Cilium Documentation

Release 0.10.1-50-gd2b9fc1

Cilium Authors

Aug 24, 2017

Table of contents:

CHAPTER 1

Introduction to Cilium

What is Cilium?

Cilium is open source software for transparently securing the network connectivity between application services deployed using Linux container management platforms like Docker and Kubernetes.

At the foundation of Cilium is a new Linux kernel technology called BPF, which enables the dynamic insertion of powerful security visibility and control logic within Linux itself. Because BPF runs inside the Linux kernel, Cilium security policies can be applied and updated without any changes to the application code or container configuration.

Why Cilium?

The development of modern datacenter applications has shifted to a service-oriented architecture often referred to as *microservices*, wherein a large application is split into small independent services that communicate with each other via APIs using lightweight protocols like HTTP. Microservices applications tend to be highly dynamic, with individual containers getting started or destroyed as the application scales out / in to adapt to load changes and during rolling updates that are deployed as part of continuous delivery.

This shift toward highly dynamic microservices presents both a challenge and an opportunity in terms of securing connectivity between microservices. Traditional Linux network security approaches (e.g., iptables) filter on IP address and TCP/UDP ports, but IP addresses frequently churn in dynamic microservices environments. The highly volatile life cycle of containers causes these approaches to struggle to scale side by side with the application as load balancing tables and access control lists carrying hundreds of thousands of rules that need to be updated with a continuously growing frequency. Protocol ports (e.g. TCP port 80 for HTTP traffic) can no longer be used to differentiate between application traffic for security purposes as the port is utilized for a wide range of messages across services.

An additional challenge is the ability to provide accurate visibility as traditional systems are using IP addresses as primary identification vehicle which may have a drastically reduced lifetime of just a few seconds in microservices architectures.

By leveraging Linux BPF, Cilium retains the ability to transparently insert security visibility + enforcement, but does so in a way that is based on service / pod / container identity (in contrast to IP address identification in traditional systems) and can filter on application-layer (e.g. HTTP). As a result, Cilium not only makes it simple to apply security policies in a highly dynamic environment by decoupling security from addressing, but can also provide stronger security isolation by operating at the HTTP-layer in addition to providing traditional Layer 3 and Layer 4 segmentation.

The use of BPF enables Cilium to achieve all of this in a way that is highly scalable even for large-scale environments.

Documentation Roadmap

The remainder of this documentation is divided into four sections:

- *Getting Started* : Provides a simple tutorial for running a small Cilium setup on your laptop. Intended as an easy way to get your hands dirty applying Cilium security policies between containers.
- *Concepts* : Describes the components of the Cilium architecture, and the different models for deploying Cilium. Provides the high-level understanding required to run a full Cilium deployment and understand its behavior.
- *Installation Guide* : Details instructions for installing, configuring, and troubleshooting Cilium in different deployment modes.
- *Developer / Contributor Guide* : Give background to those looking to develop and contribute modifications to the Cilium code or documentation.

Getting Help

We use Github issues to maintain a list of Cilium Frequently Asked Questions (FAQ). Check there to see if your question(s) is already addressed.

The best way to get help if you get stuck is to contact us on the Cilium Slack channel.

If you are confident that you have found a bug, or if you have a feature request, please go ahead and create an issue on our bug tracker.

If you are interested in contributing to the code or docs, ping us on Slack or just dive in on Github!

CHAPTER 2

Getting Started

This document serves as the easiest introduction to using Cilium. It is a detailed walk through of getting a single-node Cilium environment running on your machine. It is designed to take 15-30 minutes.

If you haven't read the *Introduction to Cilium* yet, we'd encourage you to do that first.

This document includes three different guides:

- *Getting Started Using Kubernetes*
- *Getting Started Using Docker*
- *Getting Started Using Mesos*

The guides follow the same basic flow. The flow in the Kubernetes variant is more realistic of a production deployment, but is also a bit more complex.

The best way to get help if you get stuck is to ask a question on the Cilium Slack channel. With Cilium contributors across the globe, there is almost always someone available to help.

Getting Started Using Kubernetes

This guide uses minikube to demonstrate deployment and operation of Cilium in a single-node Kubernetes cluster. The minikube VM requires approximately 2 GB of RAM and supports hypervisors like VirtualBox that run on Linux, macOS, and Windows.

If you instead want to understand the details of deploying Cilium on a full fledged Kubernetes cluster, then go straight to *Kubernetes Installation Guide*.

Step 0: Install kubectl & minikube

Install `kubectl` version `>= 1.6.3` as described in the Kubernetes Docs.

Install one of the hypervisors supported by minikube.

Install `minikube >= 0.20` as described on minikube's github page.

Then, boot a minikube cluster with the Container Network Interface (CNI) network plugin enabled:

```
$ minikube start --network-plugin=cni
```

After minikube has finished setting up your new Kubernetes cluster, you can check the status of the cluster by running `kubectl get cs`:

```
$ kubectl get cs
NAME                 STATUS    MESSAGE              ERROR
controller-manager   Healthy   ok
scheduler            Healthy   ok
etcd-0               Healthy   {"health": "true"}
```

If you see output similar to this, you are ready to proceed to installing Cilium.

Step 1: Installing Cilium

The next step is to install Cilium into your Kubernetes cluster. Cilium installation leverages the Kubernetes Daemon Set abstraction, which will deploy one Cilium pod per cluster node. This Cilium pod will run in the `kube-system` namespace along with all other system relevant daemons and services. The Cilium pod will run both the Cilium agent and the Cilium CNI plugin. The Cilium daemonset also starts a pod running Consul as the underlying key-value store.

To deploy the Cilium Daemon Set, run:

```
$ kubectl create -f https://raw.githubusercontent.com/cilium/cilium/master/examples/
↪minikube/cilium-ds.yaml
clusterrole "cilium" created
serviceaccount "cilium" created
clusterrolebinding "cilium" created
daemonset "cilium-consul" created
daemonset "cilium" created
```

Kubernetes is now deploying the Cilium Daemon Set as a pod on all cluster nodes. This operation is performed in the background. Run the following command to check the progress of the deployment:

```
$ kubectl get ds --namespace kube-system
NAME            DESIRED   CURRENT   READY     NODE-SELECTOR   AGE
cilium          1         1         1         <none>          2m
cilium-consul   1         1         1         <none>          2m
```

Wait until the cilium and cilium-consul Deployments shows a `READY` count of `1` like above.

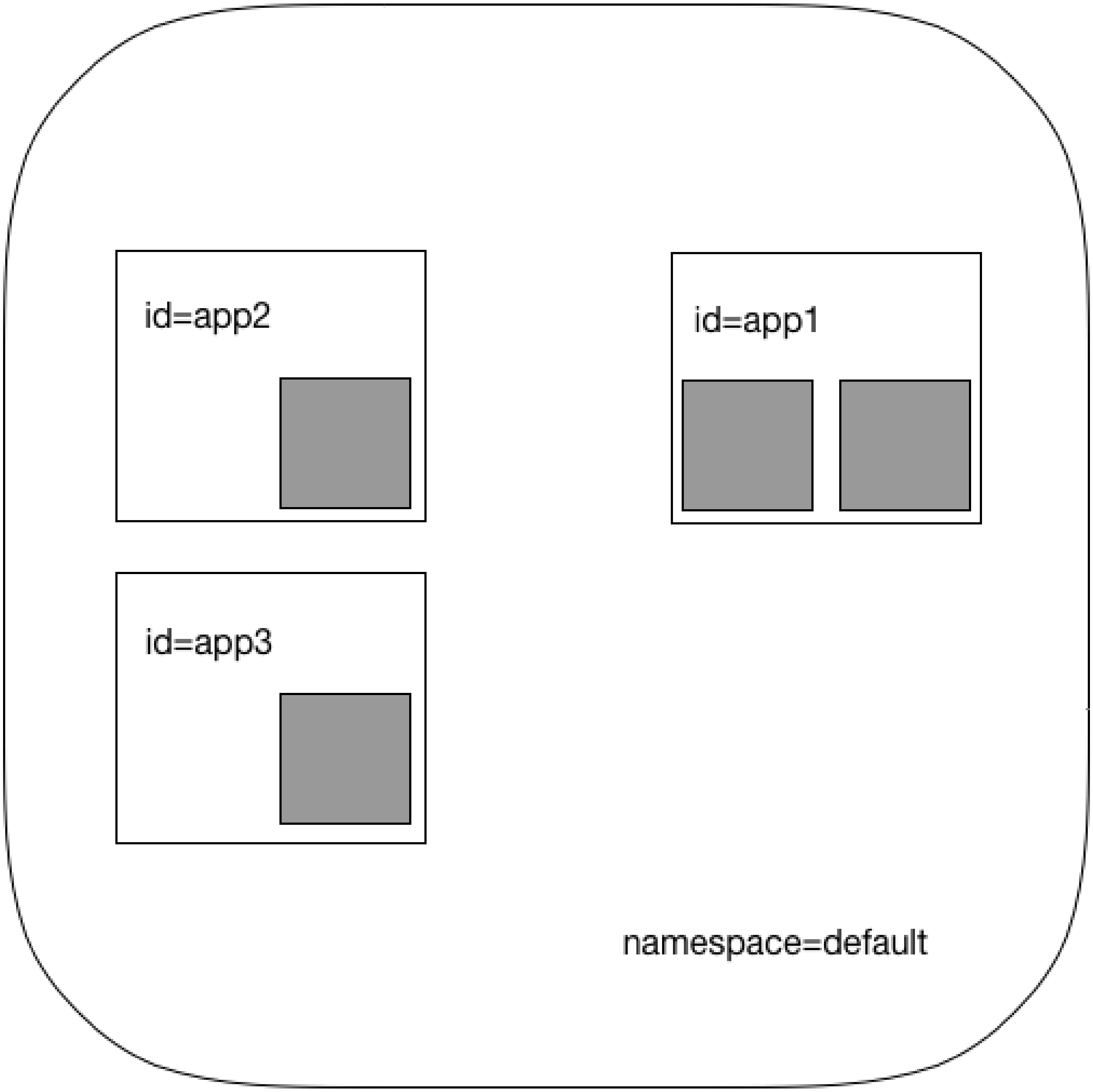
Step 2: Deploy the Demo Application

Now that we have Cilium deployed and `kube-dns` operating correctly we can deploy our demo application.

In our simple example, there are three microservices applications: *app1*, *app2*, and *app3*. *App1* runs an HTTP webservice on port 80, which is exposed as a Kubernetes Service that load-balances requests to *app1* to be across two pod replicas.

App2 and *app3* exist so that we can test different security policies for allowing applications to access *app1*.

Application Topology for Cilium and Kubernetes



The file `demo.yaml` contains a Kubernetes Deployment for each of the three applications, with each deployment identified using the Kubernetes labels id=app1, id=app2, and id=app3. It also include a app1-service, which load-balances traffic to all pods with label id=app1.

```
$ kubectl create -f https://raw.githubusercontent.com/cilium/cilium/master/examples/
↪minikube/demo.yaml
service "app1-service" created
deployment "app1" created
deployment "app2" created
deployment "app3" created
```

Kubernetes will deploy the pods and service in the background. Running `kubectl get svc,pods` will inform you about the progress of the operation. Each pod will go through several states until it reaches `Running` at which point the pod is ready.

```
$ kubectl get pods,svc

NAME                       READY     STATUS              RESTARTS   AGE
po/app1-2741898079-66lz0   0/1       ContainerCreating   0          40s
po/app1-2741898079-jwfmk   1/1       Running             0          40s
po/app2-2889674625-wxs08   0/1       ContainerCreating   0          40s
po/app3-3000954754-fbqtz   0/1       ContainerCreating   0          40s

NAME                CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
svc/app1-service    10.0.0.40     <none>        80/TCP    40s
svc/kubernetes      10.0.0.1      <none>        443/TCP   5h
```

All of these pods will be represented in Cilium as *endpoints*. We can invoke the `cilium` tool inside the Cilium pod to list them:

```
$ kubectl -n kube-system get pods -l k8s-app=cilium
NAME           READY     STATUS    RESTARTS   AGE
cilium-wjb9t   1/1       Running   0          17m
$ kubectl -n kube-system exec cilium-wjb9t cilium endpoint list
ENDPOINT   POLICY        IDENTITY   LABELS (source:key[=value])                IPv6       ␣
↪               IPv4            STATUS
           ENFORCEMENT
3365       Disabled      256        k8s:id=app1                               ␣
↪f00d::a00:20f:0:d25    10.15.191.0     ready
                                    k8s:io.kubernetes.pod.namespace=default
25917      Disabled      258        k8s:id=app3                               ␣
↪f00d::a00:20f:0:653d   10.15.100.129   ready
                                    k8s:io.kubernetes.pod.namespace=default
42910      Disabled      256        k8s:id=app1                               ␣
↪f00d::a00:20f:0:a79e   10.15.236.254   ready
                                    k8s:io.kubernetes.pod.namespace=default
50133      Disabled      257        k8s:id=app2                               ␣
↪f00d::a00:20f:0:c3d5   10.15.59.20     ready
                                    k8s:io.kubernetes.pod.namespace=default
```

Policy enforcement is still disabled on all of these pods because no network policy has been imported yet which select any of the pods.

