netoptimizer/prototype-kernel/
- XDP eBPF code example in directory:
  - kernel/samples/bpf/
- Documentation
  - https://prototype-kernel.readthedocs.io/
    - Notice two “sections”
      - XDP - eXpress Data Path
      - eBPF - extended Berkeley Packet Filter

Overview

- What is XDP – eXpress Data Path
- Using XDP for DDoS protection
  1) Linux Kernel self protection
  2) Handling volume attacks with scrubbing
- Learn to write eBPF XDP code by examples
  - Blacklist example ready to use
  - Modify and adapt to DDoS attacks

Introduction

- An eXpress Data Path (XDP) in kernel-space
  - The "packet-page" idea from NetDev1.1 "rebranded"
  - Thanks to: Tom Herbert, Alexei and Brenden Blanco, putting effort behind idea
  - Basic idea: work on raw packet-page inside driver
    - Hook before any allocations or normal stack overhead
- Performance is primary focus and concern
  - Target is competing with DPDK speeds
  - No fancy features!
    - Need features: use normal stack delivery

XDP: What is XDP (eXpress Data Path)?

- Thin layer at lowest levels of SW network stack
  - Before allocating SKBs
  - Inside device drivers RX function
  - Operate directly on RX packet-pages
- XDP is NOT kernel bypass
  - Designed to work in concert with stack
- XDP - run-time programmability via "hook"
  - Run eBPF program at hook point
  - Learn writing eBPF code later...
    - User-defined, sandboxed bytecode executed by the kernel

XDP: data-plane responsibility “split”

- Abstract “data-plane” view of XDP
- Split between kernel and eBPF
  - **Kernel:**
    - Fabric in charge of moving packets quickly
  - **eBPF:**
    - Policy logic decide action
    - Read/write access to packet

XDP: Performance evaluation, crazy fast!!!

- Evaluated on Mellanox 40Gbit/s NICs (mlx4)
  - Single CPU with DDIO performance
    - 20 Mpps – Filter drop all (but read/touch data)
    - 12 Mpps – TX-bounce forward (TX bulking)
    - 10 Mpps – TX-bounce with udp+mac rewrite
  - Single CPU without DDIO (cache-misses)
    - TX-bounce with udp+mac rewrite:
      - 8.5Mpps – cache-miss
      - 12.3Mpps – RX prefetch loop trick
    - RX cache prefetch loop trick: 20 Mpps XDP_DROP

XDP: Packet based

- Packet based decision
  - (Currently) cannot store/propagate meta per packet
  - eBPF program can build arbitrary internal state (maps/hashes)
- Got **write** access to raw packet
  - Use-cases for modifying packets:
    - Add or pop encapsulation headers
    - Rewrite packet headers for forwarding/bouncing

XDP: Disclaimer

- Enabling XDP changes (RX ring) *memory model*
  - Waste memory: Always alloc 4K (page) per RX packet
  - Needed to get write access to packet
  - Needed for fast drop (simple RX ring recycling)
- In theory cause small performance regression
  - When delivering packets to normal network stack
  - Due to bottleneck in page allocator
    - Working on page_pool project to remove this bottleneck
      - PoC code shows, faster than before!
  - Memory model “waste” can affect TCP throughput
    - Due to affecting skb->truesize

XDP: NIC hardware and driver dependency

- Not all NIC drivers support XDP
  - Software dependency:
    - expect list of NICs to increase quickly
- List of currently (v4.10) supported NIC drivers
  - Mellanox: mlx4 + mlx5
  - Netronome: nfp
  - Cavium/Qlogic: qede
  - virtio-net

XDP – basic actions or verdicts

- Currently only implement 3 basic action
  1) XDP_PASS:
     - Pass into normal network stack (could be modified)
  2) XDP_DROP:
     - Very fast drop (recycle page in driver)
  3) XDP_TX:
     - Forward or TX-bounce back-out same interface
- XDP_TX "TX-bounce" seems limiting, but useful for
  - DDoS scrubber service
  - One-legged load-balancer (Facebook)

