XDP in Practice

DDoS Mitigation @Cloudflare

CLOUDFLARE®

Gilberto Bertin

About me

Systems Engineer at Cloudflare London

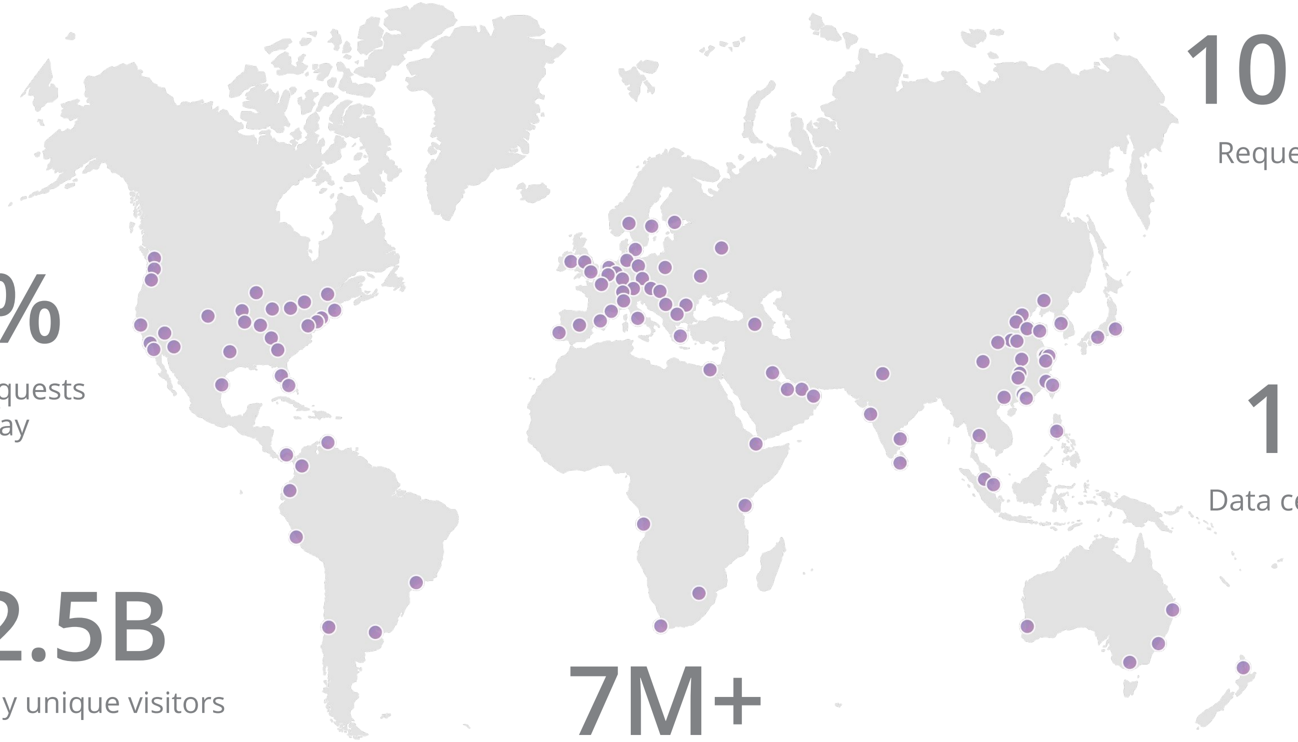
DDoS Mitigation Team

Enjoy messing with networking and Linux kernel

Agenda

- Cloudflare DDoS mitigation pipeline
- Iptables and network packets in the network stack
- Filtering packets in userspace
- XDP and eBPF: DDoS mitigation and Load Balancing

