Replacing iptables with eBPF in Kubernetes with Cilium

Cilium, eBPF, Envoy, Istio, Hubble

Michal Rostecki
Software Engineer
mrostecki@suse.com
mrostecki@opensuse.org

Swaminathan Vasudevan
Software Engineer
svasudevan@suse.com
What’s wrong with iptables?
IPtables runs into a couple of significant problems:

- Ip tables updates must be made by recreating and updating all rules in a single transaction.
- Implements chains of rules as a linked list, so all operations are O(n).
- The standard practice of implementing access control lists (ACLs) as implemented by iptables was to use sequential list of rules.
- It’s based on matching IPs and ports, not aware about L7 protocols.
- Every time you have a new IP or port to match, rules need to be added and the chain changed.
- Has high consumption of resources on Kubernetes.

Based on the above mentioned issues under heavy traffic conditions or in a system that has a large number of changes to iptable rules the performance degrades.

Measurements show unpredictable latency and reduced performance as the number of services grows.
Kubernetes uses iptables for...

- kube-proxy - the component which implements Services and load balancing by DNAT iptables rules
- the most of CNI plugins are using iptables for Network Policies
And it ends up like that

New Kubernetes cluster (1.15 w/ kubeadm, Cilium). I expected to find very few iptables rules finally (no kube-proxy etc), but, uh...

I guess it _really_ makes to make sure it DROPs that pattern. Repeating the same iptables rule 6,979 times oughta do it.
<table>
<thead>
<tr>
<th>Chained</th>
<th>Protocol</th>
<th>src</th>
<th>dst</th>
<th>Chain</th>
<th>Protocol</th>
<th>src</th>
<th>dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ICMP</td>
<td>192.168.1.2</td>
<td>192.168.1.3</td>
<td>FORWARD</td>
<td>ICMP</td>
<td>192.168.1.2</td>
<td>192.168.1.3</td>
</tr>
<tr>
<td>2</td>
<td>TCP</td>
<td>192.168.1.2</td>
<td>192.168.1.3</td>
<td>FORWARD</td>
<td>TCP</td>
<td>192.168.1.2</td>
<td>192.168.1.3</td>
</tr>
<tr>
<td>3</td>
<td>UDP</td>
<td>192.168.1.2</td>
<td>192.168.1.3</td>
<td>FORWARD</td>
<td>UDP</td>
<td>192.168.1.2</td>
<td>192.168.1.3</td>
</tr>
</tbody>
</table>

Note: The above table represents the chains and rules configured in a Linux firewall. Each row indicates a different chain and its associated rules. The columns indicate the protocol type, source address, destination address, and action.
What is BPF?
The Linux kernel stack is split into multiple abstraction layers.

Strong userspace API compatibility in Linux for years.

This shows how complex the Linux kernel is and its years of evolution.

This cannot be replaced in a short term.

Very hard to bypass the layers.

Netfilter module has been supported by Linux for more than two decades and packet filtering has to be applied to packets that move up and down the stack.
BPF kernel hooks

- Process
- System Call Interface
- Sockets
  - TCP
  - UDP
  - Raw
- Netfilter
  - IPv4
  - IPv6
- Ethernet
- Traffic Shaping
- Netdevice / Drivers
  - HW
  - Bridge
  - OVS

- BPF System calls
- BPF Sockmap and Sockops
- BPF cGroups
- BPF TC hooks
- BPF XDP
BPF goes into firewalls

- Legacy iptables
- nftables
- bpfilter (host driver)
- bpfilter (hardware offload)
BPF replaces IPtables

- **Routing Decision**
- **FILTER**
- **NAT**

- **Netdev** (Physical or virtual Device)
- **eBPF Code**

- **IPTables netfilter hooks**
- **eBPF TC hooks**
- **XDP hooks**

**Local Processes**

**PREROUTING**

**INPUT**

**FORWARD**

**OUTPUT**

**POSTROUTING**

**eBPF replaces IPtables**
BPF based filtering architecture

TC/XDP Ingress hook

NetFilter

To Linux Stack

TC Egress hook

NetFilter

Netdev (Physical or virtual Device)

Netdev (Physical or virtual Device)

Ingress Chain Selector

Update session Label Packet

INGRESS CHAIN

Store session

Egress Chain Selector

Update session Label Packet

FORWARD CHAIN

Store session

Connection Tracking

Update session Label Packet

OUTPUT CHAIN

Store session

[local dst]

[remote dst]

[local src]

[remote src]
BPF based tail calls

LBVS is implemented with a chain of eBPF programs, connected through tail calls.

