XDP in practice: integrating XDP into our DDoS mitigation pipeline

Gilberto Bertin
Cloudflare Ltd.
London, UK
gilberto@cloudflare.com

Abstract

To absorb large DDoS (distributed denial of service) attacks, the Cloudflare DDoS mitigation team has developed a solution based on kernel bypass and classic BPF. This allows us to filter network packets in userspace, skipping the usual packet processing done by Netfilter and the Linux network stack. This approach has solved performance issues that were experienced whilst handling large packet floods using solely the vanilla Linux kernel features.

In this paper we will first introduce our current architecture and then discuss a proposed solution based on XDP and eBPF. We will explain how XDP can be used in our infrastructure and which parts of our system need to be rewritten and adapted to make use of it. We will then conclude with the issues we have experienced so far with XDP.

Keywords

XDP, eBPF, DDoS

Introduction

With more than a hundred points of presence and millions of customer websites, Cloudflare has visibility into a significant portion of internet traffic. As a reverse proxy our servers sit in front of customer servers and filter out traffic that is deemed to be malicious. Every day we have to mitigate hundreds of DDoS attacks targeting many different web properties.

In the last 3 years we have developed a fully automated DDoS mitigation system that we have named GateBot. This system is constantly monitoring the traffic flowing through our network and is able to detect different kind of DDoS attacks. Once an attack is detected, GateBot automatically deploys mitigation rules to filter it.

Current architecture

The current pipeline architecture can be broken down into the following phases:

- Traffic sampling
- Traffic aggregation and analysis
- Attack reaction
- Attack mitigation

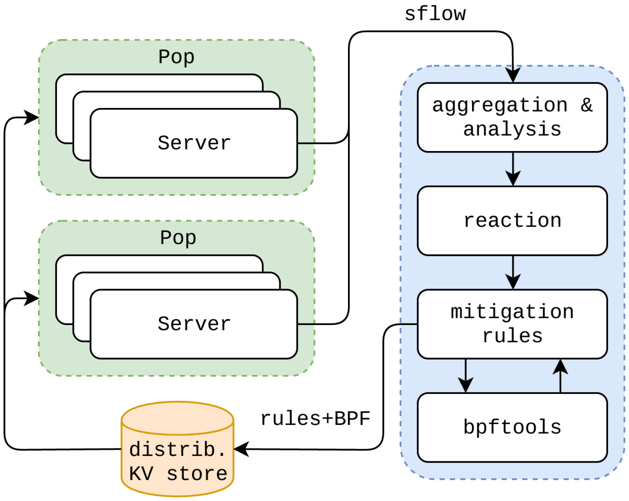

Figure 1: GateBot architecture.

It is worth noting that GateBot architecture is not based on specific scrubbing centers where attack traffic is forwarded to be filtered. Traffic is rather filtered directly on the edge. The system works as follow:

- our servers send sampled traffic to a central location
- the central system detects attacks and deploys mitigation rules back to the edge servers
- the edge servers apply the mitigations to filter out malicious traffic

Traffic sampling

Analysing the entire traffic flowing through our network would be impractical and a waste of resources. In fact, when dealing with DDoS attacks of millions of packets per second, only a fraction of the traffic is more than sufficient to detect malicious floods. This is why our pipeline works with packet samples.

Samples are collected on every machine using the Iptables `NFLOG` target, and are then forwarded to a userspace daemon, which encapsulates them in Sflow [1] UDP packets that are sent to a datacenter where they are aggregated and analysed.

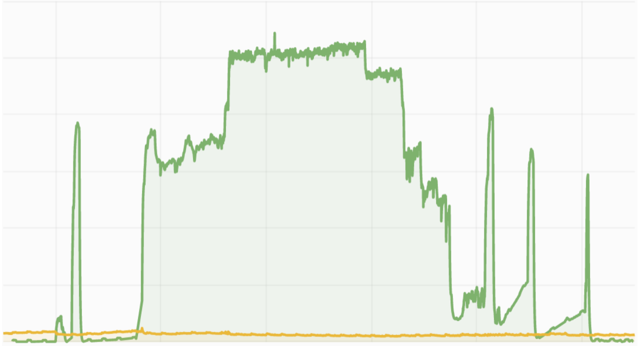

Figure 2: Example of multiple attacks over 24 hours. The green line represents the attack traffic in packets per second, while the yellow one represents the legitimate traffic.