Cloudflare's Network Map



10MM

Requests/second

10%

Internet requests everyday

120+

Data centers globally

2.5B

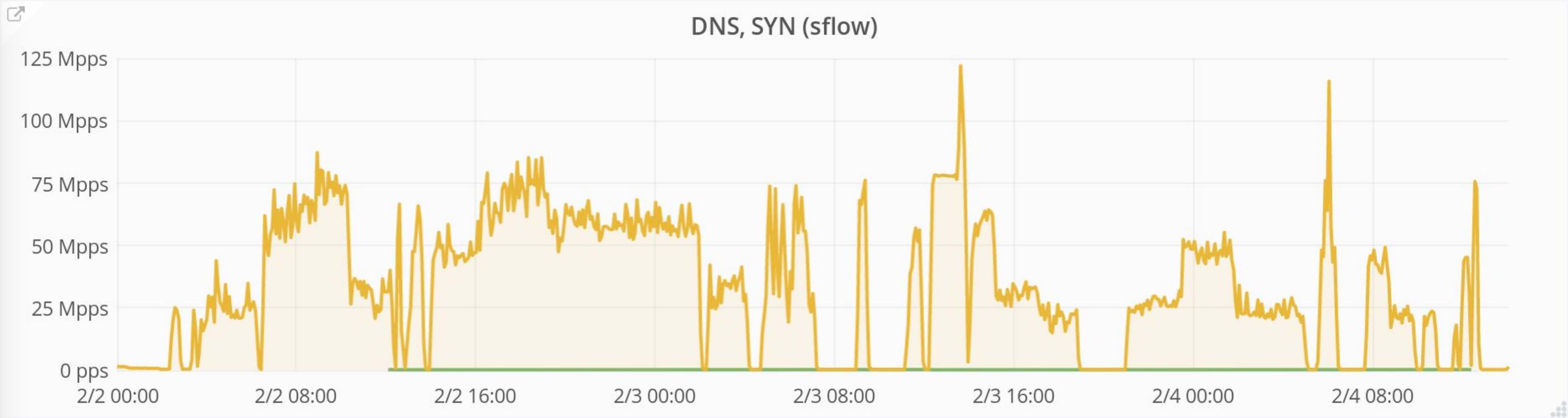
Monthly unique visitors

7M+

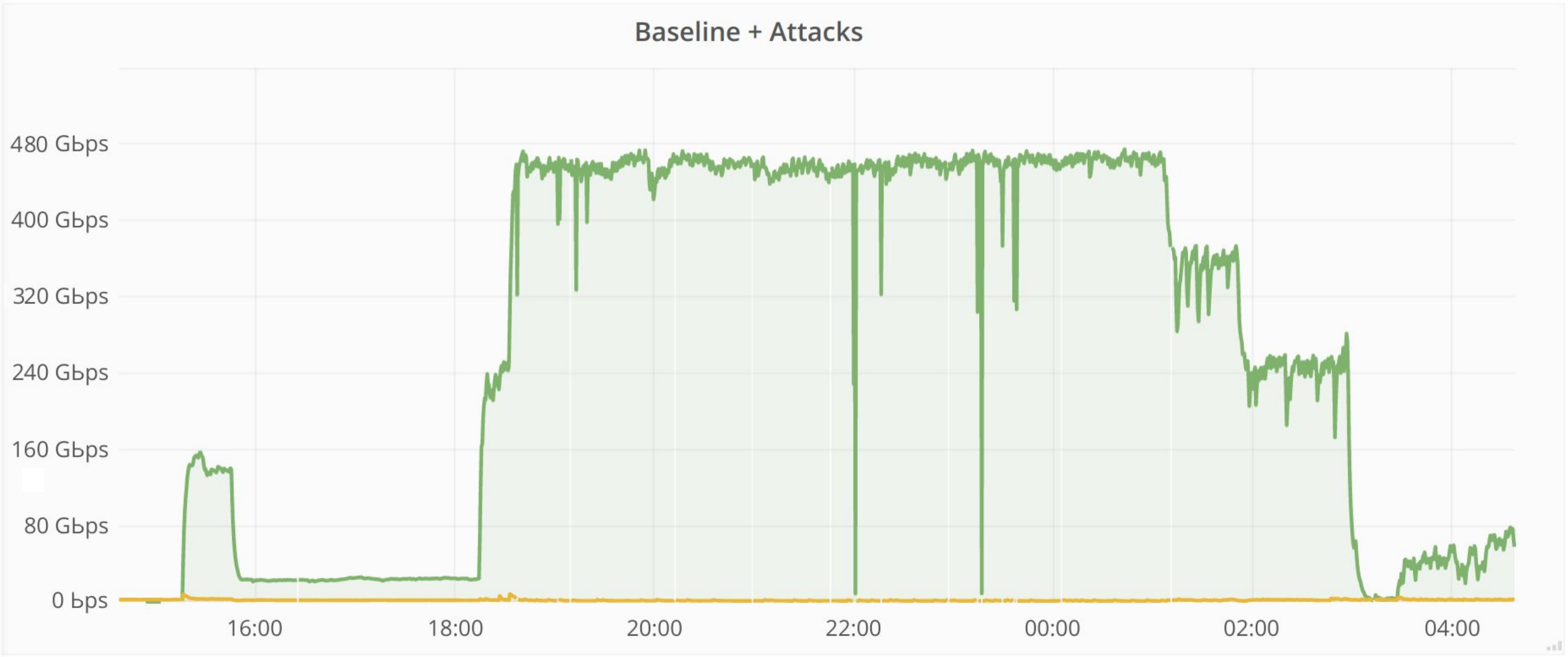
websites, apps & APIs in 150 countries

Everyday we have to mitigate hundreds of different DDoS attacks

- On a normal day: 50-100Mpps/50-250Gbps
- Recorded peaks: 300Mpps/510Gbps



DNS, SYN (sflow)
125 Mpps
100 Mpps
75 Mpps
50 Mpps
25 Mpps
0 pps
2/2 00:00
2/2 08:00
2/2 16:00
2/3 00:00
2/3 08:00
2/3 16:00
2/4 00:00
2/4 08:00