Each eBPF program is injected only if there are rules operating on that field.

Search first Matching rule

Update counters

ACTION (drop/accept)

Packet in
From eBPF hook

Packet header offsets

Per cpu_array shared across the entire program chain

Bitvector with temporary result

Per cpu_array shared across the entire program chain

Packet out
To eBPF hook

Header parsing is done once and results are kept in a shared map for performance reasons.

Each eBPF program can exploit a different matching algorithm (e.g., exact match, longest prefix match, etc.).

Head parsing
IP:dst lookup
IP.proto lookup
Bitwise AND bit-vectors

eBPF Program #1

eBPF Program #2

eBPF Program #3

eBPF Program #N

Rule 1
Action 1
Cnt 1

Rule 2
Action 2
Cnt 2

Rule 3
Action 3

Rule N

Header parsing
IP:dst lookup
IP.proto lookup

EACH eBPF program is injected only if there are rules operating on that field.

LBVS is implemented with a chain of eBPF programs, connected through tail calls.

Search first Matching rule

Update counters

ACTION (drop/accept)

Packet in
From eBPF hook

Packet header offsets

Per cpu_array shared across the entire program chain

Bitvector with temporary result

Per cpu_array shared across the entire program chain

Packet out
To eBPF hook

Header parsing is done once and results are kept in a shared map for performance reasons.

Each eBPF program can exploit a different matching algorithm (e.g., exact match, longest prefix match, etc.).

Head parsing
IP:dst lookup
IP.proto lookup

EACH eBPF program is injected only if there are rules operating on that field.

LBVS is implemented with a chain of eBPF programs, connected through tail calls.

Search first Matching rule

Update counters

ACTION (drop/accept)

Packet in
From eBPF hook

Packet header offsets

Per cpu_array shared across the entire program chain

Bitvector with temporary result

Per cpu_array shared across the entire program chain

Packet out
To eBPF hook

Header parsing is done once and results are kept in a shared map for performance reasons.

Each eBPF program can exploit a different matching algorithm (e.g., exact match, longest prefix match, etc.).

Head parsing
IP:dst lookup
IP.proto lookup

EACH eBPF program is injected only if there are rules operating on that field.

LBVS is implemented with a chain of eBPF programs, connected through tail calls.

Search first Matching rule

Update counters

ACTION (drop/accept)

Packet in
From eBPF hook

Packet header offsets

Per cpu_array shared across the entire program chain

Bitvector with temporary result

Per cpu_array shared across the entire program chain

Packet out
To eBPF hook

Header parsing is done once and results are kept in a shared map for performance reasons.

Each eBPF program can exploit a different matching algorithm (e.g., exact match, longest prefix match, etc.).

Head parsing
IP:dst lookup
IP.proto lookup

EACH eBPF program is injected only if there are rules operating on that field.

LBVS is implemented with a chain of eBPF programs, connected through tail calls.

Search first Matching rule

Update counters

ACTION (drop/accept)

Packet in
From eBPF hook

Packet header offsets

Per cpu_array shared across the entire program chain

Bitvector with temporary result

Per cpu_array shared across the entire program chain

Packet out
To eBPF hook

Header parsing is done once and results are kept in a shared map for performance reasons.

Each eBPF program can exploit a different matching algorithm (e.g., exact match, longest prefix match, etc.).
BPF goes into...

