Issuing SYN Cookies in XDP

Petar Penkov, Eric Dumazet, Stanislav Fomichev
Mountain View, USA
Google
ppenkov@google.com edumazet@google.com sdf@google.com

Abstract

Google's servers are subject to various network attacks, which at large magnitudes may affect customers. One particular type of a distributed denial-of-service (DDoS) attack is TCP SYN flood and SYN cookies have been a popular stateless mechanism for mitigating its impact. Though effective, in Linux this strategy requires traversing a lengthy TCP/IP stack, which significantly limits the rate of issuing SYN cookies, and leaves room for improvement.

In this paper we discuss an XDP-based approach to improving a host's resilience to SYN flood attacks. This approach relies on the early placement of the XDP hook on the networking stack to bypass much of the existing overhead. In addition to the performance advantages, this solution grants better visibility into ongoing attacks.

Keywords

Linux, eBPF, BPF, TCP, SYN Cookies, SYN Flood, XDP

Introduction

Google's servers experience attacks of various magnitudes, the largest of which can have significant negative impact for our customers. The emergence of the eXpress Data Path (XDP) and the extended Berkeley Packet Filter (eBPF) have opened up a world of possibilities for enhancing the Linux kernel and, in particular, for improving the robustness of our servers and performance of our defenses during DDoS attacks.

Motivation

SYN Flood

A SYN flood is an attack during which an attacker seemingly tries to open many connections with a server by sending TCP SYN packets, but never establishes the complete connections by replying to the server's SYN-ACK packets. This forces the server to maintain state for each of the half-open connections, which eventually exhausts system resources and disrupts legitimate traffic.

SYN Cookies

SYN cookies are a technique used to mitigate this attack by choosing a special initial sequence number as a function of the current time, represented by a timestamp **t**, the Maximum Segment Size **m**, and the tuple of TCP source and destination addresses and ports. The 32-bit sequence number of the SYN-ACK packet is constructed as follows:

- The top five bits are chosen as the last 5 bits of **t**.
- The following 3 bits are a mapping of **m**.
- The last 24 bits are the cryptographic hash of **t** and the source and destination addresses and ports.

When the corresponding ACK arrives, the host needs to recompute the cookie and verify the result matches the acknowledgement number.

Problem

Even though SYN cookies eliminate the need to maintain local state for opening connections during SYN flood attacks by encoding this state in the transmitted packet, issuing these cookies is expensive as it requires multiple memory allocations and a traversal of various layers (GRO, RFS, RPS, TCP/IP stacks). Therefore, a DoS might still be achieved if the server spends most of its CPU time issuing these cookies, which ultimately disrupts legitimate traffic. The following section details how XDP can be used to reduce the overhead of issuing SYN cookies in the kernel.

Design Overview

Kernel Support

Because generating a SYN cookie requires access to a cryptographic key, internal to the kernel, we defined a kernel helper function `bpf_tcp_gen_syncookie` to aid creating the SYN-ACK packet. [1]

This function takes the following 5 parameters:

- `struct bpf_sock *sk` - listener socket, retrieved via `bpf_skc_lookup_tcp()`. Because this function takes as a parameter the TCP header of the packet, full dissection to the transport header is needed. It is important to note that we cannot retrieve the socket via `bpf_sk_lookup_tcp()` instead as it does not retrieve timewait and request sockets.
- `struct iphdr *iph` - pointer to the beginning of the IP header inside the packet.
- `int ip_len` - total length of the IP header, including options.

- `struct tcphdr *th` - pointer to the beginning of the TCP header inside the packet.
- `int th_len` - total length of the TCP header, including options.

This function verifies the incoming packet is indeed a TCP SYN packet, that the headers are of the correct length, and that a SYN cookie is needed. This behavior is controlled by the `net.ipv4.tcp_syncookies` sysctl. If the function successfully generates a SYN cookie, it returns a 64-bit value with the cookie as its lower 32 bits, and the MSS as the upper 32 bits. Otherwise, a negative value is returned and the packet is passed to the kernel unmodified.

This function is present in kernels after 5.4.