Baseline + Attacks
480 Gbps
400 Gbps
320 Gbps
240 Gbps
160 Gbps
80 Gbps
0 bps
16:00
18:00
20:00
22:00
00:00
02:00
04:00

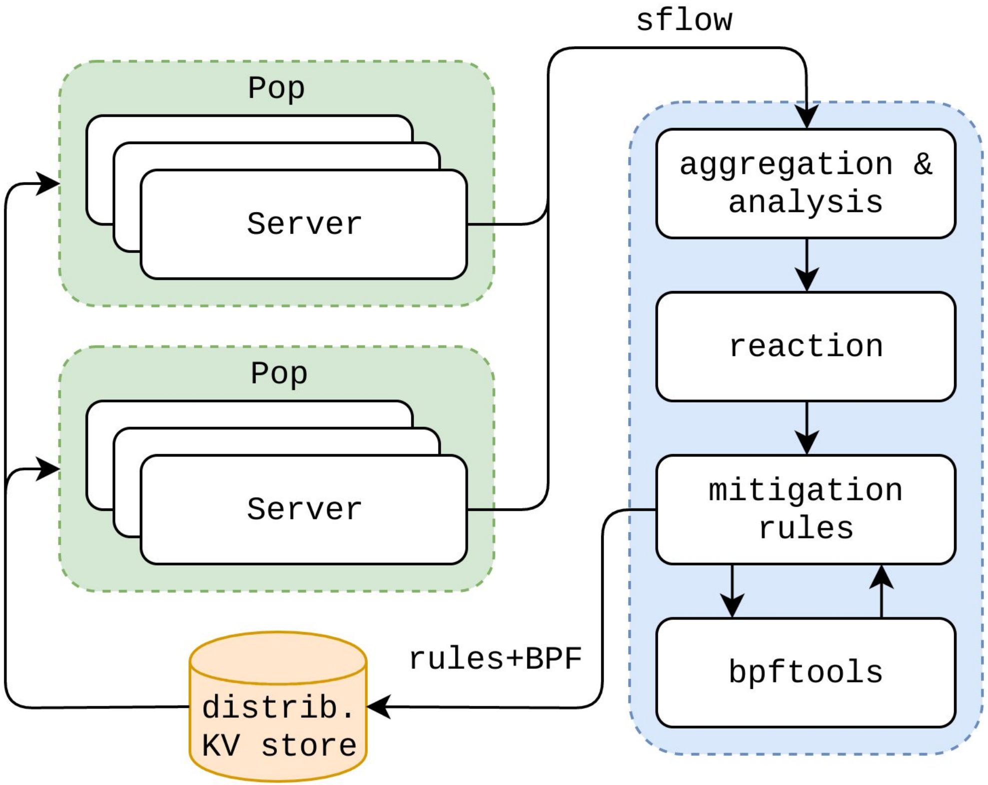
Meet Gatebot

Gatebot

Automatic DDos Mitigation system developed in the last 4 years:

- Constantly analyses traffic flowing through CF network
- Automatically detects and mitigates different kind of DDoS attacks

Gatebot architecture



Traffic Sampling

We don’t need to analyse all the traffic

Traffic is rather sampled:

- Collected on every single edge server
- Encapsulated in SFLOW UDP packets and forwarded to a central location

Traffic analysis and aggregation

Traffic is aggregated into groups e.g.:

- TCP SYNs, TCP ACKs, UDP/DNS
- Destination IP/port
- Known attack vectors and other heuristics

Traffic analysis and aggregation

Mpps	IP	Protocol	Port	Pattern
1	a.b.c.d	UDP	53	*.example.xyz
1	a.b.c.e	UDP	53	*.example.xyz

Reaction

- PPS thresholding: don’t mitigate small attacks
- SLA of client and other factors determine mitigation parameters
- Attack description is turned into BPF

Deploying Mitigations

- Deployed to the edge using a KV database
- Enforced using either Iptables or a custom userspace utility based on Kernel Bypass

Iptables

Iptables is great

- Well known CLI
- Lots of tools and libraries to interface with it
- Concept of tables and chains
- Integrates well with Linux
  - IPSET
  - Stats
- BPF matches support (xt_bpf)