- Load balancers - katran
- perf
- systemd
- Suricata
- Open vSwitch - AF_XDP
- And many many others
BPF is used by...
Cilium
What is Cilium?
CNI Functionality

CNI is a CNCF (Cloud Native Computing Foundation) project for Linux Containers. It consists of specification and libraries for writing plugins. Only care about networking connectivity of containers:

- ADD/DEL

General container runtime considerations for CNI:

The container runtime must:

- create a new network namespace for the container before invoking any plugins
- determine the network for the container and add the container to the each network by calling the corresponding plugins for each network
- not invoke parallel operations for the same container.
- order ADD and DEL operations for a container, such that ADD is always eventually followed by a corresponding DEL.
- not call ADD twice (without a corresponding DEL) for the same (network name, container id, name of the interface inside the container).

When CNI ADD call is invoked it tries to add the network to the container with respective veth pairs and assigning IP address from the respective IPAM Plugin or using the Host Scope.

When CNI DEL call is invoked it tries to remove the container network, release the IP Address to the IPAM Manager and cleans up the veth pairs.
Kubectl
Kubernetes API Server

Kubelet

CRI-Containerd

CNI-Plugin (Cilium)

Cilium Agent

eth0

BPF Maps

Linux Kernel Network Stack

Container2

Container1

K8s Pod

BPF Hook

Kernel

Userspace

cni-add()

bpf_syscall()

001 54 45 31
002 A1 B1 C1
004 32 66 AA

000 c1 FE 0A
001 04 00 31
002 A1 B1 C1
004 32 66 AA

Cilium CNI Plugin control Flow
Cilium Components with BPF hook points and BPF maps shown in Linux Stack

VM’s and Containers
- VM1
- Cont 1
- Cont 2
- Cont 3

Apps
- App

CILIUM POD (Control Plane)
- CILIUM OPERATOR
- CILIUM HOST_NET
- CILIUM CLUSTER
- CILIUM CLUSTER OPERATOR
- CILIUM CLI
- CILIUM MONITOR
- CILIUM HEALTH
- CILIUM HEALTH NAMESPACE

User Space
- VM’s and Containers
- Apps

Kernel Space
- Virtual Net Devices
- TCP/UDP Layer
- IP Layer
- Queueing and Forwarding
- TC BPF
- BPF (sockmap, sockopts)
- BPF-Cilium
- BPF-Cont1
- BPF-Cont2
- BPF-Cont3
- XDP
- Device Driver
- Bpf_create_map
- BPF maps
- BPF

Build sk_buff

Orchestrator

User Space
- VM’s and Containers
- Apps

Network Stack with BPF hook points

Physical Layer (Network Hardware)
Cilium as CNI Plugin

K8s cluster

K8s node

K8s pod

container A

eth0

lxc0

cilium

Cilium Networking CNI

K8s pod

container B

eth0

lxc0

K8s pod

container C

eth0

lxc1
Networking modes

Encapsulation

Use case:
Cilium handling routing between nodes

Direct routing

Use case:
Using cloud provider routers, using BGP routing daemon
Pod IP Routing - Overlay Routing (Tunneling mode)
Pod IP Routing - Direct Routing Mode
L3 filtering – label based, ingress

Pod
Labels: role=frontend
IP: 10.0.0.1

Pod
Labels: role=frontend
IP: 10.0.0.2

Pod
Labels: role=frontend
IP: 10.0.0.3

Pod
Labels: role=backend
IP: 10.0.0.4

Pod
IP: 10.0.0.5

allow

deny
L3 filtering – label based, ingress

apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
description: "Allow frontends to access backends"
metadata:
  name: "frontend-backend"
spec:
  endpointSelector:
    matchLabels:
      role: backend
  ingress:
  - fromEndpoints:
    - matchLabels:
      class: frontend
L3 filtering – CIDR based, egress

Cluster A

Pod
Labels: role=backend
IP: 10.0.0.1

IP: 10.0.1.1
Subnet: 10.0.1.0/24

IP: 10.0.2.1
Subnet: 10.0.2.0/24

Any IP not belonging to 10.0.1.0/24
L3 filtering – CIDR based, egress

apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
description: "Allow backends to access 10.0.1.0/24"
metadata:
  name: "frontend-backend"
spec:
  endpointSelector:
    matchLabels:
      role: backend
  egress:
  - toCIDR:
    - IP: "10.0.1.0/24"
L4 filtering

apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
description: "Allow to access backends only on TCP/80"
metadata:
  name: "frontend-backend"
spec:
  endpointSelector:
    matchLabels:
      role: backend
  ingress:
  - toPorts:
    - ports:
      - port: "80"
        protocol: "TCP"
L4 filtering

