eBPF / XDP based firewall and packet filtering

Authors : Anant Deepak, Shankaran Gnanashanmugam, Richard Huang, Puneet Mehra

Abstract

iptables have been the typical tool to create firewall for linux hosts. We have used them at Facebook for setting up host firewalls on our servers across a variety of tiers. While they are tried and tested in their ability to filter traffic, there are aspects of its internal implementation which prompted us to examine the packet filtering strategy for our infrastructure. In this paper, we present our solution of implementing a XDP based packet filter which works in functional parity to the input iptables based firewall. We will present the motivation and benefits of this implementation. Further, we apply learnings from this implementation to propose a generic way to translate a given input iptable rule set to an eBPF based version which can then have the same benefit as our custom solution without requiring a new mechanism to specify the filtering policy or requiring the user to maintain custom code. It is assumed that the reader is well versed with BPF and XDP basic and we refer the user to the widely available literature for background reading on this matter. [1][2]

Introduction

In the next section, we will first visit how one would use iptables for input packet filtering. We will visit certain characteristics of configuring them and some performance issues inherent to their implementation in the kernel. Next we showcase our XDP implementation which filters incoming packets with the same policy. We will dive into the specifics of the implementation, its performance and manageability wins and also illustrate how it fits with our overall networking solution on the host. Lastly, we propose a generic way to implement input iptables rules in a BPF program. We discuss the design based on a proof of concept implementation and discuss the performance.

IPTABLES

For the purpose of this paper, when we talk about iptables we talk strictly in the context of INPUT rules. While the concepts are applicable to FORWARD chain, we do not delve into it. OUTPUT chains solve a different use case and our solution does not apply in those cases.

iptable rules are a list of ordered rules that specify some criteria, typically 5 tuples matching packets and taking some action for those matching packets. The action is most typically PASS or DROP. A default policy or a “catch all” action decides the fate of a packet that does not match any proceeding rule. In our usage of iptables, we write rules to PASS specific kind of traffic and a default DROP rule if the packet matches none of the rules.

Note that the packet is linearly scanning the entire list of rules until it finds the first matching rule. If there is no match, it takes the default action. This behavior presents problems when the list of rules is very long (few hundreds of rules). A packet matching the first rule will yield a much quicker disposition than a packet which has to be compared to each of the rules in the list only to take the default drop action. A packet stream with high pps targeting the first rule v/s the default drop rule can manifest into significantly different system behavior in terms of processing latency and processing cycles.

Further, iptables, part of netfilter, is a core part of the linux kernel networking stack. Specific to our infrastructure where we use XDP mode programs to perform various networking operations like load-balancing [3]. This is too late in the packet processing chain. While we can enforce the filtering at the destination that the load balancer is targeting, it is wasteful to load balance traffic that will eventually be dropped at the target host.

Lastly, while iptables presents itself as a linear list of rules, where each rule specifies packet matching criteria, most large deployments implement a high level policy for filtering. They have tooling to translate their policy to rules specific to

their network.
For example, a typical internet facing host will have policy which may appear an ordered set of rules :

1. Open specific TCP and UDP destination ports to Internet
2. Open specific TCP and UDP destination ports to a specific set of hosts in an internal domain
3. Open specific TCP and UDP destination ports to all internal subnets
4. Drop everything else

The above represents a policy which rarely changes. But depending on applications and network topology, ports and networks may change often. iptables rules require specifying the matching criteria inline which results into hundreds of rules for a simple policy but a wide set of ports and networks. ipset add-on for iptables can help address this to some extent by clubbing together multiple attributes of the same kind in a logical set. This logical separation of policy and network attributes is what makes our XDP based implementation feasible and manageable to deploy.

XDP FIREWALL

For our XDP based firewall implementation we leverage our logical separation of policy v/s networking attributes for filtering. The BPF C program is a direct representation of the policy. The program parses each incoming packet to the maximum extent possible to extract the 5 tuples. We can even look beyond custom encapsulation schemes specific to our deployment (IP tunnels used by our L4 load balancer). Post parsing, each parsed attribute (or tuple) is looked up against a pre-loaded BPF map. We use the term BPF map generically to refer to a key-value store, but in our case we use arrays for ports, hash table for addresses and LPM tries for network prefixes. The values in each of these is a bitmap representing a logical grouping.