Handling SYN floods with Iptables, BPF and p0f

```
$ ./bpfgen p0f -- '4:64:0:*:mss*10,6:mss,sok,ts,nop,ws:df,id+:0'
56,0 0 0 0,48 0 0 8,37 52 0 64,37 0 51 29,48 0 0 0,84 0 0 15,21 0 48 5,48 0 0
9,21 0 46 6,40 0 0 6,69 44 0 8191,177 0 0 0,72 0 0 14,2 0 0 8,72 0 0 22,36 0 0
10,7 0 0 0,96 0 0 8,29 0 36 0,177 0 0 0,80 0 0 39,21 0 33 6,80 0 0 12,116 0 0
4,21 0 30 10,80 0 0 20,21 0 28 2,80 0 0 24,21 0 26 4,80 0 0 26,21 0 24 8,80 0
0 36,21 0 22 1,80 0 0 37,21 0 20 3,48 0 0 6,69 0 18 64,69 17 0 128,40 0 0 2,2
0 0 1,48 0 0 0,84 0 0 15,36 0 0 4,7 0 0 0,96 0 0 1,28 0 0 0,2 0 0 5,177 0 0
0,80 0 0 12,116 0 0 4,36 0 0 4,7 0 0 0,96 0 0 5,29 1 0 0,6 0 0 65536,6 0 0 0,

$ BPF=(bpfgen p0f -- '4:64:0:*:mss*10,6:mss,sok,ts,nop,ws:df,id+:0')
# iptables -A INPUT -d 1.2.3.4 -p tcp --dport 80 -m bpf --bytecode “${BPF}”
```

bpftools: https://github.com/cloudflare/bpftools

(What is p0f?)

IP version

TCP Window Size and Scale

IP Opts Len

Quirks

```
4:64:0:*:mss*10,6:mss,sok,ts,nop,ws:df,id+:0
```

TTL

MSS

TCP Options

TCP Payload Length

Iptables can’t handle big packet floods.

It can filter 2-3Mpps at most, leaving no CPU to the userspace applications.

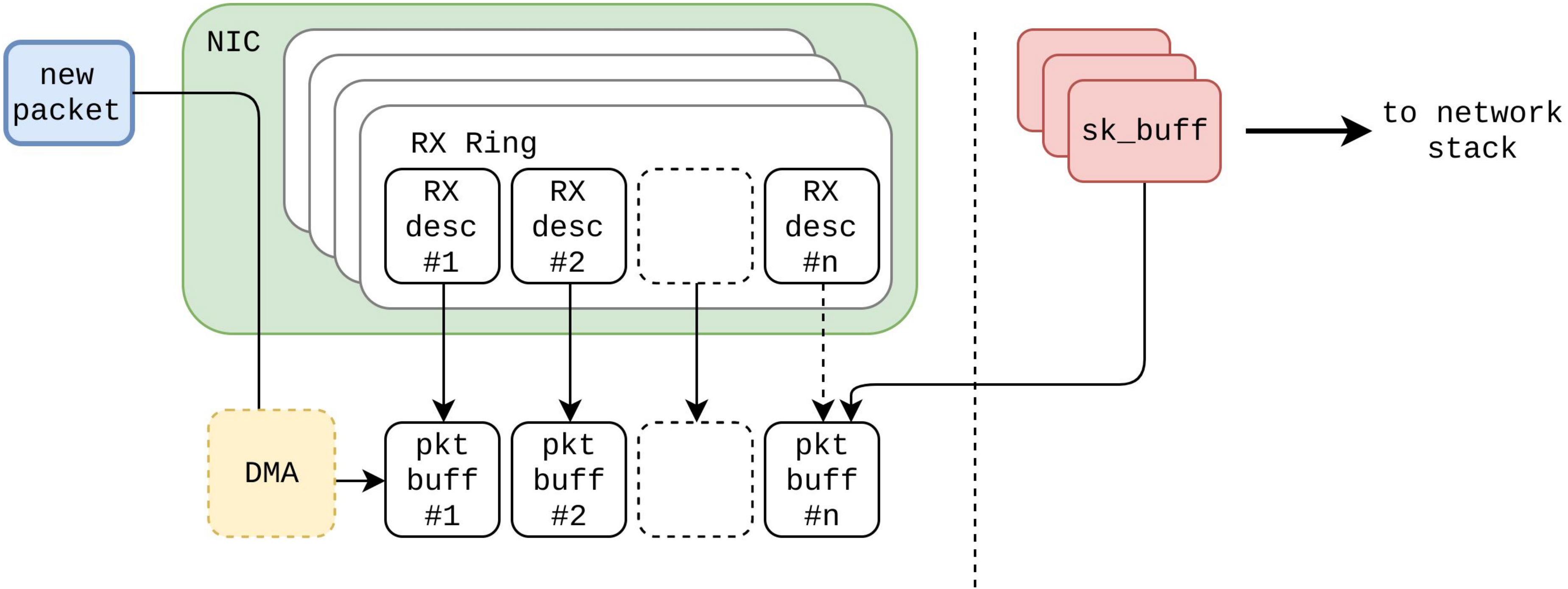
Linux alternatives

- Use raw/PREROUTING
- TC-bpf on ingress
- NFTABLES on ingress

We are not trying to squeeze some more Mpps.
We want to use as little CPU as possible to filter at line rate.

