CLOUDFLARE

XDP in practice

Integrating XDP into our DDoS mitigation pipeline

Gilberto Bertin - gilberto@cloudflare.com

Cloudflare Network

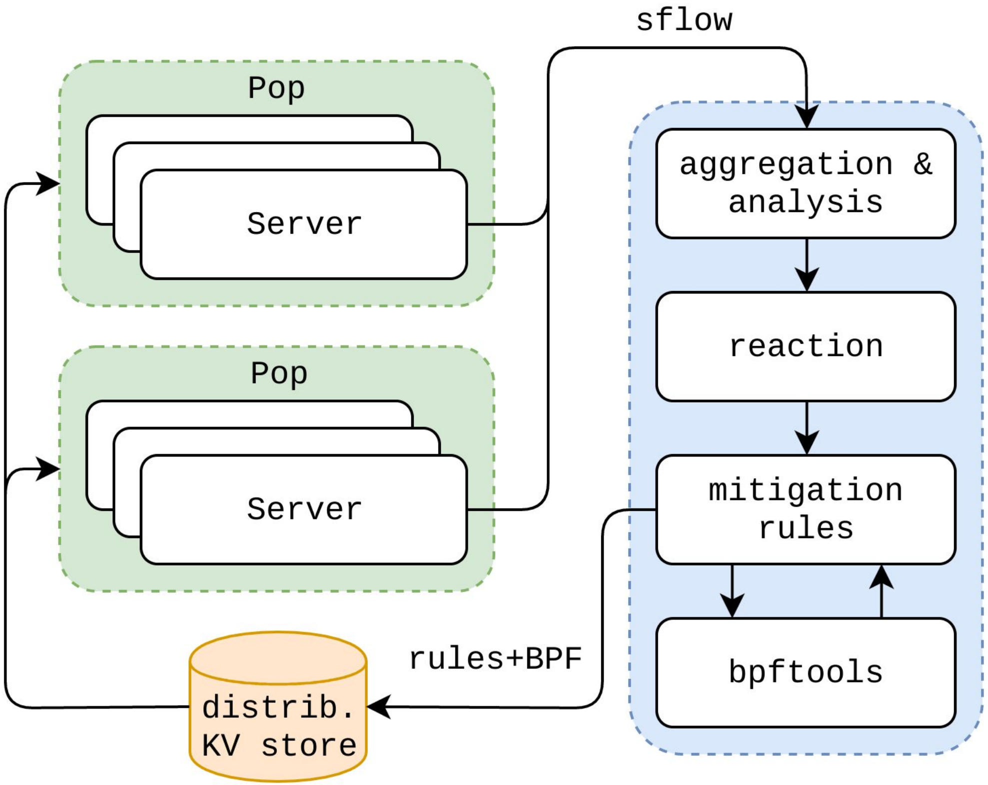
CLOUDFLARE

Gatebot

- Automatic DDoS mitigation system developed in the last 3 years which:
  - Constantly analyses traffic flowing through Cloudflare network
  - Automatically detects and mitigates different kinds of DDoS attacks