Kernel eBPF samples: General file structure

- General eBPF samples avail in kernel tree (incl XDP)
- Kernel samples/bpf are split into files for
  - eBPF code running kernel-side
    - Name convention: xxxxx_**kern**.c
      - Restricted C-code, transformed to eBPF code
    - Output ELF file with eBPF-obj code: xxxxx_**kern**.**o**
  - Userspace code loading and interacting with eBPF
    - Name convention: xxxxx_**user**.c
    - Executable named: xxxxx
      - Will load file xxxxx_**kern.o** into kernel
    - And optionally: xxxxx_**cmdline**.c

This talks focus: DDoS use-case

- Two scenarios:

1) Linux Kernel self protection
   - Cite: Thomas Graf:
     - "Empower Linux kernels to self protect in exposed environments"
2) Handling volume attacks with scrubbing
   - Deploy Linux machines to filter traffic
   - Near network edges and ISP transit points
   - (Uses action: XDP_TX)

Ready to use XDP eBPF examples

- Git clone or fork
  - https://github.com/netoptimizer/prototype-kernel/
- XDP eBPF code example in dir: kernel/samples/bpf/
- Based on kernel samples/bpf/
  - Allow out-of-kerne-tree compiling
    - And keeping track of your own git commit history
  - Still depend on kernel source for compiling
    - As many distros lack UAPI headers for eBPF

Dependencies for eBPF examples

- eBPF examples written in restricted-C
  - Requires compilers with eBPF support
    - Clang >= version 3.4.0
    - LLVM >= version 3.7.1
- Tested on Fedora 25
  - Works with disto kernel: 4.9.3-200.fc25.x86_64
  - Have: LLVM (3.8.1) + clang (3.8.0)

Documentation

- Follow documentation at
  - https://prototype-kernel.readthedocs.io/
  - Notice two “sections”
    - XDP - eXpress Data Path
    - eBPF - extended Berkeley Packet Filter
- Plan: merge doc into kernel
  - Kernels new documentation format:
    - https://www.kernel.org/doc/html/latest/

Benchmark your NIC hardware

- Sample: `xdp_bench01_mem_access_cost`
  - Purely benchmark NIC and CPU hardware limits
    - Measure cost of touching vs. not-touching packet memory
- Measure max XDP performance of NIC
  - Run: `./xdp_bench --readmem` (default XDP_DROP)
- Measure cost of enabling XDP for normal netstack
  - Run: `./xdp_bench --action XDP_PASS --readmem`
- Use as baseline: Against your own eBPF code
  - To assess cost of your eBPF program

Code: `xdp_bench01_mem_access_cost_kern.c`

Simple benchmark program

L1: SEC must start with “xdp”

L4-5: Packet ptr data + data_end

L11: Validate len, no mem touch

L16: Map determine XDP action

(used in L31)

L22: Support touch/read memory

L28-30: RX pkt counter via map

```
 1 SEC("xdp_bench01")
 2 int xdp_prog(struct xdp_md *ctx)
 3 {
 4   void *data_end = (void *)(long)ctx->data_end;
 5   void *data = (void *)(long)ctx->data;
 6   struct ethhdr *eth = data;
 7   volatile u16 eth_type;
 8   long *value; u64 offset; u32 key = 0; int *action; u64 *touch_mem;
 9
10   /* Validate packet length is minimum Eth header size */
11   offset = sizeof(*eth);
12   if (data + offset > data_end)
13           return XDP_DROP;
14
15   /* Allow userspace to choose XDP_DROP or XDP_PASS */
16   action = bpf_map_lookup_elem(&xdp_action, &key);
17   if (!action)
18           return XDP_DROP;
19
20   /* Default: Don't touch packet data, only count packets */
21   touch_mem = bpf_map_lookup_elem(&touch_memory, &key);
22   if (touch_mem && (*touch_mem == 1)) {
23           /* ... code removed for brevity */
24           eth_type = eth->h_proto;
25   }
26
27   value = bpf_map_lookup_elem(&rx_cnt, &key);
28   if (value)
29           *value += 1;
30
31   return *action;
32 }
```