Pod
Labels: role=backend
IP: 10.0.0.1

allow
TCP/80

deny
Any other port
L7 filtering – API Aware Security

Pod
IP: 10.0.0.5

Pod
Labels: role=api
IP: 10.0.0.1

GET /articles/{id}

GET /private
L7 filtering – API Aware Security

apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
description: "L7 policy to restrict access to specific HTTP endpoints"
metadata:
  name: "frontend-backend"
endpointSelector:
  matchLabels:
    role: backend
ingress:
  - toPorts:
    - ports:
      - port: "80"
        protocol: "TCP"
rules:
  http:
    - method: "GET"
      path: "/article/\$"
Standalone proxy, L7 filtering

Generating BPF programs for L7 filtering through libcilium.so

Generating BPF programs for L3/L4 filtering

Node A

Envoy

Pod A

Node B

Envoy

Pod B

Generating BPF programs for L3/L4 filtering

VXLAN
Features
Cluster Mesh

Cluster A

Node A
- Pod A
  - Container
  - eth0
  + BPF

Cluster B

Node B
- Pod B
  - Container
  - eth0
  + BPF

Node C
- Pod C
  - Container
  - eth0
  + BPF

External etcd
Istio with cilium and sockmap
Istio - Deferred kTLS

- **K8s cluster**
- **Istio**
  - Pilot/Mixer/Citadel
- **K8s node**
- **K8s pod**
  - Service A
  - Service B
  - *Deferred kTLS encryption*
- **Cilium Networking CNI**
- **External Github Service**
- **External Cloud Network**
Kubernetes Services

BPF, Cilium

- Hash table.

Key

Value

Key

Value

Key

Value

Search  O(1)
Insert  O(1)
Delete  O(1)

Iptables, kube-proxy

- Linear list.
- All rules in the chain have to be replaced as a whole.

Rule 1

Rule 2

...  

Rule n

Search  O(n)
Insert  O(1)
Delete  O(n)
Kubernetes Services - benchmark

- Cilium (BPF)
- kube-proxy (legacy iptables)

Usec

Number of services in cluster
CNI chaining

Policy enforcement, load balancing, multi-cluster connectivity

IP allocation, configuring network interface, encapsulation/routing inside the cluster
Native support for AWS ENI
Hubble is a fully distributed networking and security observability platform for cloud native workloads. It is built on top of Cilium and eBPF to enable deep visibility in a transparent manner.

Hubble provides

- Service dependencies and communication map
- Operational monitoring and alerting
- Application monitoring
- Secure observability

Known limitations of Hubble:

- Hubble is in beta
- Not all components of Hubble are covered by automated testing.
- Architecture is scalable but not all code paths have been optimized for efficiency and scalability yet
The following components make up Hubble:

- **Hubble Agent**
  - The Hubble Agent is what runs on each worker node. It interacts with the Cilium agent running on the same node and serves the flow query API as well as the metrics.

- **Hubble Storage**
  - Hubble storage layer consists of an in-memory storage able to store a fixed number of flows per node.

- **Hubble CLI**
  - The CLI connects to the flow query API of a Hubble agent running on a node and allows to query the flows stored in the in-memory storage using server-side filtering.

- **Hubble UI**
  - The Hubble UI uses the flow query API to provide a graphical service communication map based on the observed flows.
Hubble running on top of Cilium and eBPF
Hubble Service Maps
Hubble HTTP metrics
To sum it up
Why Cilium is awesome?

- It makes disadvantages of **iptables** disappear. And always gets the best from the **Linux kernel**.
- **Cluster Mesh** / multi-cluster.
- Makes **Istio** faster.
- Offers **L7 API Aware filtering** as a Kubernetes resource.
- Integrates with the other popular **CNI plugins** – **Calico**, **Flannel**, **Weave**, **Lyft**, **AWS CNI**.