The path of a packet in the Linux Kernel

NIC and kernel packet buffers



Receiving a packet is expensive

- for each RX buffer that has a new packet
  - dma_unmap() the packet buffer
  - build_skb()
  - netdev_alloc_frag() && dma_map() a new packet buffer
  - pass the skb up to the stack
  - free_skb()
  - free old packet page

```
net_rx_action() {
  e1000_clean [e1000]() {
    e1000_clean_rx_irq [e1000]() {
      build_skb() {
        __build_skb() {
          kmem_cache_alloc();
        }
      }
      _raw_spin_lock_irqsave();
      _raw_spin_unlock_irqrestore();
      skb_put();
      eth_type_trans();
      napi_gro_receive() {
        skb_gro_reset_offset();
        dev_gro_receive() {
          inet_gro_receive() {
            tcp4_gro_receive() {
              __skb_gro_checksum_complete() {
                skb_checksum() {
                  __skb_checksum() {
                    csum_partial() {
                      do_csum();
                    }
                  }
                }
              }
```

allocate skbs for the newly received packets

GRO processing

```
          tcp_gro_receive() {
            skb_gro_receive();
          }
        }
      }
    }
    kmem_cache_free() {
      ___cache_free();
    }
  }

  [ .. repeat ..]

  e1000_alloc_rx_buffers [e1000]() {
    netdev_alloc_frag() {
      __alloc_page_frag();
    }
    _raw_spin_lock_irqsave();
    _raw_spin_unlock_irqrestore();

  [ .. repeat ..]
  }
}
}
```

allocate new packet buffers

```
napi_gro_flush() {
  napi_gro_complete() {
    inet_gro_complete() {
      tcp4_gro_complete() {
        tcp_gro_complete();
      }
    }
    netif_receive_skb_internal() {
      __netif_receive_skb() {
        __netif_receive_skb_core() {
          ip_rcv() {
            nf_hook_slow() {
              nf_iterate() {
                ipv4_conntrack_defrag [nf_defrag_ipv4]();
                ipv4_conntrack_in [nf_conntrack_ipv4]() {
                  nf_conntrack_in [nf_conntrack]() {
                    ipv4_get_l4proto [nf_conntrack_ipv4]();
                    __nf_ct_l4proto_find [nf_conntrack]();
                    tcp_error [nf_conntrack]() {
                      nf_ip_checksum();
                    }
                    nf_ct_get_tuple [nf_conntrack]() {
                      ipv4_pkt_to_tuple [nf_conntrack_ipv4]();
                      tcp_pkt_to_tuple [nf_conntrack]();
                    }
                    hash_conntrack_raw [nf_conntrack]();
```

process IP header

Iptables raw/conntrack

```
          __nf_conntrack_find_get [nf_conntrack]();
          tcp_get_timeouts [nf_conntrack]();
          tcp_packet [nf_conntrack]() {
            _raw_spin_lock_bh();
            nf_ct_seq_offset [nf_conntrack]();
            _raw_spin_unlock_bh() {
              __local_bh_enable_ip();
            }
            __nf_ct_refresh_acct [nf_conntrack]();
          }
        }
      }
    }
  }
ip_rcv_finish() {
  tcp_v4_early_demux() {
    __inet_lookup_established() {
      inet_ehashfn();
    }
    ipv4_dst_check();
  }
  ip_local_deliver() {
    nf_hook_slow() {
      nf_iterate() {
        iptable_filter_hook [iptable_filter]() {
          ipt_do_table [ip_tables]() {
```

(more conntrack)

routing decisions

Iptables INPUT chain

```
                tcp_mt [xt_tcpudp]();
                __local_bh_enable_ip();
              }
            }
            ipv4_helper [nf_conntrack_ipv4]();
            ipv4_confirm [nf_conntrack_ipv4]() {
              nf_ct_deliver_cached_events [nf_conntrack]();
            }
          }
        }
        ip_local_deliver_finish() {
          raw_local_deliver();
          tcp_v4_rcv() {
            [ .. ]
          }
        }
      }
    }
  }
}
}
}
}
}
__kfree_skb_flush();
}
```

← l4 protocol handler

Iptables is not slow.

It's just executed **too late** in the stack.