Traffic aggregation and analysis

To better understand how attacks are detected it is necessary to first define what an attack is. An attack can be generically defined as a big spike of traffic targeting a specific IP/subnet and service. Attacks are usually measured in packets per second, but for certain types of attacks bytes per second is also a meaningful metric.

After samples have been collected in a central location, they are then aggregated into macro categories (e.g. TCP SYN samples are separated from UDP/DNS ones) and every group is analysed separately. For every group the traffic is then partitioned into different flows which share common characteristics. At first traffic is aggregated by the destination network and port pair, which allows us to detect attacks targeting a specific IP (or subnet) and service. On top of that we apply other aggregations based on known attack vectors and other heuristics. The result of this phase is a description of all the different floods the network is receiving.

Reacting

Once traffic is aggregated we apply a thresholding mechanism to distinguish attacks that must be mitigated, from attack traffic that is just too low to cause any harm. Then other factors like the topology of the attack and the SLA of the attacked customer are taken into account to determine parameters for the mitigation action. The output of this phase is an abstract description of the mitigations that should be applied. For example GateBot may have detected a 1 million packets per second attack against a specific DNS name server, with a random prefix DNS queries in payload.

Then every rule's abstract description is turned into a BPF bytecode that can be run on the edge to filter the attack traffic. For this task we developed a set of utilities called bpftools [2]. There are different utilities prepared for different kinds of traffic; every tool works by accepting parameters which describe the type of traffic that has to be matched, producing as output the appropriate BPF bytecode.

Choosing BPF was initially motivated by the fact that when we were relying solely on Iptables to filter traffic, Iptables was not expressive enough to describe certain kind of attacks. Thanks to the `xt_bpf` module, BPF appeared to be the best option to express arbitrarily complex filtering patterns (`u32` was another potential alternative, but it was not as flexible as BPF). Later on, when we started filtering traffic with a userspace tool, BPF proved again to be the right choice for us, since it was easy to run the same bytecode we were using with Iptables with a userspace BPF interpreter.

Pushing mitigations to the server

After the reaction phase, mitigation rules have to be applied to the servers which are targeted by the attacks. For this task we use a distributed key-value database which allows us to push rules rapidly to our fleet of machines. On every server a daemon listens for updates on the KV store, and when new rules are published, the daemon updates the mitigations on the machine accordingly.

In practice mitigations are applied using two different systems: Iptables and a program which allows us to run BPF in userspace on the traffic (bypassing the network stack and Iptables).

Iptables Initially Iptables was the only tool used to mitigate DDoS attacks. With the help of `bpftools` and the `xt_bpf` module it was possible to express complex filtering rules: basic packet patterns were expressed as a regular Iptables rule, while the remaining matching logic was expressed using BPF. However we soon started experiencing performance limitations: mitigating large attacks only with Iptables was causing our servers to suffer from IRQ storms where all the CPUs were busy processing just network packets and the applications were starved of CPU. This situation forced us to evaluate a different approach based on kernel bypass.

Over the time we noticed some improvements in the ability of Iptables to handle more and more traffic (especially with Linux 4.4 and the introduction of TCP lockless listeners[3]) but we are still observing better performance with our solution based on kernel bypass.

Kernel bypass and userspace filtering To overcome Iptables performance limitations, a userspace utility to filter network traffic has been developed. This solution is based on the SolarFlare `EF_VI` API, which allows to map one or more network card rings in userspace, bypassing completely the Linux network stack. Packets are then filtered using BPF and the legitimate traffic is reinjected back in the network stack.

The performance of this solution is an order of magnitude greater than Iptables in terms of packets per second filtered, but it still presents some issues: first the `EF_VI` API works by installing hardware flow steer rules, based on the destination IP and port, that redirect specific traffic to one or more receive queues that are then mapped in userspace. This means that in some circumstances we may end up offloading a considerable amount of legitimate traffic to userspace, which then has to be reinjected to the network stack (with potential performance implications). Moreover, to minimize the latency the userspace program has to constantly poll the event queue, which means that one or more CPUs (depending on the type of the attacks) must be completely reserved for this task.