CLOUDFLARE

Gatebot architecture



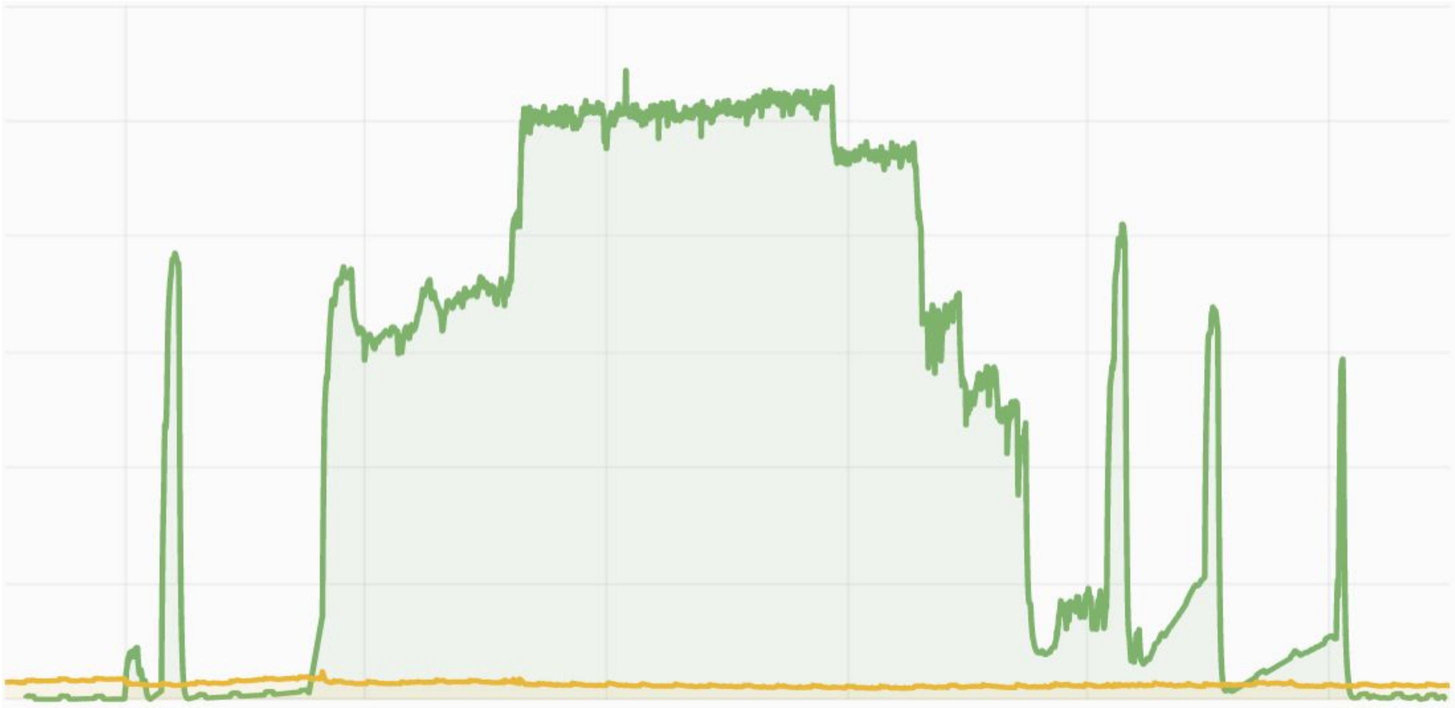
CLOUDFLARE

Traffic sampling

- To detect DDoS attacks only a small portion of traffic is needed
- Traffic samples are:
    - Collected on every edge server
    - Encapsulated in SFLOW UDP packets and forwarded to a central location

CLOUDFLARE

What is an attack



Traffic analysis and aggregation

Traffic is aggregated into flows:

- e.g. TCP SYNs, TCP ACKs, UDP/DNS
- Destination net and port
- Known attack vectors and other heuristics

Traffic analysis and aggregation

Mpps	IP	Protocol	Port	Pattern
1	x.x.x.x	UDP	53	*.example.xyz
1	y.y.y.y	UDP	53	*.example.xyz

CLOUDFLARE

Reaction

- Pps thresholding: small attacks do not need to be mitigated
- SLA of the client and other factors are taken into account to determine the mitigation parameters
- The attack description is turned into BFP

Bpftools

Set of utilities to generate BPF bytecode that can match specific traffic.

```
$ ./bpfgen dns *.example.xyz
```

```
18,177 0 0 0,0 0 0 20,12 0 0 0,7 0 0 0,80 0 0 0,12 0 0 0,4
0 0 1,7 0 0 0,64 0 0 0,21 0 7 124090465,64 0 0 4,21 0 5
1836084325,64 0 0 8,21 0 3 58227066,80 0 0 12,21 0 1 0,6 0
0 65535,6 0 0 0,
```

Pushing mitigations back to the edge

Mitigations are then:

- Deployed to the edge servers using a key value database
    - On every server a daemon listen for updates
- Applied using either Iptables or a userspace program to filter traffic with BPF