Userspace

Packet Filtering

Kernel Bypass 101

- One or more RX rings are
  - detached from the Linux network stack
  - mapped in and managed by userspace
- Network stack ignores packets in these rings
- Userspace is notified when there's a new packet in a ring

Kernel Bypass is great for high volume packet filtering

- No packet buffer or sk_buff allocation
  - Static preallocated circular packet buffers
  - It's up to the userspace program to copy data that has to be persistent
- No kernel processing overhead

Offload packet filtering to userspace

- Selectively steer traffic with flow-steering rule to a specific RX ring
    - e.g. all TCP packets with dst IP x and dst port y should go to RX ring #n
- Put RX ring #n in kernel bypass mode
- Inspect raw packets in userspace and
    - Reinject the legit ones
    - Drop the malicious one: no action required

Offload packet filtering to userspace

```
while(1) {
  // poll RX ring, wait for a packet to arrive
  u_char *pkt = get_packet();

  if (run_bpf(pkt, rules) == DROP)
    // do nothing and go to next packet
    continue;

  reinject_packet(pkt)
}
```

Netmap, EF_VI
PF_RING, DPDK

..

An order of magnitude faster than Iptables.
6-8 Mpps on a **single core**

Kernel Bypass for packet filtering - disadvantages

- Legit traffic has to be reinjected (can be expensive)
- One or more cores have to be reserved
- Kernel space/user space context switches

XDP

Express Data Path

XDP

- New alternative to Iptables or Userspace offload included in the Linux kernel
- Filter packets as soon as they are received
- Using an eBPF program
- Which returns an action (XDP_PASS, XDP_DROP,)
- It's even possible to modify the content of a packet, push additional headers and retransmit it

Should I trash my Iptables setup?

No, XDP is not a replacement for regular Iptables firewall*

* yet https://www.spinics.net/lists/netdev/msg483958.html

```
net_rx_action() {
  e1000_clean [e1000]() {
    e1000_clean_rx_irq [e1000]() {
      build_skb() {
        __build_skb() {
          kmem_cache_alloc();
        }
      }
      _raw_spin_lock_irqsave();
      _raw_spin_unlock_irqrestore();
      skb_put();
      eth_type_trans();
      napi_gro_receive() {
        skb_gro_reset_offset();
        dev_gro_receive() {
          inet_gro_receive() {
            tcp4_gro_receive() {
              __skb_gro_checksum_complete() {
                skb_checksum() {
                  __skb_checksum() {
                    csum_partial() {
                      do_csum();
                    }
                  }
                }
              }
```

BPF_PRG_RUN()

Just before allocating skbs

e1000 RX path with XDP

```
act = e1000_call_bpf(prog, page_address(p), length);

switch (act) {

/* .. */

case XDP_DROP:
default:
    /* re-use mapped page. keep buffer_info->dma
     * as-is, so that e1000_alloc_jumbo_rx_buffers
     * only needs to put it back into rx ring
     */
    total_rx_bytes += length;
    total_rx_packets++;
    goto next_desc;
}
```

XDP vs Userspace offload

- Same advantages as userspace offload:
  - No kernel processing overhead
  - No packet buffers or sk_buff allocation/deallocation cost
  - No DMA map/unmap cost
- But well integrated with the Linux kernel:
  - eBPF to express the filtering logic
  - No need to inject packets back into the network stack

eBPF

extended Berkeley
Packet Filter

eBPF

- Programmable in-kernel VM
  - Extension of classical BPF
  - Close to a real CPU
    - JIT on many arch (x86_64, ARM64, PPC64)
  - Safe memory access guarantees
  - Time bounded execution (no backward jumps)
  - Shared maps with userspace
- LLVM eBPF backend:
  - .c -> .o

XDP_DROP example

```
SEC("xdp1")
int xdp_prog1(struct xdp_md *ctx)
{
	void *data     = (void *)(long)ctx->data;
	void *data_end = (void *)(long)ctx->data_end;

	struct ethhdr *eth = (struct ethhdr *)data;
	if (eth + 1 > (struct ethhdr *)data_end)
		return XDP_ABORTED;
	if (eth->h_proto != htons(ETH_P_IP))
		return XDP_PASS;

	struct iphdr *iph = (struct iphdr *)(eth + 1);
	if (iph + 1 > (struct iphdr *)data_end)
		return XDP_ABORTED;
	// if (iph->..
	//	return XDP_PASS;

	return XDP_DROP;
}
```