Maps: `xdp_bench01_mem_access_cost_kern.c`

- Maps essential part of the eBPF toolbox
  - Counter, tables, control program
  - Even support XDP action from map

Code notes:

Definition order in kern.c, translate to map_fd[i] order in xxx_user.c code

```
 1 struct bpf_map_def SEC("maps") rx_cnt = {
 2   .type = BPF_MAP_TYPE_PERCPU_ARRAY,
 3   .key_size = sizeof(u32),
 4   .value_size = sizeof(long),
 5   .max_entries = 1,
 6 };
 7
 8 struct bpf_map_def SEC("maps") xdp_action = {
 9   .type = BPF_MAP_TYPE_ARRAY,
10   .key_size = sizeof(u32),
11   .value_size = sizeof(long),
12   .max_entries = 1,
13 };
14
15 struct bpf_map_def SEC("maps") touch_memory = {
16   .type = BPF_MAP_TYPE_ARRAY,
17   .key_size = sizeof(u32),
18   .value_size = sizeof(long),
19   .max_entries = 1,
20 };
```


Basic XDP blacklist: `xdp_ddos01_blacklist`

- Most simple filter: IPv4 blacklist facility
  - Practical use-case for non-spoofed traffic
- Features demonstrated in sample
  - Using hash-table maps
    - BPF_MAP_TYPE_PERCPU_HASH
  - Exporting eBPF map files to filesystem
    - Allow unprivileged users access (via chown)
  - Separate cmdline tool for manipulation maps
    - Basic: IP add+del, listing, stats
  - Reloading xxx_kern.o keeping maps intact

Tool: `xdp_ddos01_blacklist`

```
$ ./xdp_ddos01_blacklist --help
DOCUMENTATION:
 XDP: DDoS protection via IPv4 blacklist

This program loads the XDP eBPF program into the kernel.
Use the cmdline tool for add/removing source IPs to the blacklist
and read statistics.

 Usage: ./xdp_ddos01_blacklist (options-see-below)
 Listing options:
 --help          short-option: -h
 --remove        short-option: -r
 --dev           short-option: -d
 --quite         short-option: -q
 --owner         short-option: -o
```


Tool: `xdp_ddos01_blacklist`

Load eBPF code into kernel

- and give current $USER ownership of exported map files.

```
$ sudo ./xdp_ddos01_blacklist --dev mlx5p4 --owner $USER
Documentation:
 XDP: DDoS protection via IPv4 blacklist

This program loads the XDP eBPF program into the kernel.
Use the cmdline tool for add/removing source IPs to the blacklist
and read statistics.

 - Attached to device:mlx5p4 (ifindex:5)
 - Blacklist     map file: /sys/fs/bpf/ddos_blacklist
 - Verdict stats map file: /sys/fs/bpf/ddos_blacklist_stat_verdict
```


Blacklist performance difference

- Single CPU benchmark (with single flow)
  - Very fast CPU: i7-6700K CPU @ 4.00GHz
  - NIC 50Gbit/s Mellanox-CX4 (driver: mlx5)
- Delivery to closed UDP port (after commit 9f2f27a9):
  - UdpNoPorts 1,320,446 pps (conntrack issue)
  - UdpNoPorts 3,143,931 pps (unloaded iptables)
- Fastest iptables drop in “raw” table:
  - `iptables -t raw -I PREROUTING -p udp --dport 9 -j DROP`
  - Drop: 4,522,956 pps (prefetch 4,748,646 pps)
- XDP blacklist: (prefetch trick in mlx5 not upstream)
  - Drop: 9,697,564 pps (prefetch 16,939,941 pps)

Tool: xdp_ddos01_blacklist_cmdline

```
$ ./xdp_ddos01_blacklist_cmdline --help
DOCUMENTATION:
 XDP ddos01: command line tool

 Usage: ./xdp_ddos01_blacklist_cmdline (options-see-below)
 Listing options:
 --help          short-option: -h
 --add           short-option: -a
 --del           short-option: -x
 --ip            short-option: -i
 --stats         short-option: -s
 --sec           short-option: -s
 --list          short-option: -l

$ ./xdp_ddos01_blacklist_cmdline --add --ip 1.2.3.4
blacklist_modify() IP:1.2.3.4 key:0x4030201
```