BPF Program

The XDP program that generates SYN cookies adopts the following outline:

- Parse the packet to the TCP header.
- If it is not a TCP SYN packet, the unmodified packet is passed to the network stack via `XDP_PASS`.
- Otherwise, the XDP program looks up the corresponding socket, and invokes `bpf_tcp_gen_syncookie()` to get the sequence number for the SYN-ACK.
- If the returned value indicates that a SYN cookie was not generated, then the packet is passed unmodified to the kernel networking stack via `XDP_PASS`.
- If a SYN cookie was generated, the program will modify the packet headers in place to create the corresponding SYN-ACK reply, and send it back out on the same NIC via `XDP_TX` or `XDP_REDIRECT`.
- Additionally, the program exports counters for the numbers of SYN cookies issued on each port. There is much flexibility in the metrics that can be exported but per-port counters are both low-cost and an improvement over the in-kernel metrics.

Bimodal Operation

Even though this approach improves the performance of a host during an attack, the majority of the time any given host is not under a SYN flood. However, if the program described in the previous subsection is executed for every packet, it would introduce a small but notable regression. To mitigate this, we define two modes of operation for the BPF program (Figure 1) to only enable the defense when needed.

Passive Mode While in this mode, the assumption is that the host is not under an attack. The program skips most packets and only tries to issue a SYN cookie if a timer T_1 has expired. If issuing this cookie was successful, the program enters Flood Mode, as described below. Otherwise, the timer is reset. Because only a few short checks are done for the majority of the traffic, the overhead is significantly smaller. Small values of T_1 would improve responsiveness to SYN flood attacks, but would make the overhead more visible to user traffic. Conversely, large values of T_1 lead to small impact to user traffic and slower response time to an ongoing attack.

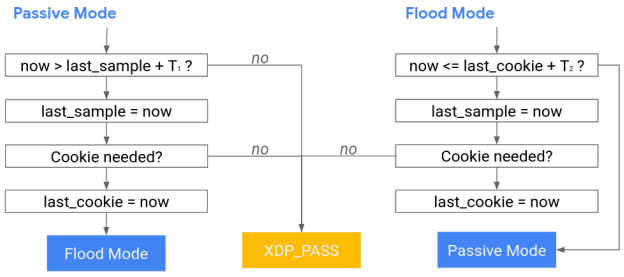

Figure 1: Relationship between the two modes of operation

Flood Mode While in this mode, the BPF program attempts to issue a SYN cookie for every incoming packet. If a SYN cookie has not been issued successfully for a time T_2, the program falls back to Passive Mode. A large T_2 would prevent the BPF program from quickly exiting this mode even after it is no longer subject to an attack, therefore imposing an unnecessary overhead to user traffic.

Implementation One possible implementation of this logic is as follows:

```
/* global variables */
const u64 passive_timer; // T_1
const u64 flood_timer; // T_2
u64 last_sample;
u64 last_cookie;

bool skip() {
  u64 now = bpf_ktime_get_ns();
  if (now < last_cookie + flood_timer) {
    // Flood mode: never skip
    last_sample = now;
    return true;
  }

  if (now > last_sample + passive_timer) {
    // Passive mode: don't skip
    last_sample = now;
    return true;
  }

  // Passive mode: skip
  return false;
}
```

Evaluation

To quantify the advantages of the XDP approach, we compared the performance of 100 TCP_CRR-style flows while the host is subjected to SYN floods of various magnitudes. Figure 2 displays the achieved throughput, as percentage of the maximum achieved throughput, at different flood rates, again displayed as percentage of the maximum achieved flood rate. The graph demonstrates two key results:

- The maximum SYN flood rate a host can absorb in XDP is about 40% higher than without XDP.
- The impact on legitimate flows is much less significant at higher flood rates when issuing SYN cookies in XDP.