Example:
TCP_PUBLIC_PORT => Destination TCP ports open to the internet
TCP_INTERNAL_NETWORK_PORT => Destination TCP ports for internal applications open to the entire internal network

We choose bitmaps so if an attribute belongs to multiple profiles we can simply mark those bits set. Using a uint64_t as a value gives us upto 64 unique groupings which is more than sufficient for our use. At the cost of a few more instructions in data-plane, we can logically extend the value to an array of integers to represent more groupings if needed. The combination of the C program and maps loaded with keys as networking attributes and value as a bitmap of logical grouping is best illustrated with some snippets .

C program:

```
int process(struct xdp_md *ctx) {
  // parse packet into tuples
   ...
  // Lookup maps for each attribute
  u64 tcp_dp_match = tcp_port_map_lookup(dport);
   ...
  if (tcp_dp_match & TCP_PUBLIC) {
    XDP_PASS;
  }
   ...
   // DEFAULT drop
   XDP_DROP;
}
```

BPF Map for ports :

	Key	Value	Enum Identifier for value
1	80	0x0001	TCP_PUBLIC
2	443	0x0001	TCP_PUBLIC
3	12345	0x0010	TCP_INTERNAL
4	12346	0x0020	TCP_DOMAIN
5	20000	0x0030	TCP_INTERNAL | TCP_DOMAIN

A similar strategy is used for ports, protocols, source and destination addresses (and prefixes). This bitmap strategy allows us to keep the C program simple and something that rarely needs to be changed. Most firewall changes are when applications are added or topologies change, so new ports / addresses need to be added, removed or modified. We simply create a new program with the new maps, load the maps with the updated key-values (values being enums which are shared with the BPF program as C identifiers) and swap the old program with the new one to mimic iptables-restore behavior.

Note : Longest Prefix Matching tables for network prefixes need special attention. Loaders need to check for overlapping prefixes and have to ensure that they mark overlapping prefixes with the logical OR of all the possible prefixes it maybe overlapping in the table. This keeps data-plane simple as it will simply determine the longest prefix match, but now our loader would have made sure that this match is indicative of any wider subnets by returning a value which has all those bits set in the bitmap and hence a match against various enums.

OUR DEPLOYMENT

In this section we describe some deployment specific details in our infrastructure and provide suggestions if a different behavior is desired. Our firewalls are stateless. Stateful firewalls may need their own connection tracking table and perhaps some more logic to achieve their objectives. We look beyond tunneled IP headers which is how our L4 load balancer sends packets towards the target host.

We run our BPF firewall program in XDP mode. We have a BPF program array which chains a few programs including our L4 load balancer. One of the elements in this chain is our firewall filter. We use PASS and DROP as the only possible dispositions. PASS lets the packet through to the L4 load balancer. REJECT is possible by tail calling into another BPF program which can take appropriate action.

We export statistics for packet pass and drops using per cpu stats array which our user-space module scrubs periodically. We also sample dropped packets and write them to a BPF_PERF_OUTPUT pipe which our user-space module drains and logs for analysis.

From a performance point of view, we see huge wins for packet streams which hit the default deny rule when compared to iptables based firewalling. With the XDP based implementation each packet does lookups against arrays / hash / tries for relevant tuples which are all optimized implementations for lookups. What was previously in iptables a linear scan for each packet through an ordered list of rules, is now practically the same amount of processing for any type of packet irrespective of rule matched or a drop.

We do not write or update these maps once the program is loaded to avoid any lock contentions. Instead, for any change in configuration, we create a new program with new maps and modify the XDP program in the program array. Lastly, by running this before our load balancer, we also ensure that we do not waste resources for load balancing traffic that we do not want to accept in our infrastructure.

GENERICAL IPTABLES IN BPF

Our solution above requires us to maintain a BPF C program which represents our filtering policy and a user-space agent which loads the BPF maps and does other things like collect stats and dropped packet samples. Our implementation however can still be logically expressed by the iptables rules where we first started. Should there be a way to transparently translate a given set of iptable rules into a BPF program with some customization for point of attachment (XDP mode, or TC mode for containers), we could get rid of all the custom code we currently maintain and enjoy the same benefits.

In this section we propose a generic way to express iptable rules with a generic BPF C program. The overall scheme remains similar. We parse each incoming packets to identify the maximum possible packet attributes that we will classify on (5 tuples for most cases). We lookup BPF maps for each of these and collect results. The disposition of the packet is then based on the cumulative content of these results.

Here is how the solution works:
Each tuple or attribute of a packet that can be present in a rule is represented by a unique BPF Map (typically 5 tuples). For most use cases that means, source and destination addresses (and prefixes), protocol and source and destination ports.

