Accelerating Linux Security with eBPF iptables

Original

Availability:
This version is available at: 11583/2712606 since: 2018-09-19T11:56:00Z

Publisher:
ACM

Published
DOI:10.1145/3234200.3234228

Terms of use:
openAccess
This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
acm_draft
© (Owner/Author | ACM) (Year). This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in {Source Publication}, http://dx.doi.org/10.1145/{number}

(Article begins on next page)
Accelerating Linux Security with eBPF iptables

Matteo Bertrone, Sebastiano Miano, Fulvio Risso, Massimo Tumolo
Department of Control and Computer Engineering, Politecnico di Torino, Italy

1 INTRODUCTION

Nowadays, the traditional security features of a Linux system are centered on iptables, which has been the most used packet filtering mechanism in the Linux kernel for almost 20+ years. However, the increase in network speed and the transformation of the type of applications running in a Linux server has led to the consciousness that the current implementation may not be able to cope with the modern requirements particularly in terms of scalability, as the number of rules is dramatically increasing [2].

In recent years, the extended BPF (eBPF) subsystem has been added to the Linux kernel, offering the possibility to execute (almost) arbitrary code when a packet is received or sent, including stateful processing. Notably, this does not require any additional kernel module and offers the possibility to compile and inject this code dynamically, hence facilitating over-the-air updates. The above characteristics make eBPF a perfect candidate to build an iptables clone such as [1], which can be considered more an initial proof-of-concept that filters traffic based on IP addresses than a full iptables replacement. This paper starts from the above activities and presents a first eBPF-based prototype, bpf-iptables, which emulates the iptables filtering semantic and exploits a more efficient matching algorithm. Finally, we evaluate our prototype comparing it with the current implementation of iptables, showing how this allows obtaining a notable advantage in terms of performance particularly when a high number of rules is involved, without requiring custom kernels or invasive software frameworks (e.g., DPDK) that could not be allowed in some scenarios (e.g., servers in large datacenters).

2 BPF-IPTABLES DESIGN

The design of bpf-iptables includes two orthogonal aspects: (i) the strategy adopted to preserve the semantic of the iptables firewall policies, and (ii) the data plane architecture, which is driven by the algorithm used for packet matching. We leave the detailed techniques and implementations to a future paper and focus here on the key design of the bpf-iptables architecture.

Iptables filtering semantics. Iptables filters packets either in the INPUT, FORWARD and OUTPUT chains of the Linux Netfilter [5] framework, which are located in a different position compared to eBPF hooks (Figure 1). In particular, the XDP hook, available only for incoming traffic, is located at the earliest point in the networking stack. Instead, the Traffic

![Figure 1: Netfilter vs eBPF hooks](image1)

![Figure 2: Overall data plane structure](image2)
list of unique values for that field present in the given ruleset, plus a wildcard. Each value in the table is associated to a bitvector of length N, equal to the number of rules, which keeps the list of rules matching that field. Finally, a bitwise AND of the intermediate bitvectors returns the list of matched rules; the one with the highest priority (corresponding to the first rule with 1 in the bitvector) is returned. Each map can be implemented in a different way, based on the field characteristics (e.g., longest prefix match in case of IP addresses and ranges; hash tables for TCP/UDP ports). Per-CPU maps are used whenever possible to avoid cache pollution among different CPU cores and increase the effectiveness of parallel processing of multiple packets on different CPU cores. Thanks to the dynamic code injection of eBPF, we created a matching pipeline that contains the minimum number of processing blocks required to handle exactly the fields required by the current ruleset, avoiding unnecessary processing for unused fields. New processing blocks can be added at runtime if the matching against a new field is required, always keeping the optimal number of programs.

**Data plane architecture.** Figure 2 shows the overall data plane architecture of **bpf-iptables**. When a packet reaches a given port, it triggers the **Ingress Chain Forwarder** program attached to the TC or XDP hook; depending on whether the packet is directed to a local application, it jumps to either the **INGRESS** or **FORWARD** chain, which implement the corresponding matching pipeline. Before entering the respective chains, the packet passes from a first **connection tracking** module (implemented as an additional eBPF program), which associates a state to each packet, so that all subsequent chain rules can be correctly applied. When the packet leaves the chains, as result of an **ALLOW** action, it passes through a second conntrack module that saves the state of the session in its internal eBPF map; depending on the chain, the packet is then delivered to the Linux stack or forwarded to the output port. On the other side, when a local application sends a packet out to a netdevice, it will trigger the **Egress Chain Forwarder** program attached to the TC egress hook that, looking at the source IP of the packet decides to forward it directly to the output port (because the **FORWARD** chain has

---

**REFERENCES**