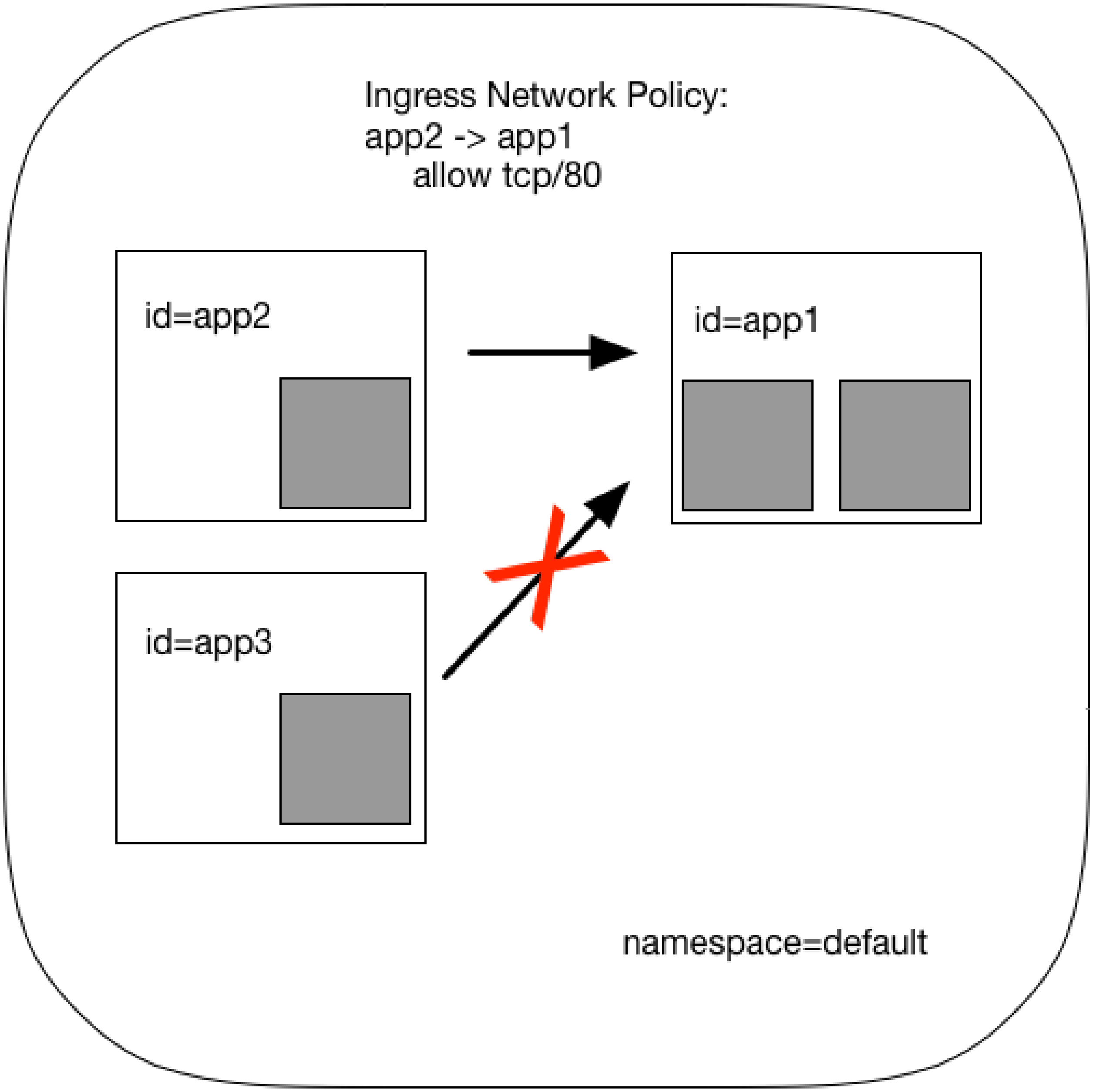
Step 3: Apply an L3/L4 Policy

When using Cilium, endpoint IP addresses are irrelevant when defining security policies. Instead, you can use the labels assigned to the VM to define security policies, which are automatically applied to any container with that label, no matter where or when it is run within a container cluster.

We'll start with a simple example where allow *app2* to reach *app1* on port 80, but disallow the same connectivity from *app3* to *app1*. This is a simple policy that filters only on IP protocol (network layer 3) and TCP protocol (network layer 4), so it is often referred to as an L3/L4 network security policy.

Note: Cilium performs stateful *connection tracking*, meaning that if policy allows the frontend to reach backend, it will automatically allow all required reply packets that are part of backend replying to frontend within the context of the same TCP/UDP connection.

L4 Policy with Cilium and Kubernetes



We can achieve that with the following Kubernetes NetworkPolicy:

```
kind: NetworkPolicy
apiVersion: extensions/v1beta1
metadata:
  name: access-app1
spec:
  podSelector:
    matchLabels:
      id: app1
  ingress:
  - from:
    - podSelector:
        matchLabels:
          id: app2
    ports:
    - protocol: tcp
      port: 80
```

Kubernetes NetworkPolicies match on pod labels using “podSelector” to identify the sources and destinations to which the policy applies. The above policy whitelists traffic sent from *app2* pods to *app1* pods on TCP port 80.

To apply this L3/L4 policy, run:

```
$ kubectl create -f https://raw.githubusercontent.com/cilium/cilium/master/examples/
↪minikube/l3_l4_policy.yaml
```

If we run `cilium endpoint list` again we will see that the pods with the label `id=app1` now have policy enforcement enabled.

```
$ kubectl -n kube-system exec cilium-wjb9t cilium endpoint list
ENDPOINT   POLICY        IDENTITY   LABELS (source:key[=value])               IPv6   ␣
↪                IPv4            STATUS
           ENFORCEMENT
3365       Enabled       256        k8s:id=app1                              ␣
↪f00d::a00:20f:0:d25     10.15.191.0     ready
                                    k8s:io.kubernetes.pod.namespace=default
25917      Disabled      258        k8s:id=app3                              ␣
↪f00d::a00:20f:0:653d    10.15.100.129   ready
                                    k8s:io.kubernetes.pod.namespace=default
42910      Enabled       256        k8s:id=app1                              ␣
↪f00d::a00:20f:0:a79e    10.15.236.254   ready
                                    k8s:io.kubernetes.pod.namespace=default
50133      Disabled      257        k8s:id=app2                              ␣
↪f00d::a00:20f:0:c3d5    10.15.59.20     ready
```

Step 4: Test L3/L4 Policy

We can now verify the network policy that was imported. You can now launch additional containers represent other services attempting to access backend. Any new container with label *id=app2* will be allowed to access the *app1* on port 80, otherwise the network request will be dropped.

To test this out, we’ll make an HTTP request to app1 from both *app2* and *app3* pods:

```
$ APP2_POD=$(kubectl get pods -l id=app2 -o jsonpath='{.items[0].metadata.name}')
$ SVC_IP=$(kubectl get svc app1-service -o jsonpath='{.spec.clusterIP}')
$ kubectl exec $APP2_POD -- curl -s $SVC_IP
<html><body><h1>It works!</h1></body></html>
```

This works, as expected. Now the same request run from an *app3* pod will fail:

```
$ APP3_POD=$(kubectl get pods -l id=app3 -o jsonpath='{.items[0].metadata.name}')
$ kubectl exec $APP3_POD -- curl -s $SVC_IP
```

This request will hang, so press Control-C to kill the curl request, or wait for it to time out.

You can observe the policy via `kubectl`

```
$ kubectl get networkpolicies
NAME              POD-SELECTOR    AGE
access-backend    id=app1         2m
$ kubectl describe networkpolicies access-backend
```

```
Name:               access-backend
Namespace:  default
Labels:             <none>
Annotations:        <none>
```

Step 5: Apply and Test HTTP-aware L7 Policy

In the simple scenario above, it was sufficient to either give *app2* / *app3* full access to *app1's* API or no access at all. But to provide the strongest security (i.e., enforce least-privilege isolation) between microservices, each service that calls *app1's* API should be limited to making only the set of HTTP requests it requires for legitimate operation.

For example, consider an extremely simple scenario where *app1* has only two API calls:

- GET /public
- GET /private

Continuing with the example from above, if *app2* requires access only to the GET /public API call, the L3/L4 policy along has no visibility into the HTTP requests, and therefore would allow any HTTP request from *app2* (since all HTTP is over port 80).

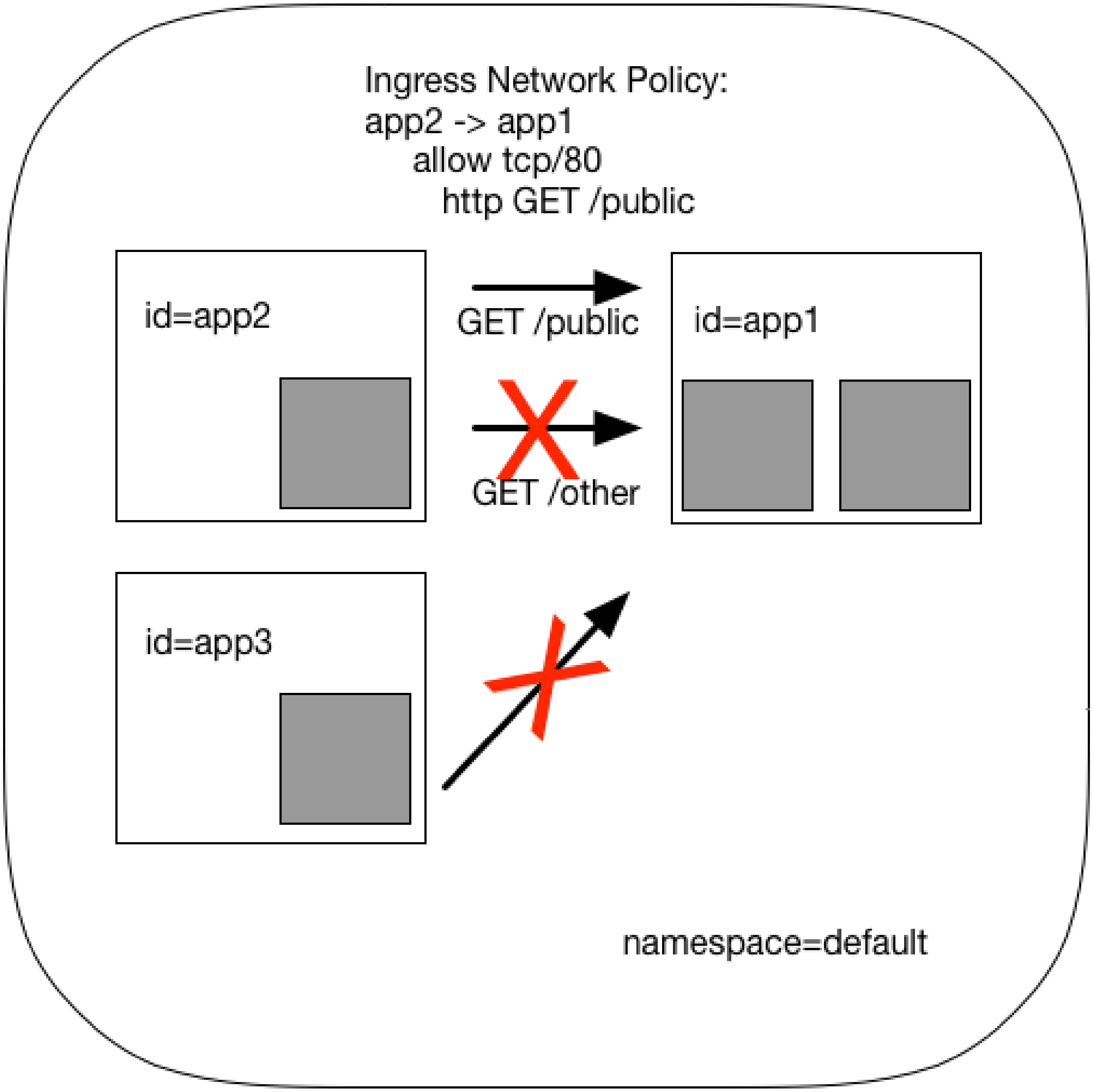
To see this, run:

```
$ kubectl exec $APP2_POD -- curl -s http://${SVC_IP}/public
{ 'val': 'this is public' }
```

and

```
$ kubectl exec $APP2_POD -- curl -s http://${SVC_IP}/private
{ 'val': 'this is private' }
```

L7 Policy with Cilium and Kubernetes



Cilium is capable of enforcing HTTP-layer (i.e., L7) policies to limit what URLs *app2* is allowed to reach. Here is an example policy file that extends our original policy by limiting *app2* to making only a GET /public API call, but disallowing all other calls (including GET /private).

```
apiVersion: "cilium.io/v1"
kind: CiliumNetworkPolicy
description: "L7 policy for getting started using Kubernetes guide"
metadata:
  name: "rule1"
spec:
  endpointSelector:
    matchLabels:
      id: app1
  ingress:
  - fromEndpoints:
    - matchLabels:
        id: app2
    toPorts:
    - ports:
      - port: "80"
        protocol: TCP
      rules:
        HTTP:
        - method: "GET"
          path: "/public"
```

Create an L7-aware policy to protect *app1* using:

```
$ kubectl create -f https://raw.githubusercontent.com/cilium/cilium/master/examples/
↪minikube/l3_l4_l7_policy.yaml
```

Note: If this step is failing with an error complaining about version `cilium.io/v1` not found then you are using a `kubectl` client which is too old. Please upgrade to version >= `1.6.3`.

We can now re-run the same test as above, but we will see a different outcome:

```
$ kubectl exec $APP2_POD -- curl -s http://${SVC_IP}/public
{ 'val': 'this is public' }
```

and

```
$ kubectl exec $APP2_POD -- curl -s http://${SVC_IP}/private
Access denied
```

As you can see, with Cilium L7 security policies, we are able to permit *app2* to access only the required API resources on *app1*, thereby implementing a "least privilege" security approach for communication between microservices.