Mitigations: Iptables

- Initially it was the only tool to filter traffic
- With the `xt_bpf` module it was possible to specify complex filtering rules
- But we soon started experiencing IRQ storms during big attacks
    - All CPUs were busy dropping packets, userspace applications were starving of CPU

Mitigations: userspace offload

So we moved to userspace offload:

- Based on SolarFlare EF_VI
- Network traffic is offloaded to userspace before it hits the Linux network stack
- Allows to run BPF in userspace
- An order of magnitude faster than Iptables

Mitigations: userspace offload

Still not the optimal solution:

- Requires one or more CPUs to busy poll the NIC event queue
- Reinjecting packets in the network stack is expensive

Migrating to XDP

Single unified solution to filter traffic:

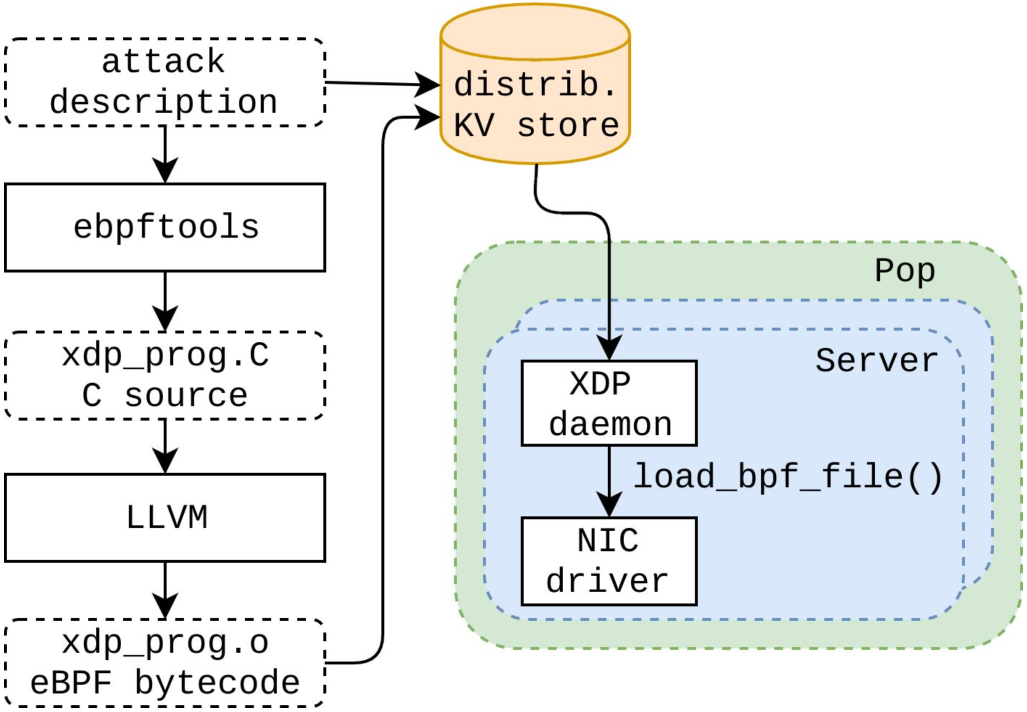
- Move away from Iptables the filtering logic
- No more need to filter network traffic in userspace

ebpftools

Automatically generate an XDP program from the attack descriptions which will be:

- Compiled to eBPF
- Distributed to edge servers using the same KV store we are using today

ebpftools


attack description
ebpftools
xdp_prog.C
C source
LLVM
xdp_prog.o
eBPF bytecode
distrib.
KV store
Pop
Server
XDP
daemon
load_bpf_file()
NIC
driver

Migrating to XDP

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

    ret = rule_1(data, data_end);
    if (ret != XDP_PASS)
        return ret;

    ret = rule_2(data, data_end);
    if (ret != XDP_PASS)
        return ret;

    //..
    return XDP_PASS;
}
```

Multiple if statements, one for each rule

CLOUDFLARE

Migrating to XDP

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

    ret = rule_1(data, data_end);
    if (ret != XDP_PASS)
        return ret;

    ret = rule_2(data, data_end);
    if (ret != XDP_PASS)
        return ret;

    //..
    return XDP_PASS;
}
```