access packet buffer begin and end

access ethernet header

make sure we are not reading past the buffer

XDP_DROP example

```
SEC("xdp1")
int xdp_prog1(struct xdp_md *ctx)
{
	void *data     = (void *)(long)ctx->data;
	void *data_end = (void *)(long)ctx->data_end;

	struct ethhdr *eth = (struct ethhdr *)data;
	if (eth + 1 > (struct ethhdr *)data_end)
		return XDP_ABORTED;
	if (eth->h_proto != htons(ETH_P_IP))
		return XDP_PASS;

	struct iphdr *iph = (struct iphdr *)(eth + 1);
	if (iph + 1 > (struct iphdr *)data_end)
		return XDP_ABORTED;
	// if (iph->..
	//    return XDP_PASS;

	return XDP_DROP;
}
```

check this is an IP packet

access IP header

make sure we are not reading past the buffer

XDP and maps

```
struct bpf_map_def SEC("maps") rxcnt = {
      .type = BPF_MAP_TYPE_PERCPU_ARRAY,
      .key_size = sizeof(unsigned int),
      .value_size = sizeof(long),
      .max_entries = 256,
};

SEC("xdp1")
int xdp_prog1(struct xdp_md *ctx)
{
      unsigned int key = 1;

// ..

      long *value = bpf_map_lookup_elem(&rxcnt, &key);
      if (value)
            *value += 1;

}
```

define a new map

get a ptr to the value indexed by "key"

update the value

Why not automatically generate XDP programs!

```
$ ./p0f2ebpf.py --ip 1.2.3.4 --port 1234 '4:64:0:*:mss*10,6:mss,sok,ts,nop,ws:df,id+:0'

static inline int match_p0f(struct xdp_md *ctx)
{
      void *data     = (void *)(long)ctx->data;
      void *data_end = (void *)(long)ctx->data_end;

      struct ethhdr *eth_hdr;
      struct iphdr  *ip_hdr;
      struct tcphdr *tcp_hdr;
      unsigned char *tcp_opts;

      eth_hdr = (struct ethhdr *)data;
      if (eth_hdr + 1 > (struct ethhdr *)data_end)
            return XDP_ABORTED;
      if_not (eth_hdr->h_proto == htons(ETH_P_IP))
            return XDP_PASS;
```

```
ip_hdr = (struct iphdr *)(eth_hdr + 1);
if (ip_hdr + 1 > (struct iphdr *)data_end)
      return XDP_ABORTED;
if_not (ip_hdr->version == 4)
      return XDP_PASS;
if_not (ip_hdr->daddr == htonl(0x1020304))
      return XDP_PASS;
if_not (ip_hdr->ttl <= 64)
      return XDP_PASS;
if_not (ip_hdr->ttl > 29)
      return XDP_PASS;
if_not (ip_hdr->ihl == 5)
      return XDP_PASS;
if_not ((ip_hdr->frag_off & IP_DF) != 0)
      return XDP_PASS;
if_not ((ip_hdr->frag_off & IP_MBZ) == 0)
      return XDP_PASS;

tcp_hdr = (struct tcphdr*)((unsigned char *)ip_hdr + ip_hdr->ihl * 4);
if (tcp_hdr + 1 > (struct tcphdr *)data_end)
      return XDP_ABORTED;
if_not (tcp_hdr->dest == htons(1234))
      return XDP_PASS;
if_not (tcp_hdr->doff == 10)
      return XDP_PASS;
if_not ((htons(ip_hdr->tot_len) - (ip_hdr->ihl * 4) - (tcp_hdr->doff * 4)) == 0)
      return XDP_PASS;
```

```
    tcp_opts = (unsigned char *)(tcp_hdr + 1);
    if (tcp_opts + (tcp_hdr->doff - 5) * 4 > (unsigned char *)data_end)
          return XDP_ABORTED;
    if_not (tcp_hdr->window == *(unsigned short *)(tcp_opts + 2) * 0xa)
          return XDP_PASS;
    if_not (*(unsigned char *)(tcp_opts + 19) == 6)
          return XDP_PASS;
    if_not (tcp_opts[0] == 2)
          return XDP_PASS;
    if_not (tcp_opts[4] == 4)
          return XDP_PASS;
    if_not (tcp_opts[6] == 8)
          return XDP_PASS;
    if_not (tcp_opts[16] == 1)
          return XDP_PASS;
    if_not (tcp_opts[17] == 3)
          return XDP_PASS;

    return XDP_DROP;
}
```

Migrating to XDP

Deploying Mitigations

Keep most of the infrastructure (detection/reaction):

- Migrate mitigation tools from cBPF to eBPF
  - Generate an eBPF program out of all the rule descriptions
- Use eBPF maps for metrics
- bpf_perf_event_output to sample dropped packets
- Get rid of kernel-bypass

Deploying Mitigations

Keep most of the infrastructure (detection/reaction):