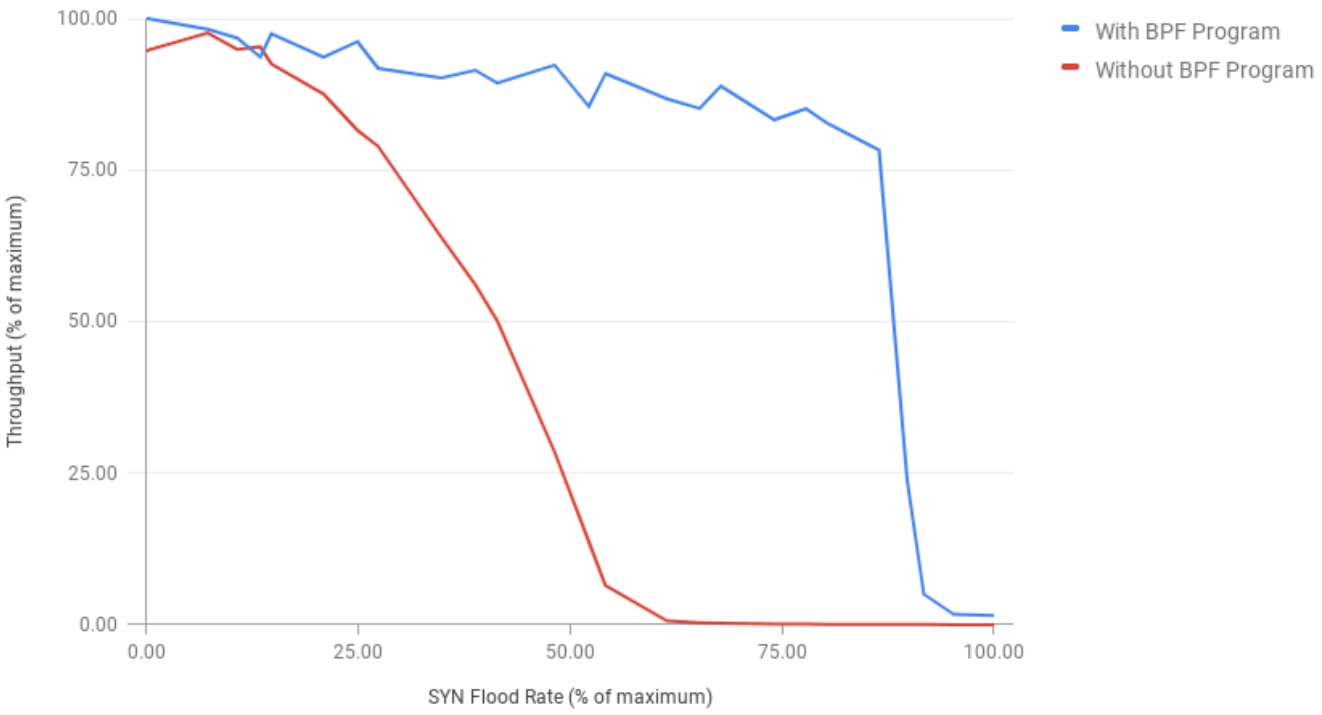

Figure 2: 100-flow throughput at different SYN flood rates compared between kernel and XDP SYN cookie generation

Challenges and Future Work

One of the limitations of using XDP today is that there is no agreed upon solution for supporting multi-buffer packets. [2] In this particular case, this means we cannot attach the SYN cookie BPF program on the driver XDP hook if the driver supports packet header split. A workaround for this issue is to attach the BPF program to either TC ingress or generic XDP, both of which trade back some of the performance benefits.

One clear area for future work is to extend the BPF program so it verifies that an incoming SYN-ACK carries a correct SYN cookie. While the support for this functionality is already present in the Linux Kernel, it is unclear how to propagate this information to the TCP/IP stack so a connection can be established without duplicating the SYN cookie verification. [3]

Conclusion

Issuing SYN cookies in XDP significantly improves the resilience of the networking stack to SYN flood attacks both in terms of peak SYN flood absorption rate, and impact on customer traffic. Additionally, the BPF nature of the implementation offers better visibility to ongoing attacks and higher velocity for configuration changes, new metrics, and bug fixes.

Acknowledgements

Thanks to Willem de Bruijn, Jason Zhang, Wei Wang, Mahesh Bandewar, Luigi Rizzo, Vlad Dumitrescu, and everyone else for their continued feedback and support in the design of this work.

References

[1] P. Penkov, "[bpf-next,v2 3/6] bpf: add bpf_tcp_gen_syncookie helper." [Online]. Available: https://lore.kernel.org/netdev/20190729165918.92933-4-ppenkov.kernel@gmail.com/

[2] J. Brouer, "XDP multi buffer design." [Online]. Available: https://github.com/xdp-project/xdp-project/blob/master/areas/core/xdp-multi-buffer01-design.org

[3] L. Bauer, "[PATCH bpf-next v3 4/8] bpf: add helper to check for a valid SYN cookie." [Online]. Available: https://lore.kernel.org/netdev/20190322015406.26453-5-lmb@cloudflare.com/