You can observe the L7 policy via `kubectl`:

```
$ kubectl get ciliumnetworkpolicies
NAME      KIND                                 DESCRIPTION
rule1     CiliumNetworkPolicy.v2.cilium.io     L7 policy for getting started using␣
↪Kubernetes guide
$ kubectl describe networkpolicies access-backend
Name:               access-backend
Namespace:  default
Labels:             <none>
Annotations:        <none>
$ kubectl describe ciliumnetworkpolicies rule1
Name:               rule1
Namespace:  default
Labels:             <none>
Annotations:        <none>
API Version:        cilium.io/v2
Description:        L7 policy for getting started using Kubernetes guide
Kind:               CiliumNetworkPolicy
Metadata:
  Cluster Name:
  Creation Timestamp:                     2017-08-14T17:52:51Z
  Deletion Grace Period Seconds:  <nil>
  Deletion Timestamp:                     <nil>
  Resource Version:                 94966
  Self Link:                                /apis/cilium.io/v2/namespaces/default/
↪ciliumnetworkpolicies/rule1
  UID:                                      6491cbe9-8119-11e7-8dcd-080027babb71
Spec:
  Endpoint Selector:
    Match Labels:
      Id:   app1
  Ingress:
    From Endpoints:
      Match Labels:
        Id: app2
    To Ports:
      Ports:
        Port:               80
        Protocol:   TCP
      Rules:
        Http:
          Method:   GET
          Path:             /public
Events:                     <none>
```

and `cilium` CLI:

```
$ kubectl exec cilium-<xxx> -n kube-system cilium policy get
[
  {
    "endpointSelector": {
      "matchLabels": {
        "k8s:id": "app1",
        "k8s:io.kubernetes.pod.namespace": "default"
      }
    },
    "ingress": [
      {
        "fromEndpoints": [
          {
            "matchLabels": {
              "k8s:id": "app2",
              "k8s:io.kubernetes.pod.namespace": "default"
            }
          }
        ],
        "toPorts": [
          {
            "ports": [
              {
                "port": "80",
                "protocol": "TCP"
              }
            ]
          }
        ]
      }
    ],
    "labels": [
      {
        "key": "io.cilium.k8s-policy-name",
        "value": "access-backend",
        "source": "unspec"
      }
    ]
  },
  {
    "endpointSelector": {
      "matchLabels": {
        "any:id": "app1",
        "k8s:io.kubernetes.pod.namespace": "default"
      }
    },
    "ingress": [
       {
        "fromEndpoints": [
          {
            "matchLabels": {
              "any:id": "app2",
              "k8s:io.kubernetes.pod.namespace": "default"
            }
           }
        ],
        "toPorts": [
          {
            "ports": [
              {
                "port": "80",
                "protocol": "TCP"
              }
            ],
            "rules": {
              "http": [
                {
                  "path": "/public",
                  "method": "GET"
                }
              ]
            }
          }
        ]
      }
    ],
    "labels": [
      {
        "key": "io.cilium.k8s-policy-name",
        "value": "rule1",
        "source": "unspec"
      }
    ]
  }
]
Revision: 26
```

We hope you enjoyed the tutorial. Feel free to play more with the setup, read the rest of the documentation, and reach out to us on the Cilium Slack channel with any questions!

Step 6: Clean-Up

You have now installed Cilium, deployed a demo app, and tested both L3/L4 and L7 network security policies.

```
$ minikube delete
```

After this, you can re-run the *Getting Started Using Kubernetes* from Step 1.

Getting Started Using Docker

This tutorial leverages Vagrant and VirtualBox, thus should run on any operating system supported by Vagrant, including Linux, macOS, and Windows.

Step 0: Install Vagrant

If you don't already have Vagrant installed, refer to the *Developer / Contributor Guide* for links to installation instructions for Vagrant.

Step 1: Download the Cilium Source Code

Download the latest Cilium source code and unzip the files.

Alternatively, if you are a developer, feel free to clone the repository:

```
$ git clone https://github.com/cilium/cilium
```

Step 2: Starting the Docker + Cilium VM

Open a terminal and navigate into the top of the *cilium* source directory.

Then navigate into *examples/getting-started* and run *vagrant up*:

```
$ cd examples/getting-started
$ vagrant up
```

The script usually takes a few minutes depending on the speed of your internet connection. Vagrant will set up a VM, install the Docker container runtime and run Cilium with the help of Docker compose. When the script completes successfully, it will print:

```
==> cilium-1: Creating cilium-kvstore
==> cilium-1: Creating cilium
==> cilium-1: Creating cilium-docker-plugin
$
```

If the script exits with an error message, do not attempt to proceed with the tutorial, as later steps will not work properly. Instead, contact us on the Cilium Slack channel.

Step 3: Accessing the VM

After the script has successfully completed, you can log into the VM using `vagrant ssh`:

```
$ vagrant ssh
```

All commands for the rest of the tutorial below should be run from inside this Vagrant VM. If you end up disconnecting from this VM, you can always reconnect in a new terminal window just by running `vagrant ssh` again from the Cilium directory.

Step 4: Confirm that Cilium is Running

The Cilium agent is now running as a system service and you can interact with it using the `cilium` CLI client. Check the status of the agent by running `cilium status`:

```
$ cilium status
KVStore:            Ok
ContainerRuntime:   Ok
Kubernetes:         Disabled
Cilium:             Ok
```

The status indicates that all components are operational with the Kubernetes integration currently being disabled.

Step 5: Create a Docker Network of Type Cilium

Cilium integrates with local container runtimes, which in the case of this demo means Docker. With Docker, native networking is handled via a component called libnetwork. In order to steer Docker to request networking of a container from Cilium, a container must be started with a network of driver type “cilium”.

With Cilium, all containers are connected to a single logical network, with isolation added not based on IP addresses but based on container labels (as we will do in the steps below). So with Docker, we simply create a single network named ‘cilium-net’ for all containers:

```
$ docker network create --ipv6 --subnet ::1/112 --driver cilium --ipam-driver cilium␣
↪cilium-net
```

Step 6: Start an Example Service with Docker

In this tutorial, we’ll use a container running a simple HTTP server to represent a microservice application which we will refer to as *app1*. As a result, we will start this container with the label “id=app1”, so we can create Cilium security policies for that service.

Use the following command to start the *app1* container connected to the Docker network managed by Cilium:

```
$ docker run -d --name app1 --net cilium-net -l "id=app1" cilium/demo-httpd
e5723edaa2a1307e7aa7e71b4087882de0250973331bc74a37f6f80667bc5856
```

This has launched a container running an HTTP server which Cilium is now managing as an *endpoint*. A Cilium endpoint is one or more application containers which can be addressed by an individual IP address.

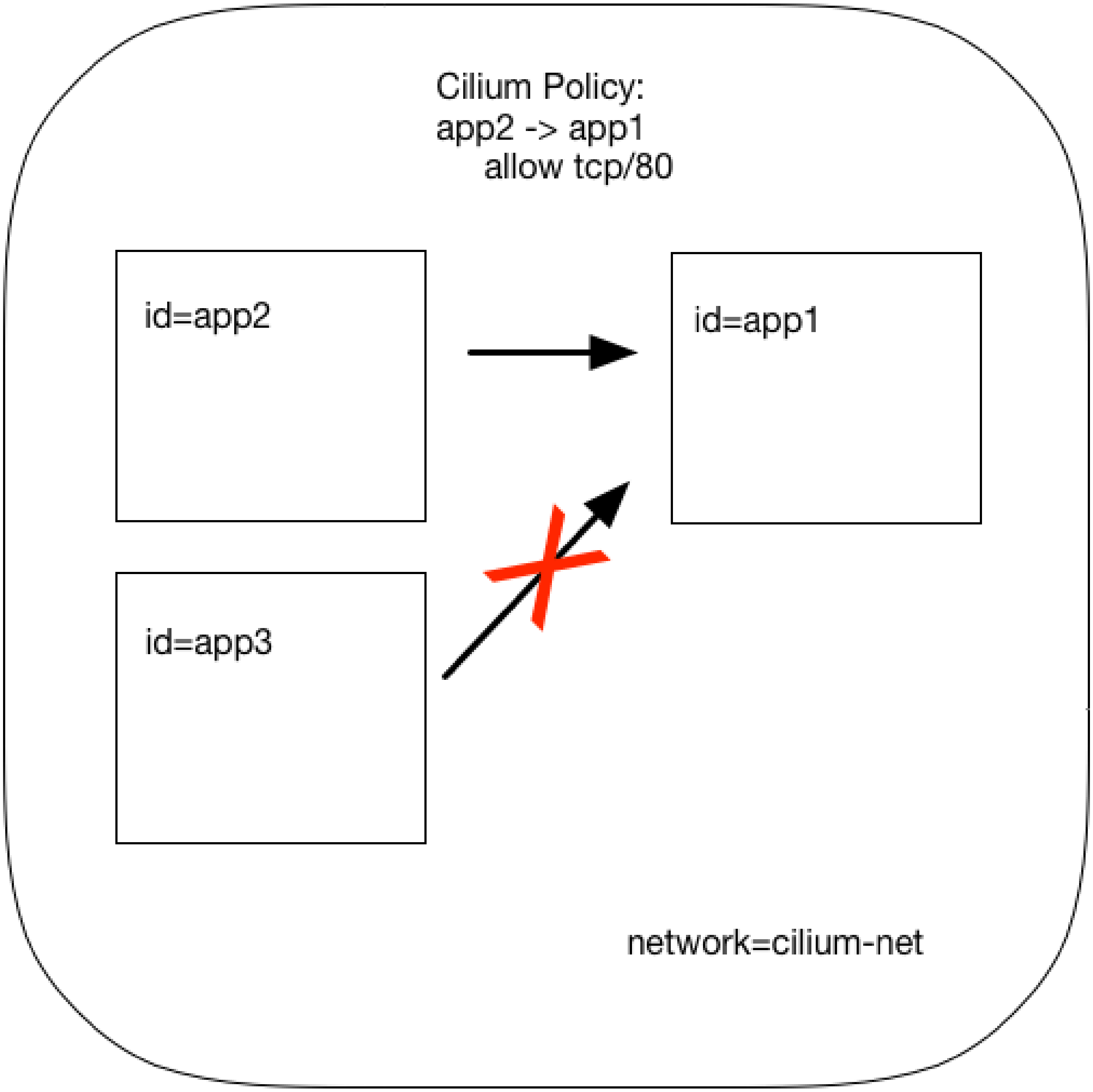
Step 7: Apply an L3/L4 Policy With Cilium

When using Cilium, endpoint IP addresses are irrelevant when defining security policies. Instead, you can use the labels assigned to the VM to define security policies, which are automatically applied to any container with that label, no matter where or when it is run within a container cluster.

We'll start with an overly simple example where we create two additional apps, *app2* and *app3*, and we want *app2* containers to be able to reach *app1* containers, but *app3* containers should not be allowed to reach *app1* containers. Additionally, we only want to allow *app1* to be reachable on port 80, but no other ports. This is a simple policy that filters only on IP address (network layer 3) and TCP port (network layer 4), so it is often referred to as an L3/L4 network security policy.

Cilium performs stateful ''connection tracking'', meaning that if policy allows the *app2* to contact *app3*, it will automatically allow return packets that are part of *app1* replying to *app2* within the context of the same TCP/UDP connection.

L4 Policy with Cilium and Docker



We can achieve that with the following Cilium policy:

```
[{
    "endpointSelector": {"matchLabels":{"id":"app1"}},
    "ingress": [{
        "fromEndpoints": [
            {"matchLabels":{"id":"app2"}}
        ],
        "toPorts": [{
                "ports": [{"port": "80", "protocol": "tcp"}]
        }]
    }]
}]
```

Save this JSON to a file named l3_l4_policy.json in your VM, and apply the policy by running:

```
$ cilium policy import l3_l4_policy.json
```

Step 8: Test L3/L4 Policy

You can now launch additional containers represent other services attempting to access *app1*. Any new container with label “id=app2” will be allowed to access *app1* on port 80, otherwise the network request will be dropped.

To test this out, we’ll make an HTTP request to *app1* from a container with the label “id=app2” :

```
$ docker run --rm -ti --net cilium-net -l "id=app2" cilium/demo-client curl -m 10␣
↪http://app1
<html><body><h1>It works!</h1></body></html>
```

We can see that this request was successful, as we get a valid ping responses.

Now let’s run the same ping request to *app1* from a container that has label “id=app3”:

```
$ docker run --rm -ti --net cilium-net -l "id=app3" cilium/demo-client curl -m 10␣
↪http://app1
```

You will see no reply as all packets are dropped by the Cilium security policy. The request will time-out after 10 seconds.

So with this we see Cilium’s ability to segment containers based purely on a container-level identity label. This means that the end user can apply security policies without knowing anything about the IP address of the container or requiring some complex mechanism to ensure that containers of a particular service are assigned an IP address in a particular range.

Step 9: Apply and Test an L7 Policy with Cilium

In the simple scenario above, it was sufficient to either give *app2* / *app3* full access to *app1’s* API or no access at all. But to provide the strongest security (i.e., enforce least-privilege isolation) between microservices, each service that calls *app1’s* API should be limited to making only the set of HTTP requests it requires for legitimate operation.

For example, consider a scenario where *app1* has two API calls:

- GET /public
- GET /private

Continuing with the example from above, if *app2* requires access only to the GET /public API call, the L3/L4 policy along has no visibility into the HTTP requests, and therefore would allow any HTTP request from *app2* (since all HTTP is over port 80).