Tool: `xdp_ddos01_blacklist_cmdline`

- Reading stats
  - “Attack” 8 src-IPs, and 7 blocked
    - Tool: `pktgen_sample05_flow_per_thread.sh`

```
$ ./xdp_ddos01_blacklist_cmdline --stats

XDP_action   pps        pps-human-readable period/sec
XDP_ABORTED  0          0                  1.000089
XDP_DROP     30463237   30,463,237         1.000089
XDP_PASS     3438094    3,438,094          1.000089
XDP_TX       0          0                  1.000089
```


Map type: `BPF_MAP_TYPE_PERCPU_HASH`

- Blacklist program uses hash map
  - Hash calc does come with a CPU cost
  - Percpu variant for lockless (kern-side)

Defined in _kern.c like this:

Key is IP (32bit)

Value(64bit) count drops

Limit max entries

Don't prealloc elements,

```
1 struct bpf_map_def SEC("maps") blacklist = {
2   .type        = BPF_MAP_TYPE_PERCPU_HASH,
3   .key_size    = sizeof(u32),
4   .value_size  = sizeof(u64), /* Drop counter */
5   .max_entries = 100000,
6   .map_flags   = BPF_F_NO_PREALLOC,
7 };
```

assume add/del of elements is rare/not-frequent, and only happens from userspace.

eBPF _kern.c side map-lookup

- Accessing map from _kern.c eBPF code

Code example:

L20-23: Extract key. Validate length before accessing: iph->saddr (else kernel validator will reject ebpf code)

L25: bpf_map_lookup_elem, kernel side return pointer to (percpu) value.

L27-28: value safe percpu pointer, running under RCU read-side.

```
 9 /* eBPF xxx_kern.c side map-lookup */
10 static __always_inline
11 u32 parse_ipv4(struct xdp_md *ctx, u64 l3_offset)
12 {
13   void *data_end = (void *)(long)ctx->data_end;
14   void *data     = (void *)(long)ctx->data;
15   struct iphdr *iph = data + l3_offset;
16   u64 *value;
17   u32 ip_src; /* type need to match map */
18
19   /* Hint: +1 is sizeof(struct iphdr) */
20   if (iph + 1 > data_end) {
21           return XDP_ABORTED;
22   }
23   ip_src = iph->saddr; /* Extract key */
24
25   value = bpf_map_lookup_elem(&blacklist, &ip_src);
26   if (value) {
27           /* Don't need __sync_fetch_and_add(); as percpu map */
28           *value += 1; /* Keep a counter for drop matches */
29           return XDP_DROP;
30   }
31   return XDP_PASS;
32 }
```


Userspace (_user.c) side map-lookup

- Maps lookups from userspace
  - Need file-descriptor (fd) as handle
  - Go through bpf-syscall and memory is copied
- Blacklist exports maps into bpf-filesystem
  - Must mount this: `mount -t bpf bpf /sys/fs/bpf/`
  - _user.c pin/export map via: `bpf_obj_pin()`
    - Unpin via `unlink()` or simply `rm` file
  - _cmdline.c open via: `bpf_obj_get()`
    - File `close()` the fd afterwards

Userspace and percpu maps (read)

- Userspace see all CPUs values
  - Need to sum over these,
  - and size depend on possible CPUs in system

```
 1 static __u64 get_key32_value64_percpu(int fd, __u32 key)
 2 {
 3   /* For PERCPU maps, userspace gets a value per possible CPU */
 4   unsigned int nr_cpus = bpf_num_possible_cpus();
 5   __u64 values[nr_cpus];
 6   __u64 sum = 0;
 7   int i;
 8
 9   if ((bpf_map_lookup_elem(fd, &key, values)) != 0) {
10           fprintf(stderr,
11            "ERR: bpf_map_lookup_elem failed key:0x%X\n", key);
12           return 0;
13   }
14
15   /* Sum values from each CPU */
16   for (i = 0; i < nr_cpus; i++) {
17           sum += values[i];
18   }
19   return sum;
20 }
```