- Migrate mitigation tools from cBPF to eBPF
    - Generate an eBPF program out of all the rule descriptions
- Use eBPF maps for metrics
- bpf_perf_event_output to sample dropped packets
- Get rid of kernel-bypass

```
$ ./ctoebpf '35,0 0 0 0,48 0 0 8,37 31 0 64,37 0 30 29,48 0 0 0,84 0 0 15,21 0 27 5,48 0 0 9,21 0 25 6,40 0 0
6,69 23 0 8191,177 0 0 0,80 0 0 12,116 0 0 4,21 0 19 5,48 0 0 6,69 17 0 128,40 0 0 2,2 0 0 14,48 0 0 0,84 0 0
15,36 0 0 4,7 0 0 0,96 0 0 14,28 0 0 0,2 0 0 2,177 0 0 0,80 0 0 12,116 0 0 4,36 0 0 4,7 0 0 0,96 0 0 2,29 0 1
0,6 0 0 65536,6 0 0 0,'

int func(struct xdp_md *ctx)
{
    uint32_t a, x, m[16];
    uint8_t *sock = ctx->data;
    uint8_t *sock_end = ctx->data_end;
    uint32_t sock_len = sock_end - sock;
    uint32_t l3_off = 14;

    sock     += l3_off;
    sock_len -= l3_off;

    a = 0x0;
    if (sock + l3_off + 0x8 + 0x1 > sock_end)
       return XDP_ABORTED;
    a = *(sock + 0x8);
    if (a > 0x40)
       goto ins_34;
    if (!(a > 0x1d))
       goto ins_34;
    if (sock + l3_off + 0x0 + 0x1 > sock_end)
       return XDP_ABORTED;
```

```
 // ..
    a = htons(*(uint16_t *) (sock + 0x2));
    m[0xe] = a;
    if (sock + l3_off + 0x0 + 0x1 > sock_end)
       return XDP_ABORTED;
    a = *(sock + 0x0);
    a &= 0xf;
    a *= 0x4;
    x = a;
    a = m[0xe];
    a -= x;
    m[0x2] = a;
    if (sock + 0x0 > sock_end)
       return XDP_ABORTED;
    x = 4 * (*(sock + 0x0) & 0xf);
    if (sock + x + 0xc + 0x1 > sock_end)
       return XDP_ABORTED;
    a = *(sock + x + 0xc);
    a >>= 0x4;
    a *= 0x4;
    x = a;
    a = m[0x2];
    if (!(a == x))
       goto ins_34;
    return XDP_DROP;
ins_34:
    return XDP_PASS;
}
```

Load Balancing with XDP

XDP_TX

- XDP allows to modify and retransmit a packet: XDP_TX target
  - Rewrite DST MAC address or
  - IP in IP encapsulation with bpf_xdp_adjust_head()
- eBPF maps to keep established connections state
- Add packet filtering XDP program in front
  - Chain multiple XDP programs with BPF_MAP_TYPE_PROG_ARRAY and bpf_tail_call

```
int xdp_l4tx(struct xdp_md *ctx)
{
    void *data     = (void *)(long)ctx->data;
    void *data_end = (void *)(long)ctx->data_end;

    struct ethhdr *eth;
    struct iphdr  *iph;
    struct tcphdr *tcph;
    unsigned char *next_hop;

    eth = (struct ethhdr *)data;
    if (eth + 1 > (struct ethhdr *)data_end)
      return XDP_ABORTED;

    /* - access IP and TCP header
     * - return XDP_PASS if not TCP packet
     * - track_tcp_flow if it’s a new one
     */

    next_hop = get_next_hop(iph, tcph);

    memcpy(eth->h_dest, next_hop, 6);
    memcpy(eth->h_source, IFACE_MAC_ADDRESS, 6);

    return XDP_TX;
}
```

How to try it

- Generic XDP from Linux 4.12
- Take a look at /samples/bpf in Linux kernel sources:
  - Actual XDP programs: (xdp1_kern.c, xdp1_user.c)
  - Helpers: bpf_helpers.h, bpf_load.{c,h}, Libbpf.h
- Take a look at bcc and its examples

```
from bcc import BPF

device = "eth0"
flags = 2 # XDP_FLAGS_SKB_MODE

b = BPF(text = """
// Actual XDP C Source
""", cflags=["-w"])

fn = b.load_func("xdp_prog1", BPF.XDP)
b.attach_xdp(device, fn, flags)

counters = b.get_table("counters")

b.remove_xdp(device, flags)
```

Conclusions

XDP is a great tool for 2 reasons

- Speed: back to drop or modify/retransmit packets in kernel space at the lowest layer of the network stack
- Safety: eBPF allows to run C code in kernel space with program termination and memory safety guarantees (i.e. your eBPF program is not going to cause a kernel panic)

Thank You!

Questions?

gilberto@cloudflare.com
@akajibi