To see this, run:

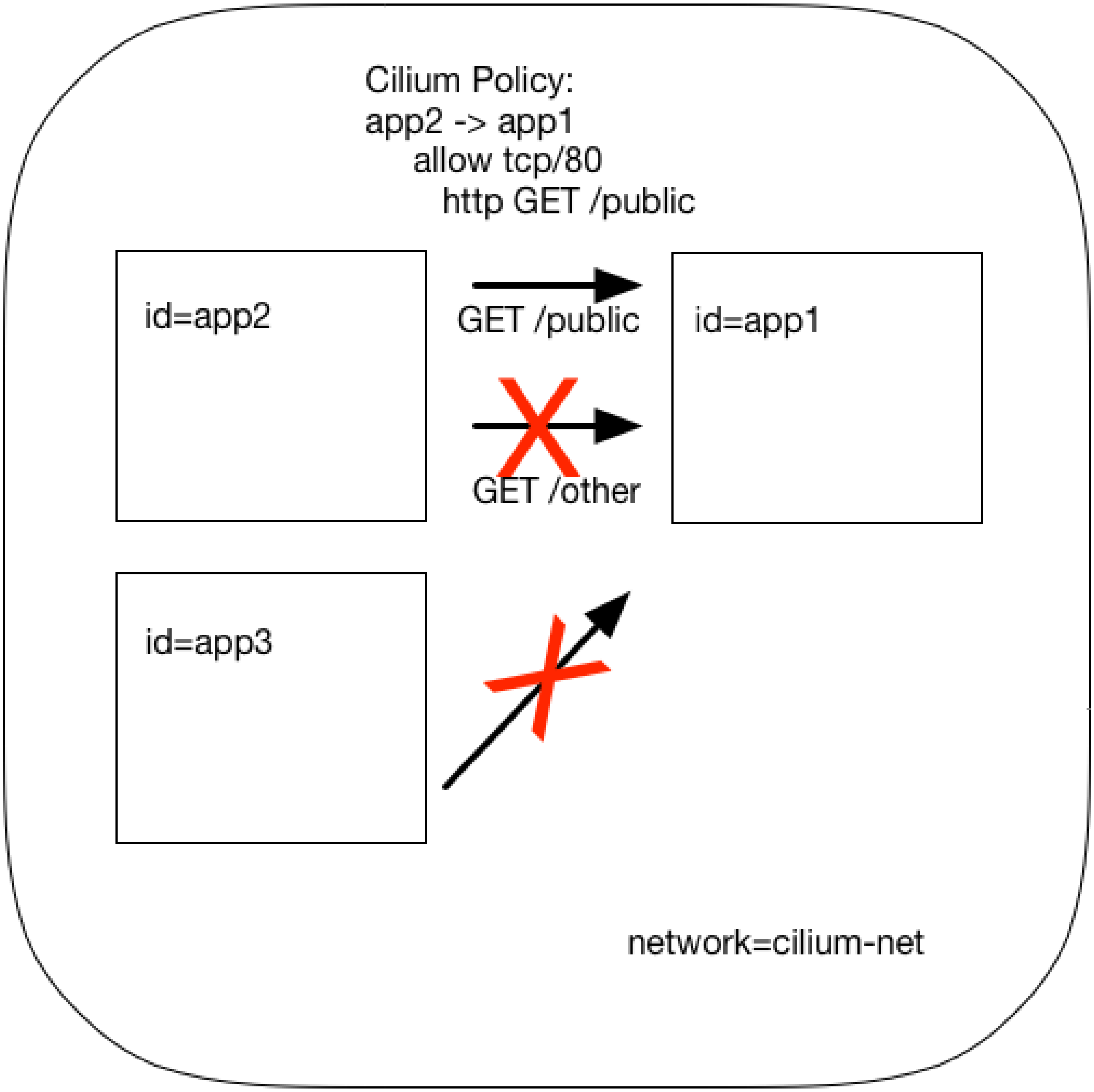
```
$ docker run --rm -ti --net cilium-net -l "id=app2" cilium/demo-client curl 'http://
↪app1/public'
{ 'val': 'this is public' }
```

and

```
$ docker run --rm -ti --net cilium-net -l "id=app2" cilium/demo-client curl 'http://
↪app1/private'
{ 'val': 'this is private' }
```

Cilium is capable of enforcing HTTP-layer (i.e., L7) policies to limit what URLs *app2* is allowed to reach. Here is an example policy file that extends our original policy by limiting *app2* to making only a GET /public API call, but disallowing all other calls (including GET /private).

L7 Policy with Cilium and Docker



The following Cilium policy file achieves this goal:

```
[{
    "endpointSelector": {"matchLabels":{"id":"app1"}},
    "ingress": [{
        "fromEndpoints": [
            {"matchLabels":{"id":"app2"}}
        ],
        "toPorts": [{
            "ports": [{"port": "80", "protocol": "tcp"}],
            "rules": {
                "HTTP": [{
                    "method": "GET",
                    "path": "/public"
                }]
            }
        }]
    }]
}]
```

Create a file with this contents and name it l7_aware_policy.json. Then import this policy to Cilium by running:

```
$ cilium policy delete --all
$ cilium policy import l7_aware_policy.json
```

```
$ docker run --rm -ti --net cilium-net -l "id=app2" cilium/demo-client curl -si
↪'http://app1/public'
{ 'val': 'this is public' }
```

and

```
$ docker run --rm -ti --net cilium-net -l "id=app2" cilium/demo-client curl -si
↪'http://app1/private'
Access denied
```

As you can see, with Cilium L7 security policies, we are able to permit *app2* to access only the required API resources on *app1*, thereby implementing a “least privilege” security approach for communication between microservices.

We hope you enjoyed the tutorial. Feel free to play more with the setup, read the rest of the documentation, and reach out to us on the Cilium Slack channel with any questions!

Step 10: Clean-Up

Exit the vagrant VM by typing `exit`.

When you are done with the setup and want to tear-down the Cilium + Docker VM, and destroy all local state (e.g., the VM disk image), open a terminal in the cilium/examples/getting-started directory and type:

```
$ vagrant destroy cilium-1
```

You can always re-create the VM using the steps described above.

If instead you just want to shut down the VM but may use it later, `vagrant halt cilium-1` will work, and you can start it again later using the contrib/vagrant/start.sh script.

Getting Started Using Mesos

This tutorial leverages Vagrant and VirtualBox to deploy Apache Mesos, Marathon and Cilium. You will run Cilium to apply a simple policy between a simulated web-service and clients. This tutorial can be run on any operating system supported by Vagrant including Linux, macOS, and Windows.

For more information on Apache Mesos and Marathon orchestration, check out the Mesos and Marathon GitHub pages, respectively.

Step 0: Install Vagrant

You need to run at least Vagrant version 1.8.3 or you will run into issues booting the Ubuntu 16.10 base image. You can verify by running `vagrant --version`.

If you don't already have Vagrant installed, follow the Vagrant Install Instructions or see Download Vagrant for newer versions.

Step 1: Download the Cilium Source Code

Download the latest Cilium source code and unzip the files.

Alternatively, if you are a developer, feel free to clone the repository:

```
$ git clone https://github.com/cilium/cilium
```

Step 2: Starting a VM with Cilium

Open a terminal and navigate into the top of the `cilium` source directory.

Then navigate into `examples/mesos` and run `vagrant up`:

```
$ cd examples/mesos
$ vagrant up
```

The script usually takes a few minutes depending on the speed of your internet connection. Vagrant will set up a VM, install the Docker container runtime and run Cilium with the help of Docker compose. When the script completes successfully, it will print:

```
==> default: Creating cilium-kvstore
Creating cilium-kvstore ... done
==> default: Creating cilium ...
==> default: Creating cilium
Creating cilium ... done
==> default: Installing loopback driver...
==> default: Installing cilium-cni to /host/opt/cni/bin/ ...
==> default: Installing new /host/etc/cni/net.d/10-cilium.conf ...
==> default: Deploying Vagrant VM + Cilium + Mesos...done
$
```

If the script exits with an error message, do not attempt to proceed with the tutorial, as later steps will not work properly. Instead, contact us on the Cilium Slack channel.

Step 3: Accessing the VM

After the script has successfully completed, you can log into the VM using `vagrant ssh`:

```
$ vagrant ssh
```

All commands for the rest of the tutorial below should be run from inside this Vagrant VM. If you end up disconnecting from this VM, you can always reconnect in a new terminal window just by running `vagrant ssh` again from the Cilium directory.

Step 4: Confirm that Cilium is Running

The Cilium agent is now running and you can interact with it using the `cilium` CLI client. Check the status of the agent by running `cilium status`:

```
$ cilium status
KVStore:            Ok
ContainerRuntime:   Ok
<...>
Cilium:             Ok
```

The status indicates that all necessary components are operational.

Step 5: Run Script to Start Marathon

Start Marathon inside the Vagrant VM:

```
$ ./start_marathon.sh
Starting marathon...
...
...
...
...
Done
```

Step 6: Simulate a Web-Server and Clients

Use `curl` to submit a task to Marathon for scheduling, with data to run the simulated web-server provided by the `web-server.json`. The web-server simply responds to requests on a particular port.

```
$ curl -i -H 'Content-Type: application/json' -d @web-server.json 127.0.0.1:8080/v2/
↪apps
```

You should see output similar to the following:

```
$ curl -i -H 'Content-Type: application/json' -d @web-server.json 127.0.0.1:8080/v2/
↪apps
HTTP/1.1 201 Created
[...]
Marathon-Deployment-Id: [UUID]
[...]
```

Confirm that Cilium sees the new workload. The output should return the endpoint with label `mesos:id=web-server` and the assigned IP:

```
$ cilium endpoint list
```

And observe similar output to the following:

```
vagrant@vagrant:~$ cilium endpoint list
ENDPOINT   POLICY        IDENTITY   LABELS (source:key[=value])   IPv6                  ␣
↪   IPv4            STATUS
           ENFORCEMENT
29898      Disabled      256        mesos:id=web-server                ␣
↪f00d::a00:20f:0:74ca   10.15.242.54   ready
```

Use the IP from the previous output to test the web-server:

```
$ curl <IP>:8181/api
```

The output should simply return:

```
$ curl <IP>:8181/api
OK
```

Run a script to create "good client" and "bad client" requests to the newly created web-server. The script should generate `goodclient.json` and `badclient.json` files for the client tasks, respectively:

```
$ ./generate_client_file.sh goodclient
$ ./generate_client_file.sh badclient
```

Then submit the client tasks to Marathon, which will generate `GET /public` and `GET /private` requests:

```
$ curl -i -H 'Content-Type: application/json' -d @goodclient.json 127.0.0.1:8080/v2/
↪apps
$ curl -i -H 'Content-Type: application/json' -d @badclient.json 127.0.0.1:8080/v2/
↪apps
```

You can observe the newly created endpoints in Cilium, similar to the following output:

```
$ cilium endpoint list
ENDPOINT   POLICY        IDENTITY   LABELS (source:key[=value])   IPv6                ␣
↪    IPv4            STATUS
           ENFORCEMENT
29898      Disabled      256        mesos:id=web-server          ␣
↪f00d::a00:20f:0:74ca   10.15.242.54   ready
33115      Disabled      257        mesos:id=goodclient          ␣
↪f00d::a00:20f:0:815b   10.15.220.6    ready
64189      Disabled      258        mesos:id=badclient           ␣
↪f00d::a00:20f:0:fabd   10.15.152.27   ready
```

Marathon runs the tasks as batch jobs with `stdout` available under `/var/lib/mesos`. To simplify the retrieval of the `stdout` log, use the `tail_client.sh` script to output each of the client logs. Tail the *goodclient* logs:

```
$ ./tail_client.sh goodclient
```

and in a separate terminal, start another ssh session to the Vagrant VM with `vagrant ssh` and observe the *badclient* logs:

```
$ ./tail_client.sh badclient
```

Make sure the previous command continuously prints the result of the client accessing */public* and */private* API of the web-server:

```
...
---------------------------------
GET /public
OK

HTTP code: 200
GET /private
OK

HTTP code: 200
```

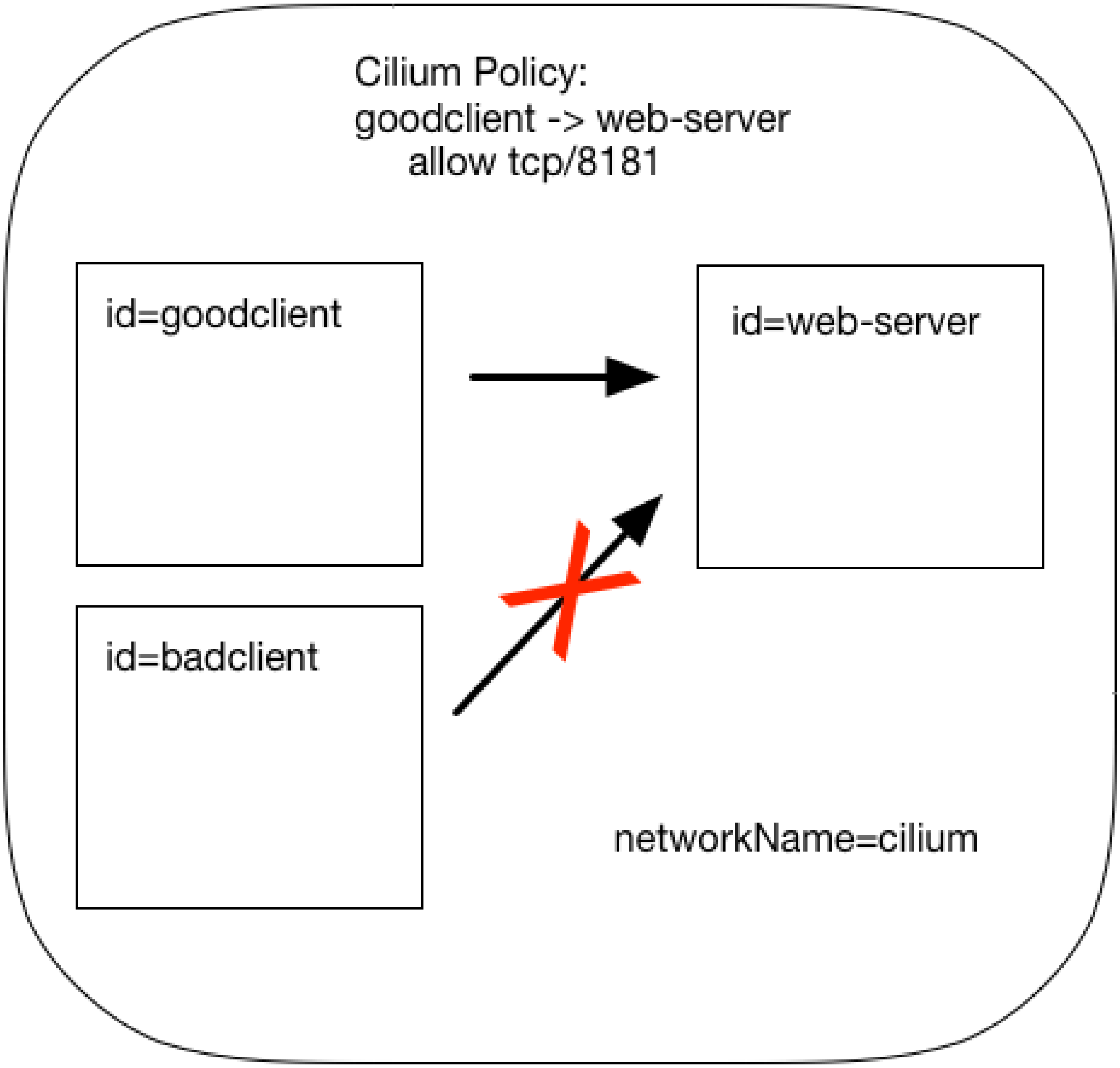
```
--------------------------------
...
```