For each such map, keys are the specific value of the tuple (port map has keys = 80, 443, etc) and the values for each of these is a simple bit map indicating the rule number where this specific value appeared. A “1” in a given position in the bitmap would mean the value is a match for that specific rule in the list. A “0” would mean otherwise. Since not all rules will specify the entire list of tuples, any rule where that key type (or tuple) is not specified at all is marked as a “1” - a match.

For example: A rule :
```
  [Rule-num-x] : ACCEPT tcp ::/0 ::/0 tcp dpt:80 flags:0x17/0x02
```
is accepting any TCP SYN packet on destination port 80 from anywhere destined to anywhere. It only specifies the TCP dest port and flag. All others fields are wildcards. If the position of this rule is “X”, then for all those other fields each value in those maps will be marked with a “1” in position “X” - implying a match. Only dest port map with key 80 and TCP flags fields map with key “SYN” will have the X bit position marked as “1”.

Being a bit-map, a list of 1K rules can be represented by 128 Bytes value (or an array of 16 uint64_t)

Once we have all the results with all the bitmaps for each attribute lookup, we simply find the first bit set to find the most specific rule. A separate array (say “Actions”) can hold the result or disposition for each rule.

Lets illustrate this with an example.

Sample BPF Map for Destination port number:

	Key	Value (binary)	Notes
1	80	**1**00011…..1110	Rule 1 match. Rule 5 onwards are wild-cards. Last rule is not a match
2	443	0**1**0011…..1110	Rule 2 match. Rule 5 onwards are wild-cards. Last rule is not a match
3	12345	00**1**011…..1110	Rule 3 match. Rule 5 onwards are wild-cards. Last rule is not a match
4	12346	000**1**11…..1110	Rule 4 match. Rule 5 onwards are wild-cards. Last rule is not a match
5	……..	………………	
6	20000	000011…..111**1**	Last Rule match. Rule 5 onwards are wild-cards
7	key-not-found	000011…..1110	Only the wild-cards are marked as a match.

Similar maps exist for other attributes like source / dest addresses, prefixes, protocols and ports. The BPF C program below illustrates how we collect all the results and find the first matching rule.

C Program :

```
// Parse the packet and lookup against maps for each attribute
// *_res are the results of the lookup

ip4_src_addr_res = lookup_ipv4_src_addr(iph->saddr);
// ..collect _res for all other tuples

#pragma clang loop unroll(full)
  for (rule_word = 0; rule_word < RULE_IDS_MAX_WORDS; rule_word++) {
    u64 rule_id =  ip4_src_addr_res->rule_ids[rule_word];
        rule_id &= ip4_src_lpm_res->rule_ids[rule_word];
        rule_id &= ip4_dst_addr_res->rule_ids[rule_word];
        rule_id &= ip4_dst_lpm_res->rule_ids[rule_word];
        rule_id &= ip6_src_addr_res->rule_ids[rule_word];
        rule_id &= ip6_dst_addr_res->rule_ids[rule_word];
        rule_id &= ip6_src_lpm_res->rule_ids[rule_word];
        rule_id &= ip6_dst_lpm_res->rule_ids[rule_word];
        rule_id &= src_port_res->rule_ids[rule_word];
        rule_id &= dst_port_res->rule_ids[rule_word];
        rule_id &= ip_proto_res->rule_ids[rule_word];
        rule_id &= tcp_flag_res->rule_ids[rule_word];
    if (rule_id) {
      // find first bit set (MSB)
      int rule_num = (rule_word * 64) + get_msb_set(rule_id);
      // Take action
      int action = lookup_action(rule_num);
      if (action == BPFILTER_ACTION_PASS) {
        XDP_PASS;
      }
      if (action == BPFILTER_ACTION_DENY) {
        XDP_DROP;
      }
    }
  }
```

Conclusion

iptables has been the de-facto mechanism for host filtering in linux. With the proliferation of networking capabilities of BPF and specifically XDP, we need to revisit packet filtering and firewall implementation. We have implemented a custom solution that works well for our networking stack and provides remarkable performance characteristics and manageability from an operational point of view. We believe that with our proposal to translate iptables rules into a performant BPF based solution we offer the community a better solution without needing to reimplement or learn a new way of configuring their firewalls.

References

[1] BPF : https://prototype-kernel.readthedocs.io/en/latest/bpf/
[2] XDP : https://cilium.readthedocs.io/en/v1.2/bpf/
[3] Katran : https://github.com/facebookincubator/katran
[4] Accelerating Linux Security with eBPF iptables : https://dl.acm.org/citation.cfm?id=3234228&preflayout=flat