Userspace and percpu maps (update+delete)

- Userspace see all CPUs values
  - Update map elem, need nr_cpus values

```c
static int blacklist_modify(int fd, char *ip_string, unsigned int action)
{
  unsigned int nr_cpus = bpf_num_possible_cpus();
  __u64 values[nr_cpus];
  __u32 key; int res;

  /* Update values for all possible CPUs */
  memset(values, 0, sizeof(__u64) * nr_cpus);

  /* Convert IP-string into 32-bit network byte-order value */
  if (inet_pton(AF_INET, ip_string, &key) <=0)
          return EXIT_FAIL_IP;

  if (action == ACTION_ADD) {
          res = bpf_map_update_elem(fd, &key, values, BPF_NOEXIST);
  } else if (action == ACTION_DEL) {
          res = bpf_map_delete_elem(fd, &key);
  } else {
          return EXIT_FAIL_OPTION;
  }

  if (res != 0) { /* 0 == success */
          if (errno == 17) /* Already in blacklist */
                  return EXIT_OK;
          return EXIT_FAIL_MAP_KEY;
  }
  return EXIT_OK;
}
```


DDoS volume attacks

- Blacklist offer ready-to-use
  - Linux Kernel self protection, at >10G wirespeed!!!
  - Simply modify runtime for further specific filtering
- How to handle volume attacks
  - Exhaust network bandwidth before reaching servers
  - Deploy XDP scrubber machines, closer to edge
    - Modify blacklist to: Use XDP_TX
    - Modify packet headers, e.g. change VLAN header
      - Scrubber part of separate routing VRF
      - Clean packets reinjected into normal VRF
      - Asymmetric routing, XDP_TX is sufficient

The end

- Exciting times for network performance!
  - As of Kernel 4.9
    - For DDoS protection features
    - Linux can compete with DPDK speeds

EXTRA SLIDES

How to modify packet header

- XDP support push/pop headers
  - Done via helper: bpf_xdp_adjust_head()
- Relate to scrubber
  - Redirect packet into another VLAN, to change VRF
  - Customer avoided adjust_head “push” of header
    - Simply add VLAN header on “input” link
    - Thus header already contain a VLAN tag (that is modified)

Status: Linux perf improvements

- Linux performance, recent improvements
  - approx past 2 years:
- Lowest TX layer (single core, pktgen):
  - Started at: 4 Mpps → 14.8 Mpps (← max 10G wirespeed)
- Lowest RX layer (single core):
  - Started at: 6.4 Mpps → 16 Mpps
  - XDP: drop 20Mpps (looks like HW limit)
- IPv4-forwarding: kernel scaling works
  - Single core: 1 Mpps → 2 Mpps → (experiment) 2.5Mpps
  - Multi core : 6 Mpps → 12 Mpps (RHEL7.2 benchmark)
  - XDP single core TX-bounce fwd: 10Mpps

XDP: Types of DDoS

- DDoS filtering types:
  - Best suited for packet based filter decisions (L2 or L3)
  - eBPF could store historic state
    - Arbitrary advanced based on eBPF expressiveness
  - Use another tool for application layer attacks
- Really fast!
  - Realize: Can do wirespeed filtering of small packets
- Fast enough for
  - Filtering DDoS volume attacks on network edge

Blacklist performance difference

- Single CPU benchmark (with single flow)
  - Very fast CPU: i7-6700K CPU @ 4.00GHz
  - NIC 50Gbit/s Mellanox-CX4 (driver: mlx5)
- **(WARNING: this was with mlx5 staging prefetch)**
- Fastest iptables drop in “raw” table:
  - `iptables -t raw -I PREROUTING -p udp --dport 9 -j DROP`
  - Drop: 4,748,646 pps
- XDP blacklist:
  - Drop: 16,939,941 pps