Step 7: Apply L3/L4 Policy with Cilium

In a new terminal session, do a `vagrant ssh` to get into the Vagrant VM and apply an L3/L4 policy only allowing the *goodclient* to access the *web-server*. The L3/L4 json policy looks like:

```
[{
    "endpointSelector": {"matchLabels":{"id":"web-server"}},
    "ingress": [{
        "fromEndpoints": [
            {"matchLabels":{"id":"goodclient"}}
        ],
        "toPorts": [{
                "ports": [{"port": "8181", "protocol": "tcp"}]
        }]
    }]
}]
```

L3/L4 Policy with Cilium and Mesos



Use `cilium` CLI to apply the L3/L4 policy:

```
$ cilium policy import l3-l4-policy.json
```

You can observe that the policy is applied via `cilium` CLI:

```
$ cilium endpoint list
ENDPOINT   POLICY        IDENTITY   LABELS (source:key[=value])   IPv6                   ␣
↪   IPv4            STATUS
           ENFORCEMENT
29898      Enabled       256        mesos:id=web-server                 ␣
↪f00d::a00:20f:0:74ca   10.15.242.54   ready
33115      Enabled       257        mesos:id=goodclient                 ␣
↪f00d::a00:20f:0:815b   10.15.220.6    ready
64189      Enabled       258        mesos:id=badclient                  ␣
↪f00d::a00:20f:0:fabd   10.15.152.27   ready
```

You should also observe that the *goodclient* logs continue to output the *web-server* responses, whereas the *badclient* request does not reach the *web-server* because of policy enforcement, and logging output similar to below.

```
--------------------------------
GET /public
HTTP code: 000
GET /private
HTTP code: 000
--------------------------------
```

You can remove the L3/L4 policy in order to give *badclient* access to the *web-server* again.

```
$ cilium policy delete --all
Revision: 2
```

The *badclient* logs should resume outputting the *web-server*'s response and Cilium is configured to no longer enforce policy:

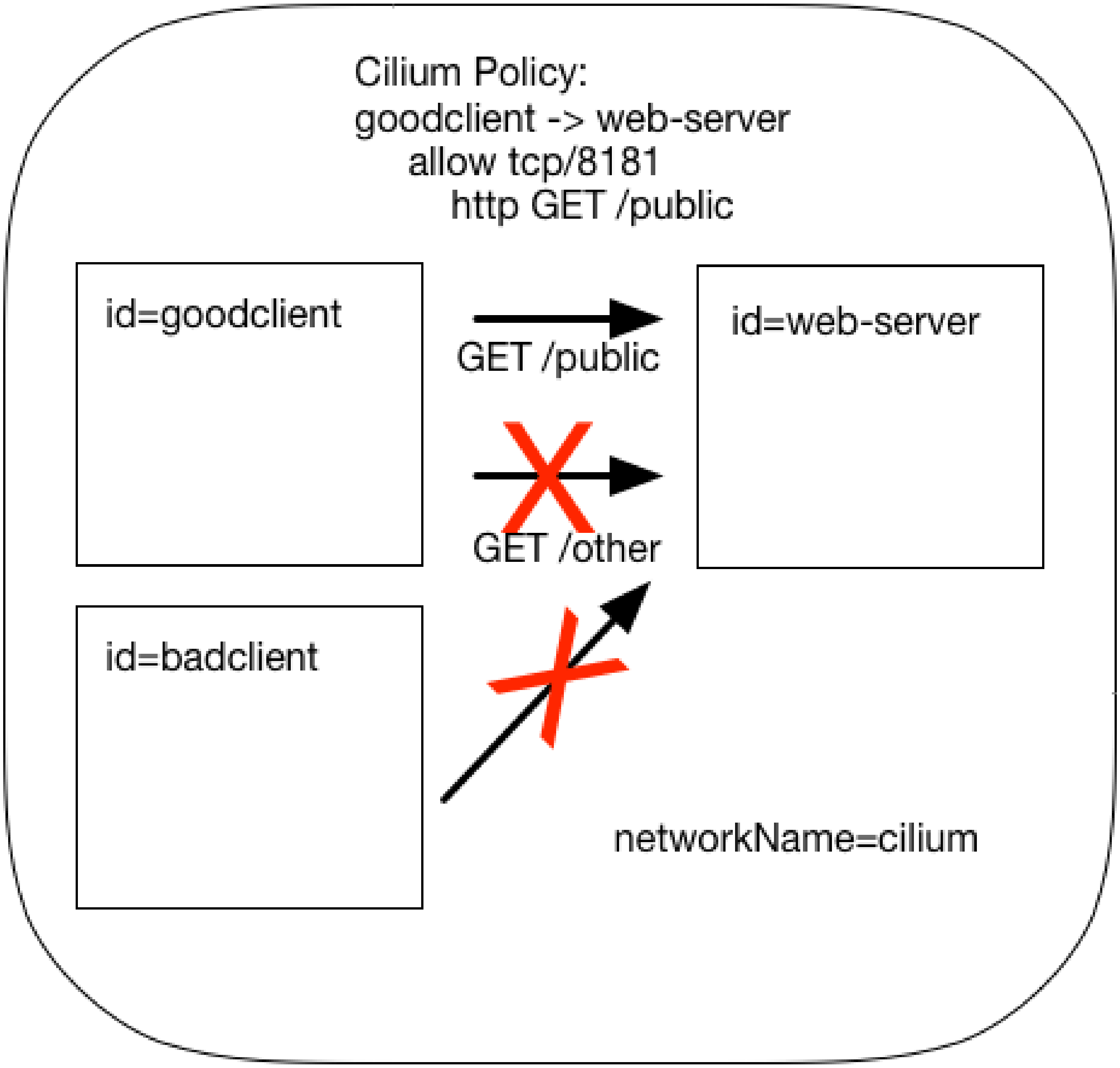
```
$ cilium endpoint list
ENDPOINT   POLICY        IDENTITY   LABELS (source:key[=value])   IPv6                ␣
↪   IPv4           STATUS
           ENFORCEMENT
29898      Disabled      256        mesos:id=web-server                              ␣
↪f00d::a00:20f:0:74ca   10.15.242.54   ready
33115      Disabled      257        mesos:id=goodclient                              ␣
↪f00d::a00:20f:0:815b   10.15.220.6    ready
64189      Disabled      258        mesos:id=badclient                               ␣
↪f00d::a00:20f:0:fabd   10.15.152.27   ready
```

Step 8: Apply L7 Policy with Cilium

Now, in a different terminal, do a `vagrant ssh` and apply the provided policy that allows accessing only the */public* API:

```
$ cilium policy import l7-policy.json
```

L7 Policy with Cilium and Mesos



Check the *goodclient*'s log and you should see that */private* is no longer accessible, and the *badclient*'s requests see the same results as the previous step.

```
--------------------------------
GET /public
OK

HTTP code: 200
GET /private
Access denied
HTTP code: 403
--------------------------------
```

You can also observe that the policy is applied with `cilium` CLI:

```
$ cilium endpoint list
ENDPOINT   POLICY        IDENTITY   LABELS (source:key[=value])   IPv6                              ␣
↪    IPv4            STATUS
```

```
           ENFORCEMENT
29898      Enabled       256        mesos:id=web-server           ␣
↪f00d::a00:20f:0:74ca   10.15.242.54   ready
33115      Enabled       257        mesos:id=goodclient           ␣
↪f00d::a00:20f:0:815b   10.15.220.6    ready
64189      Enabled       258        mesos:id=badclient            ␣
↪f00d::a00:20f:0:fabd   10.15.152.27   ready
```

(optional) Remove the policy and notice that the access to */private* is unrestricted again:

```
$ cilium policy delete --all
```

Step 9: Clean-Up

Exit the vagrant VM by typing `exit`.

When you want to tear-down the Cilium + Mesos VM and destroy all local state (e.g., the VM disk image), open a terminal in the `cilium/examples/mesos` directory and type:

```
$ vagrant destroy
```

You can always re-create the VM using the steps described above.

If instead you just want to shut down the VM but may use it later, `vagrant halt default` will work, and you can start it again later using the contrib/vagrant/start.sh script.

To clean the node kill all mesos-master, mesos-agent, marathon-related processes manually.

Troubleshooting

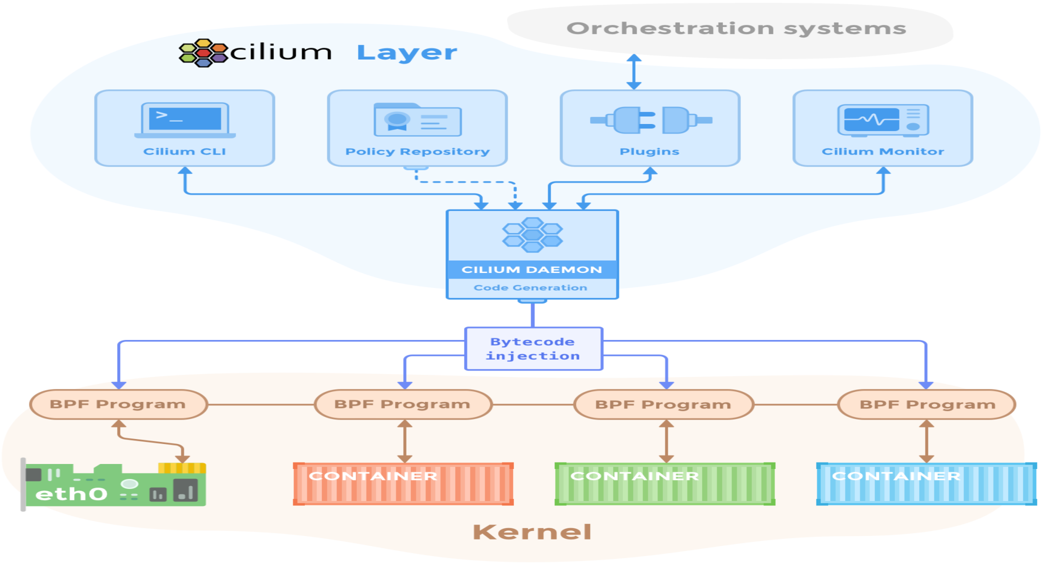
For assistance on any of the Getting Started Guides, please reach out and ask a question on the Cilium Slack channel.

CHAPTER 3

Concepts

The goal of this document is to describe the components of the Cilium architecture, and the different models for deploying Cilium within your datacenter or cloud environment. It focuses on the higher-level understanding required to run a full Cilium deployment. You can then use the more detailed *Troubleshooting* to understand the details of setting up Cilium.

Cilium Components



A deployment of Cilium consists of the following components running on each Linux container node in the container cluster:

- **Cilium Agent (Daemon):** Userspace daemon that interacts with the container runtime and orchestration systems such as Kubernetes via Plugins to setup networking and security for containers running on the local server. Provides an API for configuring network security policies, extracting network visibility data, etc.
- **Cilium CLI Client:** Simple CLI client for communicating with the local Cilium Agent, for example, to configure network security or visibility policies.
- **Linux Kernel BPF:** Integrated capability of the Linux kernel to accept compiled bytecode that is run at various hook / trace points within the kernel. Cilium compiles BPF programs and has the kernel run them at key points in the network stack to have visibility and control over all network traffic in / out of all containers.
- **Container Platform Network Plugin:** Each container platform (e.g., Docker, Kubernetes) has its own plugin model for how external networking platforms integrate. In the case of Docker, each Linux node runs a process (cilium-docker) that handles each Docker libnetwork call and passes data / requests on to the main Cilium Agent.

In addition to the components that run on each Linux container host, Cilium leverages a key-value store to share data between Cilium Agents running on different nodes. The currently supported key-value stores are:

- etcd
- consul
- local storage (golang hashmap)

Cilium Agent

The Cilium agent (cilium-agent) runs on each Linux container host. At a high-level, the agent accepts configuration that describes service-level network security and visibility policies. It then listens to events in the container runtime to learn when containers are started or stopped, and it creates custom BPF programs which the Linux kernel uses to control all network access in / out of those containers. In more detail, the agent:

- Exposes APIs to allow operations / security teams to configure security policies (see below) that control all communication between containers in the cluster. These APIs also expose monitoring capabilities to gain additional visibility into network forwarding and filtering behavior.
- Gathers metadata about each new container that is created. In particular, it queries identity metadata like container / pod labels, which are used to identify endpoints in Cilium security policies.
- Interacts with the container platforms network plugin to perform IP address management (IPAM), which controls what IPv4 and IPv6 addresses are assigned to each container. The IPAM is managed by the agent in a shared pool between all plugins which means that the Docker and CNI network plugin can run side by side allocating a single address pool.
- Combines its knowledge about container identity and addresses with the already configured security and visibility policies to generate highly efficient BPF programs that are tailored to the network forwarding and security behavior appropriate for each container.
- Compiles the BPF programs to bytecode using clang/LLVM and passes them to the Linux kernel to run for all packets in / out of the container's virtual ethernet device(s).