Migrating to XDP

In this section we will present how we are planning to migrate from our current solution based on classical BPF and Iptables/userspace offload to eBPF and XDP.

ebpftools

Currently every filtering rule is expressed as a list of Iptables parameters (like the IP or network, TCP flags, ...) and optionally some BPF bytecode to match specific patterns that is not possible to express by just using Iptables.

With eBPF and XDP it will be possible to express the entire filtering logic for all the rules in a single XDP program. The following is a simplified example of how a XDP program that mitigates multiple attacks may look like:

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

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;
}

static inline int rule_1(void *data, void *data_end) {
    // if any of the rule conditions is not met
    //     return XDP_PASS;

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

    return XDP_DROP;
}

// static inline int rule_2(..)

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;
}
```

The main function (xdp_prog) is composed of a list of if statements, one for each of the filtering rules. Each rule is then expressed as another list of if statements which represent all the conditions that the packet must meet to be discarded; in case any of the statements is false, the rule is not a match for the packet and the next rule is evaluated.

If instead a rule happens to be a match two more actions are required. First a small fraction of the traffic that is going to be dropped should be sampled (i.e. collected by the userspace daemon that sends Sflow packets to the location where they will be analysed). This is needed because sampling happens on the edge, and we cannot lose visibility of the traffic while the attacks are being mitigated.

The next action is accounting: by keeping an eBPF map shared with a userspace program it is possible to collect metrics about the number of packets that were dropped by XDP. This can be done simply by using the id of the attack (a unique number we assign to every attack we detect) as key of the map, and by incrementing the associated value each time a packet is dropped. After sampling and accounting, the function will just return XDP_DROP.

Otherwise, if none of the rules are a match, the main function of the XDP program will just return XDP_PASS.

This simple and linear structure is motivated by the fact that the C source of the XDP program will be generated automatically by another script, which will replace bpftools.

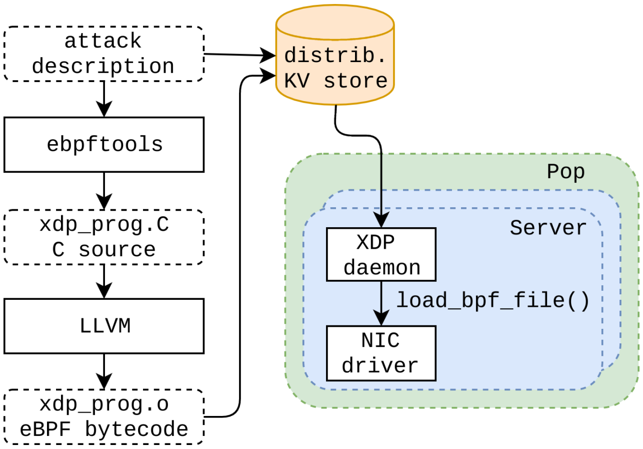

Figure 3: How the attack abstract description is turned into eBPF bytecode and distributed to the edge servers.

This script will accept as input the same abstract description of the attacks that we have already covered in the ”Reacting” section, and will produce a C XDP program file. This source file will be then compiled to eBPF, and the bytecode will be distributed (together with the attacks metadata) to our servers using the same KV store we are already using.

Iptables

In our current setup Iptables is used to filter small floods and to apply connection tracking rules.

With XDP it will be possible to move away the entire filter logic from Iptables. However we still think Iptables is the right tool to apply connection tracking rules, since XDP has no access to the information about the state of TCP connections.

Userspace offload

Our userspace offload utility will not be needed anymore. However we will still need a userspace utility which will:

- listen for rule updates from the KV store (the same we are already using)
- deploy and replace new/updated XDP filtering programs
- collect and expose metrics from the eBPF maps

p0f to eBPF compiler

One of the first projects we adapted to eBPF is our p0f to BPF compiler [4].

p0f[5] is a tool to passively analyse and categorise network traffic. For us one of its most useful features is the signature format it uses to serialise all the meaningful fields of a TCP SYN packet. In practice a signature is represented as a comma separated list of values. A sample one looks like this:

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

We found this format extremely concise yet powerful to describe some kind of attacks, and so we adopted it in our mitigation pipeline. Our original p0f compiler is a Python script that accepts a p0f signature as input and produces BPF bytecode that matches that exact signature. It works by first building a Tcpdump filter for the signature, which is then compiled down to BPF assembly. This BPF bytecode can then be used in Iptables or in our userspace offload program to match specific attack patterns.

Moving from BPF to eBPF brought many improvements:

- the script now emits C code, which is more readable than the previous Tcpdump filter and can be optimized by Clang
- there is no longer a 64 instruction limitation, which can be hit in case of very articulated signatures
- it is easy to combine multiple p0f programs together, since they are just C functions that accept an XDP context and return an XDP action

The following extract is an example of the C code that the p0f to eBPF compiler generates automatically:

```
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;

    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)
        return XDP_PASS;

    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;

    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;
}
```

this can be then compiled with the LLVM eBPF target and used as an XDP program.

Minor issues and solutions

We conclude our article with some of the minor issues we experienced during our initial tests with XDP.

Recent kernel and lack of drivers

Being a new technology, XDP is only supported in recent kernel versions (at least 4.8 is required). At Cloudflare we try to follow the LTS kernel release cycle, which means that using XDP requires us to run at least Linux 4.9. Secondly, we use mostly SolarFlare network cards, whose drivers have currently no support for XDP. At the time of writing (with Linux 4.10 being just released) this applies to most of the other NICs, excluding the Mellanox and QLogic ones.

Sampling packets

XDP does not have support for packets sampling. Currently it is not possible to add a mark to a packet or to make it member of a nflog group. This is because the XDP hook only has access to the raw packet buffer.

It is however possible to ”mark” a packet by manipulating its buffer: setting a specific field that we are not interested in or adding a VLAN tag (which implies shifting the whole packet content of 8 bytes) would allows us to match at a later time the specific packet from Iptables (so it would be possible to maintain the existing sampling architecture based on the NFLOG target). Although functional, this solution is sub-optimal, because it requires to either modify a packet field or to add a header with potential performance implications.

No C standard library

eBPF programs do not have access to the C standard library, which means all the library functions we usually rely on have to be rewritten. Although this may be a problem, in practice we did not find the need for any complex external function. Moreover, the LLVM compiler exposes as `__builtin` some of the functions from the C standard library (such as functions that are part of `string.h`, which for us proved to be the most useful for an XDP program). These functions can be used in a XDP program because the compiler will take care of inlining them.

Conclusion and Future Directions

We think XDP is a promising technology for 2 main reasons:

- it is possible to inspect network packets directly in kernel space and at the lowest possible layer, with a very low cost to drop them, and without resorting to userspace/kernel bypass solutions
- it is possible to express the mitigation logic in a high level language like C (with limitations that in practice do not affect the way XDP programs are written), while maintaining strong safety guarantees about the termination of the program and the memory accesses

As we were expecting, our initial research showed that only a part of our pipeline requires a significative amount changes to support XDP. Moreover, most of the issues we have seen are related to XDP being a relatively new technology, and they do not represent a blocker (expect for the lack of drivers) to our plans to start taking advantage of XDP in production.

We currently do not have any benchmark for XDP on our platform, and this is our primary open question, but as soon as the drivers for our network cards will support it, we will start comparing XDP performance with our current solution based on `EF_VI`.

Acknowledgements

Without the work of the other members of the GateBot team this research would not have been possible: thanks to Marek Majkowski who started and is leading the project, and Chris Branch, who brought many ideas and contributions.

References

[1] Phaal, P.; Lavine M. 2004. sFlow Version 5. `http://www.sflow.org/sflow_version_5.txt`

[2] Majkowski, M. 2014. Introducing the BPF Tools. `https://blog.cloudflare.com/introducing-the-bpf-tools`

[3] Dumazet, E. 2015. Merge branch tcp-lockless-listener. Linux kernel, commit c3fc7ac9a0b97

[4] Bertin, G. 2016. Introducing the p0f BPF compiler. `https://blog.cloudflare.com/introducing-the-p0f-bpf-compiler/`

[5] Zalewski, M. 2014. p0f v3. `http://lcamtuf.coredump.cx/p0f3/`