If none is a match, the packet is accepted

CLOUDFLARE

Migrating to XDP

```
static inline int rule_1(void *data, void *data_end) {
    if (! condition_1)
        return XDP_PASS;

    if (! condition_2)
        return XDP_PASS;

    // ..

    update_rule_counters(1);
    sample_packet(data, data_end);

    return XDP_DROP;
}
```

A rule is made up of multiple simple if conditions

Migrating to XDP

```
static inline int rule_1(void *data, void *data_end) {
    if (! condition_1)
        return XDP_PASS;

    if (! condition_2)
        return XDP_PASS;

    // ..

    update_rule_counters(1);
    sample_packet(data, data_end);

    return XDP_DROP;
}
```

Before dropping a packet 2 more actions are needed

CLOUDFLARE

Migrating to XDP

```
void sample_packet(void *data, void *data_end) {
    // mark the packet to be sampled
}

static inline void update_rule_counters(int rule_id) {
    long *value =
        bpf_map_lookup_elem(&c_map, &rule_id);

    if (value)
        *value += 1;
}
```

eBPF map shared with userspace

CLOUDFLARE

Example: p0f

- Tool to passively analyse and categorise network traffic
- Extremely concise syntax to serialise TCP SYN packets:

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

CLOUDFLARE

p0f and bpftools

Bpftools has already support for p0f signatures:

```
$ ./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 0 1 0,6 0
0 65536,6 0 0 0,
```

p0f and ebpftools

Moving to eBPF brings many benefits:

- Emit C code
  - can be optimised by Clang
- No longer a 64 instruction limitation
- Easy to combine multiple p0f signatures
  - C functions: xdp_md * → XDP_ACTION

p0f

```
$ ./p0f2ebpf -- p0f 4:64:0:*:mss*10,6:mss,sok,ts,nop,ws:df,id+:0

static inline int match_p0f(void *data, void *data_end) {
    struct ethhdr *eth_hdr;
    struct iphdr  *ip_hdr;
    struct tcphdr *tcp_hdr;
    u8            *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;
```

Packet boundary checks before accessing it

CLOUDFLARE

p0f

```
ip_hdr = (struct iphdr *)(eth_hdr + 1);
if (ip_hdr + 1 > (struct iphdr *)data_end)
    return XDP_ABORTED;
if_not (ip_hdr->daddr == htonl(0x1020304))
    return XDP_PASS;
if_not (ip_hdr->version == 4)
    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)
```

Boundary checks for IP header

p0f

```
ip_hdr = (struct iphdr *)(eth_hdr + 1);
if (ip_hdr + 1 > (struct iphdr *)data_end)
    return XDP_ABORTED;
if_not (ip_hdr->daddr == htonl(0x1020304))
    return XDP_PASS;
if_not (ip_hdr->version == 4)
    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)
```

Conditions the IP header must meet

CLOUDFLARE

p0f

```
tcp_hdr = (struct tcphdr*)((u8 *)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;
```

p0f

```
tcp_opts = (u8 *)(tcp_hdr + 1);
if (tcp_opts + (tcp_hdr->doff - 5) * 4 > (u8 *)data_end)
    return XDP_ABORTED;
if_not (htons(tcp_hdr->window) == htons(*(u16 *)(tcp_opts + 2)) * 0xa)
    return XDP_PASS;
if_not (*(u8 *)(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;
```

CLOUDFLARE

XDP Issues

Recent kernel and drivers with XDP support are required:

- At least Linux 4.8
- As of Linux 4.10 only Mellanox and QLogic support XDP

CLOUDFLARE

XDP Issues

Sampling packets that are going to be discarded is suboptimal:

- No XDP action or access to sk_buff to mark the packet
- Still possible to mangle the packet buffer by adding a header or changing a field

Conclusions

We think XDP is great for 2 reasons:

- Speed: back to dropping packets in kernel space at the lowest layer
- Safety: eBPF allows to run C code in kernel space with program termination and memory safety guarantees

Questions?

Gilberto Bertin
gilberto@cloudflare.com

CLOUDFLARE