Cilium CLI Client

The Cilium CLI Client (cilium) is a command-line tool that is installed along with the Cilium Agent. It gives a command-line interface to interact with all aspects of the Cilium Agent API. This includes inspecting Cilium's state about each network endpoint (i.e., container), configuring and viewing security policies, and configuring network monitoring behavior.

Linux Kernel BPF

Berkeley Packet Filter (BPF) is a Linux kernel bytecode interpreter originally introduced to filter network packets, e.g. tcpdump and socket filters. It has since been extended with additional data structures such as hashtable and arrays as well as additional actions to support packet mangling, forwarding, encapsulation, etc. An in-kernel verifier ensures that BPF programs are safe to run and a JIT compiler converts the bytecode to CPU architecture specific instructions for native execution efficiency. BPF programs can be run at various hooking points in the kernel such as for incoming packets, outgoing packets, system calls, kprobes, etc.

BPF continues to evolve and gain additional capabilities with each new Linux release. Cilium leverages BPF to perform core datapath filtering, mangling, monitoring and redirection, and requires BPF capabilities that are in any Linux kernel version 4.8.0 or newer. On the basis that 4.8.x is already declared end of life and 4.9.x has been nominated as a stable release we recommend to run at least kernel 4.9.17 (the latest current stable Linux kernel as of this writing is 4.10.x).

Cilium is capable of probing the Linux kernel for available features and will automatically make use of more recent features as they are detected.

Linux distros that focus on being a container runtime (e.g., CoreOS, Fedora Atomic) typically already ship kernels that are newer than 4.8, but even recent versions of general purpose operating systems such as Ubuntu 16.10 ship fairly recent kernels. Some Linux distributions still ship older kernels but many of them allow installing recent kernels from separate kernel package repositories.

For more detail on kernel versions, see: *Linux Kernel*.

Key-Value Store

The Key-Value (KV) Store is used for the following state:

- Policy Identities: list of labels <=> policy identity identifier
- Global Services: global service id to VIP association (optional)
- Encapsulation VTEP mapping (optional)

To simplify things in a larger deployment, the key-value store can be the same one used by the container orchestrater (e.g., Kubernetes using etcd). In single node Cilium deployments used for basic testing / learning, Cilium can use a local store implemented as a golang hash map, avoiding the need to setup a dedicated KV store.

Labels

Labels are a generic, flexible and highly scaleable way of addressing a large set of resources as they allow for arbitrary grouping and creation of sets. Whenever something needs to be described, addressed or selected this is done based on labels:

- Endpoints are assigned labels as derived from container runtime or the orchestration system.
- Network policies select endpoints based on labels and allow consumers based on labels.
- Network policies themselves are described and addressed by labels.

Basic Label: Key/Value Pair

A label is a pair of strings consisting of a `key` and `value`. A label can be formatted as a single string with the format `key=value`. The key portion is mandatory and must be unique. This is typically achieved by using the reverse domain name notion, e.g. `io.cilium.mykey=myvalue`. The value portion is optional and can be omitted, e.g. `io.cilium.mykey`.

Key names should typically consist of the character set `[a-z0-9-.]`.

When using labels to select resources, both the key and the value must match, e.g. when a policy should be applied to all endpoints with the label `my.corp.foo` then the label `my.corp.foo=bar` will not match the selector.

Label Source

A label can be derived from various sources. For example, a Cilium endpoint will derive the labels associated to the container by the local container runtime as well as the labels associated with the pod as provided by Kubernetes. As these two label namespaces are not aware of each other, this may result in conflicting label keys.

To resolve this potential conflict, Cilium prefixes all label keys with `source:` to indicate the source of the label when importing labels, e.g. `k8s:role=frontend`, `container:user=joe`, `k8s:role=backend`. This means that when you run a Docker container using `docker run [...] -l foo=bar`, the label `container:foo=bar` will appear on the Cilium endpoint representing the container. Similiarly, a Kubernetes pod started with the label `foo: bar` will be represented with a Cilium endpoint associated with the label `k8s:foo=bar`. A unique name is allocated for each potential source. The following label sources are currently supported:

- `container:` for labels derived from the local container runtime
- `k8s:` for labels derived from Kubernetes
- `reserved:` for special reserved labels, see *Special Identities*.
- `unspec:` for labels with unspecified source

When using labels to identify other resources, the source can be included to limit matching of labels to a particular type. If no source is provided, the label source defaults to `any:` which will match all labels regardless of their source. If a source is provided, the source of the selecting and matching labels need to match.

Endpoints

Cilium makes application containers available on the network by assigning them IP addresses. Multiple application containers can share the same IP address; a typical example for this model is a Kubernetes Pod. All application containers which share a common address are grouped together in what Cilium refers to as an *endpoint*.

Allocating individual IP addresses enables the use of the entire Layer 4 port range by each *endpoint*. This essentially allows multiple application containers running on the same cluster node to all bind to well known ports such *80* without causing any conflicts.

The default behaviour of Cilium is to assign both an IPv6 and IPv4 address to every *endpoint*. However, this behaviour can be configured to only allocate an IPv6 address with the `--disable-ipv4` option. If both an IPv6 and IPv4 address are assigned, either address can be used to reach the endpoint. The same behaviour will apply with regard to policy rules, load-balancing, etc. See *address management* for more details.

Identification

For identification purposes, Cilium assigns an *endpoint id* to all endpoints on a cluster node. The *endpoint id* is unique within the context of an individual cluster node.

Endpoint Metadata (Labels)

An *endpoint* automatically derives metadata from the application containers associated with the *endpoint*. The metadata can then be used to identify the endpoint for security/policy, load-balancing and routing purposes.

The source of the metadata will depend on the orchestration system and container runtime in use. The following metadata retrieval mechanisms are currently supported:

System	Description
Kubernetes	Pod labels (via k8s API)
Mesos	Labels (via CNI)
containerd (Docker)	Container labels (via Docker API)

Metadata is attached to *endpoints* in the form of *Labels*.

The following example launches a container with the label `app=benchmark` which is then associated with the *endpoint*. The label is prefixed with `container:` to indicate that the label was derived from the container runtime.

```
$ docker run --net cilium -d -l app=benchmark tgraf/netperf
aaff7190f47d071325e7af06577f672beff64ccc91d2b53c42262635c063cf1c
$  cilium endpoint list
ENDPOINT   POLICY        IDENTITY   LABELS (source:key[=value])   IPv6                 ␣
↪   IPv4            STATUS
           ENFORCEMENT
62006      Disabled      257        container:app=benchmark        ␣
↪f00d::a00:20f:0:f236   10.15.116.202   ready
```

An endpoint can have metadata associated from multiple sources. A typical example is a Kubernetes cluster which uses containerd as the container runtime. Endpoints will derive Kubernetes pod labels (prefixed with the *k8s:* source prefix) and containerd labels (prefixed with *container:* source prefix).

Cluster Node

Cilium refers to a node as an individual member of a cluster. Each node must be running the `cilium-agent` and will operate in a mostly autonomous manner. Synchronization of state between Cilium agent's running on different nodes is kept to a minimum for simplicity and scale. It occurs exclusively via the Key-Value store or with packet metadata.

Node Address

Cilium will automatically detect the node's IPv4 and IPv6 address. The detected node address is printed out when the `cilium-agent` starts:

```
Local node-name: worker0
Node-IPv6: f00d::ac10:14:0:1
External-Node IPv4: 172.16.0.20
Internal-Node IPv4: 10.200.28.238
```

Address Management

The address management is designed with simplicity and resilience in mind. This is achieved by delegating the address allocation for *endpoints* to each individual node in the cluster. Each cluster node is assigned a *node address allocation prefix* out of an overarching *cluster address prefix* and will allocate IPs for *endpoints* independently.

This simplifies address handling and allows one to make a fundamental assumption:

- No state needs to be synchronized between cluster nodes to allocate IP addresses and to determine whether an IP address belongs to an *endpoint* of the cluster and whether that *endpoint* resides on the local cluster node.

Note: If you are using Kubernetes, the allocation of the node address prefix can be simply delegated to Kubernetes by specifying `--allocate-node-cidrs` flag to `kube-controller-manager`. Cilium will automatically use the IPv4 node CIDR allocated by Kubernetes.

The following values are used by default if the cluster prefix is left unspecified. These are meant for testing and need to be adjusted according to the needs of your environment.

Type	Cluster	Node Prefix
IPv4	`10.0.0.0/8`	`10.X.0.0/16` where `X` is derived using the last 8 bits of the first IPv4 address in the list of global scope addresses on the cluster node.
IPv6	`f00d::/48`	`f00d:0:0:0:<ipv4-address>::/96` where the IPv4 address is the first address in the list of global scope addresses on the cluster node. Note: Only 16 bits out of the `/96` node prefix are currently used when allocating container addresses. This allows to use the remaining 16 bits to store arbitrary connection state when sending packets between nodes. A typical use case for the state is direct server return.

The size of the IPv4 cluster prefix can be changed with the `--ipv4-cluster-cidr-mask-size` option. The size of the IPv6 cluster prefix is currently fixed sized at `/48`. The node allocation prefixes can be specified manually with the option `--ipv4-range` respectively `--ipv6-range`.

Multi Host Networking

Cilium is in full control over both ends of the connection for connections inside the cluster. It can thus transmit state and security context information between two container hosts by embedding the information in encapsulation headers or even unused bits of the IPv6 packet header. This allows Cilium to transmit the security context of where the packet originates, which allows tracing back which container labels are assigned to the origin container.

Note: As the packet headers contain security sensitive information, it is higly recommended to either encrypt all traffic or run Cilium in a trusted network environment.

Cilium keeps the networking concept as simple as possible. There are two networking models to choose from.

- *Overlay Network Mode*
- *Direct / Native Routing Mode*

Regardless of the option chosen, the container itself has no awareness of the underlying network it runs on; it only contains a default route which points to the IP address of the cluster node. Given the removal of the routing cache in the Linux kernel, this reduces the amount of state to keep in the per connection flow cache (TCP metrics), which allows to terminate millions of connections in each container.

Overlay Network Mode

When no configuration is provided, Cilium automatically runs in this mode.

In this mode, all cluster nodes form a mesh of tunnels using the UDP based encapsulation protocols VXLAN or Geneve. All container-to-container network traffic is routed through these tunnels. This mode has several major advantages:

- **Simplicity:** The network which connects the cluster nodes does not need to be made aware of the *cluster prefix*. Cluster nodes can spawn multiple routing or link-layer domains. The topology of the underlying network is irrelevant as long as cluster nodes can reach each other using IP/UDP.
- **Auto-configuration:** When running together with an orchestration system such as Kubernetes, the list of all nodes in the cluster including their associated allocation prefix node is made available to each agent automatically. This means that if Kubernetes is being run with the *–allocate-node-cidrs* option, Cilium can form an overlay network automatically without any configuration by the user. New nodes joining the cluster will automatically be incorporated into the mesh.
- **Identity transfer:** Encapsulation protocols allow for the carrying of arbitrary metadata along of with the network packet. Cilium makes use of this ability to transfer metadata such as the source security identity and load balancing state to perform direct-server-return.

Direct / Native Routing Mode

Note: This is an advanced networking mode which requires the underlying network to be made aware of container IPs. You can enable this mode by running Cilium with the option `--tunnel disabled`.

In direct routing mode, Cilium will hand all packets which are not addressed for another local endpoint to the routing subsystem of the Linux kernel. This means that the packet will be routed as if a local process would have emitted the packet. As a result, the network connecting the cluster nodes must be aware that each of the node IP prefixes are reachable by using the node's primary IP address as an L3 next hop address.

Cilium automatically enables IP forwarding in Linux when direct mode is configured, but it is up to the container cluster administrator to ensure that each routing element in the underlying network has a route that describes each node IP as the IP next hop for the corresponding node prefix.

This is typically achieved using two methods:

- Operation of a routing protocol such as OSPF or BPG via routing daemon such as Zebra, bird, bpgd. The routing protocols will announce the *node allocation prefix* via the node's IP to all other nodes.
- Use of the cloud provider's routing functionality. Refer to the documentation of your cloud provider for additional details (e.g,. AWS VPC Route Tables or GCE Routes). These APIs can be used to associate each node prefix with the appropriate next hop IP each time a container node is added to the cluster. If you are running Kubernetes with the *–cloud-provider* in combination with the *–allocate-node-cidrs* option then this is configured automatically for IPv4 prefixes.

Note: Use of direct routing mode currently only offers identity based security policy enforcement for IPv6 where the security identity is stored in the flowlabel. IPv4 is currently not supported and thus security must be enforced using CIDR policy rules.

There are two possible approaches to performing network forwarding for container-to-container traffic:

Container Communication with External Hosts

Container communication with the outside world has two primary modes:

- Containers exposing API services for consumption by hosts outside of the container cluster.
- Containers making outgoing connections. Examples include connecting to 3rd-party API services like Twillio or Stripe as well as accessing private APIs that are hosted elsewhere in your enterprise datacenter or cloud deployment.

In the *Direct / Native Routing Mode* mode described before, if container IP addresses are routable outside of the container cluster, communication with external hosts requires little more than enabling L3 forwarding on each of the Linux nodes.

External Network Connectivity

If the destination of a packet lies outside of the cluster, Cilium will delegate routing to the routing subsystem of the cluster node to use the default route which is installed on the node of the cluster.

As the IP addresses used for the **cluster prefix** are typically allocated from RFC1918 private address blocks and are not publicly routable. Cilium will automatically masquerade the source IP address of all traffic that is leaving the cluster. This behaviour can be disabled by running `cililum-agent` with the option `--masquerade=false`.

Public Endpoint Exposure

In direct routing mode, *endpoint* IPs can be publicly routable IPs and no additional action needs to be taken.

In overlay mode, *endpoints* that are accepting inbound connections from cluster external clients likely want to be exposed via some kind of load-balancing layer. Such a load-balancer will have a public external address that is not part of the Cilium network. This can be achieved by having a load-balancer container that both has a public IP on an externally reachable network and a private IP on a Cilium network. However, many container orchestration frameworks, like Kubernetes, have built in abstractions to handle this “ingress” load-balancing capability, which achieve the same effect that Cilium handles forwarding and security only for ‘’internal” traffic between different services.

Security

Cilium provides security on multiple levels. Each can be used individually or combined together.

- *Identity based Connectivity Access Control*: Connectivity policies between endpoints (Layer 3), e.g. any endpoint with label *role=frontend* can connect to any endpoint with label *role=backend*.
- Restriction of accessible ports (Layer 4) for both incoming and outgoing connections, e.g. endpoint with label *role=frontend* can only make outgoing connections on port 443 (https) and endpoint *role=backend* can only accept connections on port 443 (https).
- Fine grained access control on application protocol level to secure HTTP and remote procedure call (RPC) protocols, e.g the endpoint with label *role=frontend* can only perform the REST API call *GET /userdata/[0-9]+*, all other API interactions with *role=backend* are restricted.

Currently on the roadmap, to be added soon:

- Authentication: Any endpoint which wants to initiate a connection to an endpoint with the label *role=backend* must have a particular security certificate to authenticate itself before being able to initiate any connections. See GH issue 502 for additional details.
- Encryption: Communication between any endpoint with the label *role=frontend* to any endpoint with the label *role=backend* is automatically encrypted with a key that is automatically rotated. See GH issue 504 to track progress on this feature.

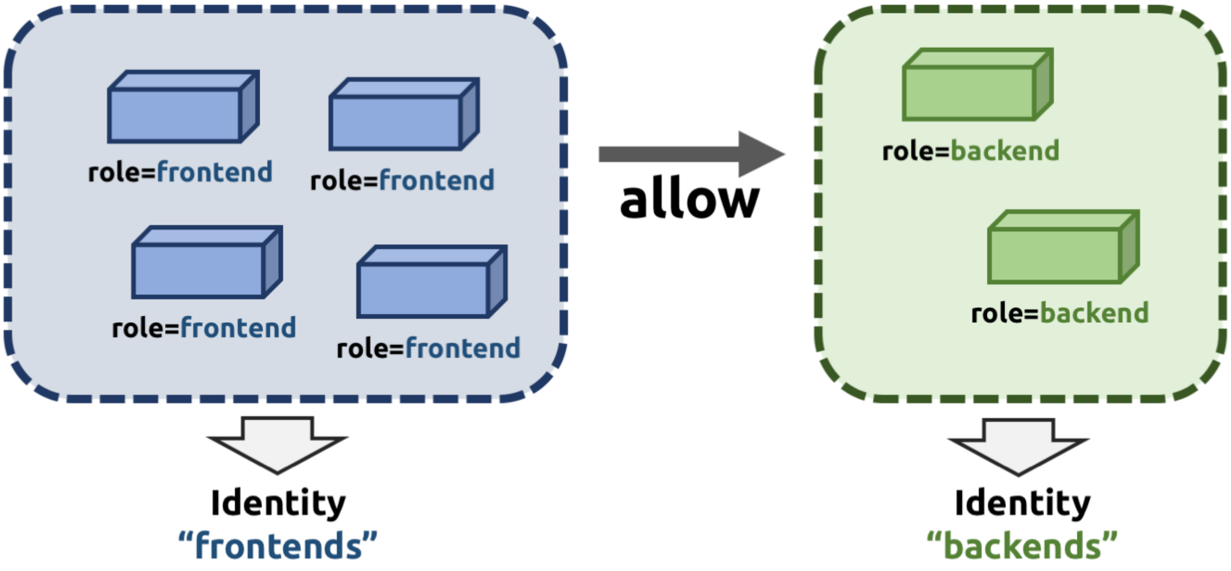
Identity based Connectivity Access Control

Container management systems such as Kubernetes deploy a networking model which assigns an individual IP address to each pod (group of containers). This ensures simplicity in architecture, avoids unnecessary network address translation (NAT) and provides each individual container with a full range of port numbers to use. The logical consequence of this model is that depending on the size of the cluster and total number of pods, the networking layer has to manage a large number of IP addresses.

Traditionally security enforcement architectures have been based on IP address filters. Let's walk through a simple example: If all pods with the label *role=frontend* should be allowed to initiate connections to all pods with the label *role=backend* then each cluster node which runs at least one pod with the label *role=backend* must have a corresponding filter installed which allows all IP addresses of all *role=frontend* pods to initiate a connection to the IP addresses of all local *role=backend* pods. All other connection requests should be denied. This could look like this: If the destination address is *10.1.1.2* then allow the connection only if the source address is one of the following *[10.1.2.2,10.1.2.3,20.4.9.1]*.

Every time a new pod with the label *role=frontend* or *role=backend* is either started or stopped, the rules on every cluster node which run any such pods must be updated by either adding or removing the corresponding IP address from the list of allowed IP addresses. In large distributed applications, this could imply updating thousands of cluster nodes multiple times per second depending on the churn rate of deployed pods. Worse, the starting of new *role=frontend* pods must be delayed until all servers running *role=backend* pods have been updated with the new security rules as otherwise connection attempts from the new pod could be mistakenly dropped. This makes it difficult to scale efficiently.

In order to avoid these complications which can limit scalability and flexibility, Cilium entirely separates security from network addressing. Instead, security is based on the identity of a pod, which is derived through labels. This identity can be shared between pods. This means that when the first *role=frontend* pod is started, Cilium assigns an identity to that pod which is then allowed to initiate connections to the identity of the *role=backend* pod. The subsequent start of additional *role=frontend* pods only requires to resolve this identity via a key-value store, no action has to be performed on any of the cluster nodes hosting *role=backend* pods. The starting of a new pod must only be delayed until the identity of the pod has been resolved which is a much simpler operation than updating the security rules on all other cluster nodes.



What is an Endpoint Identity?

The identity of an endpoint is derived based on the labels associated with the pod or container. When a pod or container is started, Cilium will create an endpoint based on the event received by the container runtime to represent the pod or container on the network. As a next step, Cilium will resolve the identity of the endpoint created. Whenever the labels of the pod or container change, the identity is reconfirmed and automatically modified as required.

Not all labels associated with a container or pod are meaningful when deriving the security identity. Labels may be used to store metadata such as the timestamp when a container was launched. Cilium requires to know which labels are meaningful and are subject to being considered when deriving the identity. For this purpose, the user is required to specify a list of string prefixes of meaningful labels. The standard behavior is to include all labels which start with the prefix *id.*, e.g. *id.service1*, *id.service2*, *id.groupA.service44*. The list of meaningful label prefixes can be specified when starting the cilium agent, see *Command Line Options*.

Special Identities

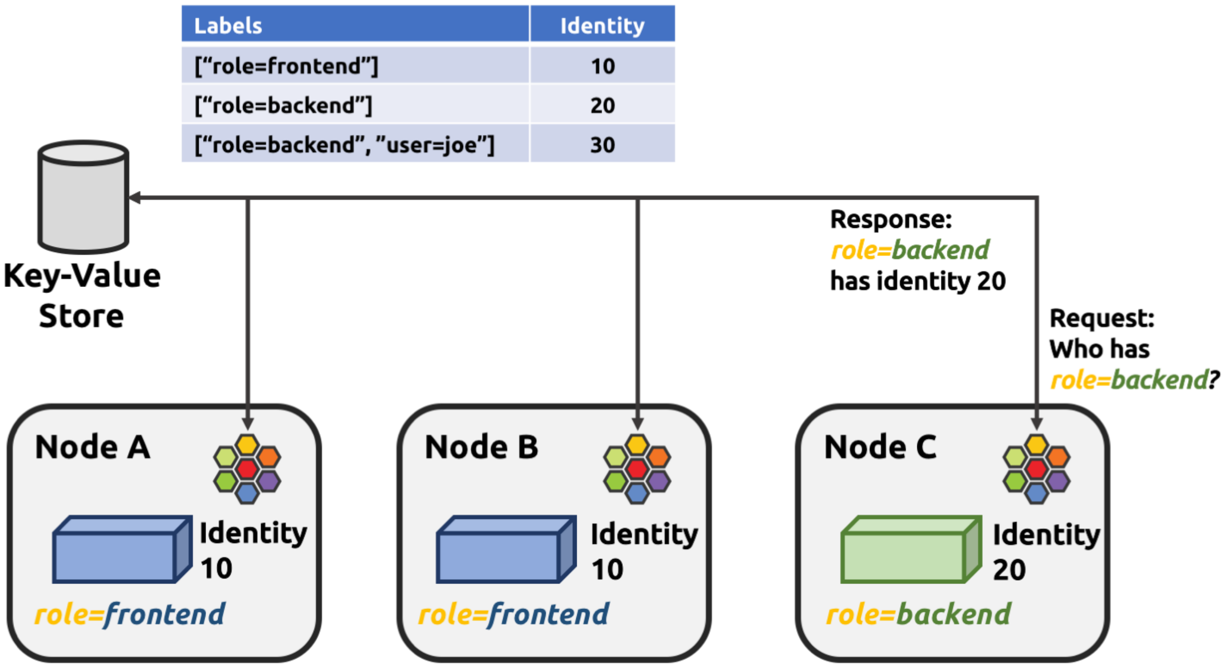
All endpoints which are managed by Cilium will be assigned an identity. In order to allow communication to network endpoints which are not managed by Cilium, special identities exist to represent those. Special reserved identities are prefixed with the string *reserved:*.

Identity	Description
reserved:host	The host network namespace on which the pod or container is running.
reserved:world	Any network endpoint outside of the cluster

TODO: Document *cidr:* identity once implemented.

Identity Management in the Cluster

Identities are valid in the entire cluster which means that if several pods or containers are started on several cluster nodes, all of them will resolve and share a single identity if they share the identity relevant labels. This requires coordination between cluster nodes.



Labels	Identity
["role=frontend"]	10
["role=backend"]	20
["role=backend", "user=joe"]	30

The operation to resolve an endpoint identity is performed with the help of the distributed key-value store which allows to perform atomic operations in the form *generate a new unique identifier if the following value has not been seen before*. This allows each cluster node to create the identity relevant subset of labels and then query the key-value store to derive the identity. Depending on whether the set of labels has been queried before, either a new identity will be created, or the identity of the initial query will be returned.

Policy Enforcement

All security policies are described assuming stateful policy enforcement for session based protocols. This means that the intent of the policy is to describe allowed direction of connection establishment. If the policy allows *A* => *B* then reply packets from *B* to *A* are automatically allowed as well. However, *B* is not automatically allowed to initiate connections to *A*. If that outcome is desired, then both directions must be explicitly allowed.

Security policies are primarily enforced at *ingress* which means that each cluster node verifies all incoming packets and determines whether the packet is allowed to be transmitted to the intended endpoint. Policy enforcement also occurs at *egress* if required by the specific policy, e.g. a Layer 7 policy restricting outgoing API calls.

Layer 3 policies are currently not enforced at *egress* to avoid the complexity of resolving the destination endpoint identity before sending out the packet. Instead, the identity of the source endpoint is embedded into the packet.

In order to enforce identity based security in a multi host cluster, the identity of the transmitting endpoint is embedded into every network packet that is transmitted in between cluster nodes. The receiving cluster node can then extract the identity and verify whether a particular identity is allowed to communicate with any of the local endpoints.

Default Security Policy

If no policy is loaded, the default behaviour is to allow all communication unless policy enforcement has been explicitly enabled. As soon as the first policy rule is loaded, policy enforcement is enabled automatically and any communication must then be white listed or the relevant packets will be dropped.

Similarly, if an endpoint is not subject to an *L4* policy, communication from and to all ports is permitted. Associating at least one *L4* policy to an endpoint will block all connectivity to ports unless explicitly allowed.

Orchestration System Specifics

Kubernetes

Cilium regards each deployed Pod as an endpoint with regards to networking and security policy enforcement. Labels associated with pods can be used to define the identity of the endpoint.

When two pods communicate via a service construct, then the labels of the origin pod apply to determine the identity.

Integrations

Cilium is deeply integrated with container platforms like Docker or Kubernetes. This enables Cilium to perform network forwarding and security using a model that maps directly to notions of identity (e.g., labels) and service abstractions that are native to the container platform.

In this section, we will provide more detail on how Cilium integrates with Docker and Kubernetes.

Kubernetes

When deployed with Kubernetes, Cilium provides four core Kubernetes networking capabilities:

- Direct pod-to-pod network inter-connectivity.
- Service-based load-balancing for pod-to-pod inter-connectivity (i.e., a kube-proxy replacement).
- Identity-based security policies for all (direct and service-based) Pod-to-Pod inter-connectivity.
- External-to-Pod service-based load-balancing (referred to as *Ingress* in Kubernetes)

The Kubernetes documentation contains more background on the Kubernetes Networking Model and Kubernetes Network Plugins .

Pod-to-Pod Connectivity

In Kubernetes, containers are deployed within units referred to as Pod, which include one or more containers reachable via a single IP address. With Cilium, each Pod gets an IP address from the node prefix of the Linux node running the Pod. See *address management* for additional details. In the absence of any network security policies, all Pods can reach each other.

Pod IP addresses are typically local to the Kubernetes cluster. If pods need to reach services outside the cluster as a client, the network traffic is automatically masqueraded as it leaves the node. You can find additional information in the section *External Network Connectivity*.

Service Load-balancing

Kubernetes has developed the Services abstraction which provides the user the ability to load balance network traffic to different pods. This abstraction allows the pods reaching out to other pods by a single IP address, a virtual IP address, without knowing all the pods that are running that particular service.

Without Cilium, kube-proxy is installed on every node, watches for endpoints and services addition and removal on the kube-master which allows it to to apply the necessary enforcement on iptables. Thus, the received and sent traffic from and to the pods are properly routed to the node and port serving for that service. For more information you can check out the kubernetes user guide for Services.

Cilium load-balancer acts on the same principles as kube-proxy, it watches for services addition or removal, but instead of doing the enforcement on the iptables, it updates BPF map entries on each node. For more information, see the Pull Request.

Docker

Docker supports network plugins via the libnetwork plugin interface .

When using Cilium with Docker, one creates a single logical Docker network of type *cilium* and with an IPAM-driver of type *cilium*, which delegates control over IP address management and network connectivity to Cilium for all containers attached to this network for both IPv4 and IPv6 connectivity. Each Docker container gets an IP address from the node prefix of the node running the container.

When deployed with Docker, each Linux node runs a *cilium-docker* agent, which receives libnetwork calls from Docker and then communicates with the Cilium Agent to control container networking.

Security policies controlling connectivity between the Docker containers can be written in terms of the Docker container labels passed to Docker while creating the container. These policies can be created/updated via communication directly with the Cilium agent, either via API or by using the Cilium CLI client.

CHAPTER 4

Installation Guide

This document describes how to install and configure Cilium in different deployment modes. It focuses on a full deployment of Cilium within a datacenter or public cloud. If you are just looking for a simple way to experiment, we highly recommend trying out the *Getting Started* instead.

This guide assumes that you have read the *Concepts* which explains all the components and concepts.

System Requirements

Before installing Cilium. Please ensure that your system is meeting the minimal requirements to run Cilium. Most modern Linux distributions will automatically meet the requirements.

Summary

When running Cilium using the container image `cilium/cilium`, these are the requirements your system has to fulfill:

- Linux kernel >= 4.8 (>= 4.9.17 LTS recommended)
- Key-Value store (see *Key-Value store* section for version details)

The following additional dependencies are **only** required if you choose **not** to use the `cilium/cilium` container image and want to run Cilium as a native process on your host:

- *clang+LLVM* >=3.7.1
- iproute2 >= 4.8.0

Linux Distribution Compatibility Matrix

The following table lists Linux distributions versions which are known to work well with Cilium.

Distribution	Minimal Version
CoreOS	stable (>= 1298.5.0)
Debian	>= 9 Stretch
Fedora Atomic/Core	>= 25
LinuxKit	all
Ubuntu	>= 16.04.2, >= 16.10

Note: The above list is composed based on feedback by users, if you have good experience with a particular Linux distribution which is not listed below, please let us know by opening a GitHub issue or by creating a pull request to update this guide.

Linux Kernel

Cilium leverages and builds on the kernel functionality BPF as well as various subsystems which integrate with BPF. Therefore, all systems that will run a Cilium agent are required to run the Linux kernel version 4.8.0 or later.

The 4.8.0 kernel is the minimal kernel version required, more recent kernels may provide additional BPF functionality. Cilium will automatically detect additional available functionality by probing for the functionality when the agent starts.

In order for the BPF feature to be enabled properly, the following kernel configuration options must be enabled. This is typically the case automatically with distribution kernels. If an option provides the choice to build as module or statically linked, then both choices are valid.

```
CONFIG_BPF=y
CONFIG_BPF_SYSCALL=y
CONFIG_NET_CLS_BPF=y
CONFIG_BPF_JIT=y
CONFIG_NET_CLS_ACT=y
CONFIG_NET_SCH_INGRESS=y
CONFIG_CRYPTO_SHA1=y
CONFIG_CRYPTO_USER_API_HASH=y
```

Key-Value store

Cilium uses a distributed Key-Value store to manage and distribute security identities across all cluster nodes. The following Key-Value stores are currently supported:

- etcd >= 3.1.0
- consul >= 0.6.4

See section *Key-Value Store* for details on how to configure the *cilium-agent* to use a Key-Value store.

clang+LLVM

Note: This requirement is only needed if you run `cilium-agent` natively. If you are using the Cilium container image `cilium/cilium`, clang+LLVM is included in the container image.

LLVM is the compiler suite which Cilium uses to generate BPF bytecode before loading the programs into the Linux kernel. The minimal version of LLVM installed on the system is >=3.7.1. The version of clang installed must be compiled with the BPF backend enabled.

See http://releases.llvm.org/ for information on how to download and install LLVM. Be aware that in order to use clang 3.9.x, the kernel version requirement is >= 4.9.17.

iproute2

Note: This requirement is only needed if you run `cilium-agent` natively. If you are using the Cilium container image `cilium/cilium`, iproute2 is included in the container image.

iproute2 is a low level tool used to configure various networking related subsystems of the Linux kernel. Cilium uses iproute2 to configure networking and `tc` which is part of iproute2 to load BPF programs into the kernel.

The minimal version of iproute2 installed must be >= 4.8.0. Please see https://www.kernel.org/pub/linux/utils/net/iproute2/ for documentation on how to install iproute2.

Kubernetes Installation Guide

This section describes how to install and run Cilium on Kubernetes. The deployment method we are using is called DaemonSet which is the easiest way to deploy Cilium in a Kubernetes environment. It will request Kubernetes to automatically deploy and run a `cilium/cilium` container image as a pod on all Kubernetes worker nodes.

Should you encounter any issues during the installation, please refer to the *Kubernetes* section and / or seek help on Slack channel.

TL;DR Version (Expert Mode)

If you know what you are doing, then the following quick instructions get you started in the shortest time possible. If you require additional details or are looking to customize the installation then read the remaining sections of this chapter.

1. Mount the BPF filesystem on all k8s worker nodes. There are many ways to achieve this, see section *Step by Step Instructions* for more details.

```
mount bpffs /sys/fs/bpf -t bpf
```

2. Download the DaemonSet template `cilium-ds.yaml` and specify the k8s API server and Key-Value store addresses:

```
$ wget https://raw.githubusercontent.com/cilium/cilium/master/examples/kubernetes/
↪cilium-ds.yaml
$ vim cilium-ds.yaml
[adjust --k8s-api-server or --k8s-kubeconfig-path]
[adjust --kvstore and --kvstore-opts]
```

3. Deploy the `cilium` DaemonSet

```
$ kubectl create -f cilium-ds.yaml
daemonset "cilium" created

$ kubectl get ds --namespace kube-system
NAME             DESIRED   CURRENT   READY     NODE-SELECTOR   AGE
cilium           1         1         1         <none>          2m
```

Step by Step Instructions

Mounting the BPF FS

This step is optional but recommended. It allows the `cilium-agent` to pin BPF resources to a persistent filesystem and make them persistent across restarts of the agent. If the BPF filesystem is not mounted in the host filesystem, Cilium will automatically mount the filesystem in the mount namespace of the container when the agent starts. This will allow operation of Cilium but will result in unmounting of the filesystem when the pod is restarted. This in turn will cause resources such as the connection tracking table of the BPF programs to be released which will cause all connections into local containers to be dropped. Mounting the BPF filesystem in the host mount namespace will ensure that the agent can be restarted without affecting connectivity of any pods.

In order to mount the BPF filesystem, the following command must be run in the host mount namespace. The command must only be run once during the boot process of the machine.

```
mount bpffs /sys/fs/bpf -t bpf
```

A portable way to achieve this with persistence is to add the following line to */etc/fstab* and then run *mount /sys/fs/bpf*. This will cause the filesystem to be automatically mounted when the node boots.

```
bpffs                      /sys/fs/bpf             bpf     defaults 0 0
```

If you are using systemd to manage the kubelet, another option is to add a `ExecStartPre` line in the `/etc/systemd/kubelet.service` file as follows:

```
[Service]
ExecStartPre=/bin/bash -c ' \\
        if [[ \$(/bin/mount | /bin/grep /sys/fs/bpf -c) -eq 0 ]]; then \\
           /bin/mount bpffs /sys/fs/bpf -t bpf; \\
        fi'
```

CNI Configuation

CNI - Container Network Interface is the plugin layer used by Kubernetes to delegate networking configuration. You can find additional information on the CNI project website.

Note: Kubernetes ‘‘ >= 1.3.5‘‘ requires the `loopback` CNI plugin to be installed on all worker nodes. The binary is typically provided by most Kubernetes distributions. See section *Installing CNI and loopback* for instructions on how to install CNI in case the `loopback` binary is not already installed on your worker nodes.

CNI configuration is automatically being taken care of when deploying Cilium via the provided DaemonSet. The script `cni-install.sh` is automatically run via the `postStart` mechanism when the `cilium` pod is started.

Note: In order for the the `cni-install.sh` script to work properly, the `kubelet` task must either be running on the host filesystem of the worder node, or the `/etc/cni/net.d` and `/opt/cni/bin` directories must be mounted into the container where `kubelet` is running. This can be achieved with Volumes mounts.

The CNI auto installation is performed as follows:

1. The `/etc/cni/net.d` and `/opt/cni/bin` directories are mounted from the host filesystem into the pod where Cilium is running.
2. The file `/etc/cni/net.d/10-cilium.conf` is written in case it does not exist yet.

3. The binary `cilium-cni` is installed to `/opt/cni/bin`. Any existing binary with the name `cilium-cni` is overwritten.

Installing CNI and loopback

Since Kubernetes `v1.3.5` the `loopback` CNI plugin must be installed. There are many ways to install CNI, the following is an example:

```
sudo mkdir -p /opt/cni
wget https://storage.googleapis.com/kubernetes-release/network-plugins/cni-
↪0799f5732f2a11b329d9e3d51b9c8f2e3759f2ff.tar.gz
sudo tar -xvf cni-0799f5732f2a11b329d9e3d51b9c8f2e3759f2ff.tar.gz -C /opt/cni
rm cni-0799f5732f2a11b329d9e3d51b9c8f2e3759f2ff.tar.gz
```

Adjusting CNI configuration

If you want to adjust the CNI configuration you may do so by creating the CNI configuration `/etc/cni/net.d/10-cilium.conf` manually:

```
sudo mkdir -p /etc/cni/net.d
sudo sh -c 'echo "{
    "name": "cilium",
    "type": "cilium-cni",
    "mtu": 1450
}
" > /etc/cni/net.d/10-cilium.conf'
```

Cilium will use any existing `/etc/cni/net.d/10-cilium.conf` file if it already exists on a worker node and only creates it if it does not exist yet.

RBAC integration

If you have RBAC enabled in your Kubernetes cluster, create appropriate cluster roles and service accounts for Cilium:

```
$ kubectl create -f https://raw.githubusercontent.com/cilium/cilium/master/examples/
↪kubernetes/rbac.yaml
clusterrole "cilium" created
serviceaccount "cilium" created
clusterrolebinding "cilium" created
```

Configuring the DaemonSet

```
$ wget https://raw.githubusercontent.com/cilium/cilium/master/examples/kubernetes/
↪cilium-ds.yaml
$ vim cilium-ds.yaml
```

The following configuration options *must* be specified:

- `--k8s-api-server` or `--k8s-kubeconfig-path` must point to at least one Kubernetes API server address.
- `--kvstore` with optional `--kvstore-opts` to configure the Key-Value store. See section *Key-Value Store* for additional details on how to configure the Key-Value store.

Deploying the DaemonSet

After configuring the `cilium` DaemonSet it is time to deploy it using `kubectl`:

```
$ kubectl create -f cilium-ds.yaml
```

Kubernetes will deploy the `cilium` DaemonSet as a pod in the `kube-system` namespace on all worker nodes. This operation is performed in the background. Run the following command to check the progress of the deployment:

```
$ kubectl --namespace kube-system get ds
NAME            DESIRED   CURRENT   READY     NODE-SELECTOR   AGE
cilium          4         4         4         <none>          2m
```

As the pods are deployed, the number in the ready column will increase and eventually reach the desired count.

```
$ kubectl --namespace kube-system describe ds cilium
Name:           cilium
Image(s):       cilium/cilium:stable
Selector:       io.cilium.admin.daemon-set=cilium,name=cilium
Node-Selector:  <none>
Labels:         io.cilium.admin.daemon-set=cilium
                name=cilium
Desired Number of Nodes Scheduled: 1
Current Number of Nodes Scheduled: 1
Number of Nodes Misscheduled: 0
Pods Status:    1 Running / 0 Waiting / 0 Succeeded / 0 Failed
Events:
  FirstSeen     LastSeen        Count   From            SubObjectPath   Type         ␣
 ↪  Reason                  Message
  ---------     --------        -----   ----            -------------   --------     ␣
 ↪  ------                  -------
  35s           35s             1       {daemon-set }                   Normal       ␣
 ↪  SuccessfulCreate        Created pod: cilium-2xzqm
```

We can now check the logfile of a particular cilium agent:

```
$ kubectl --namespace kube-system get pods
NAME           READY     STATUS    RESTARTS   AGE
cilium-2xzqm   1/1       Running   0          41m

$ kubectl --namespce kube-system logs cilium-2xzqm
INFO      _ _ _
INFO  ___|_| |_|_ _ _____
INFO |  _| | | | | |     |
INFO |___|_|_|_|___|_|_|_|
INFO Cilium 0.8.90 f022e2f Thu, 27 Apr 2017 23:17:56 -0700 go version go1.7.5 linux/
 ↪amd64
INFO clang and kernel versions: OK!
INFO linking environment: OK!
[...]
```

Deploying to selected nodes

To deploy Cilium only to a selected list of worker nodes, you can add a NodeSelector to the `cilium-ds.yaml` file like this:

```
spec:
  template:
    spec:
      nodeSelector:
        with-network-plugin: cilium
```

And then label each node where Cilium should be deployed:

```
kubectl label node worker0 with-network-plugin=cilium
kubectl label node worker1 with-network-plugin=cilium
kubectl label node worker2 with-network-plugin=cilium
```

Networking For Existing Pods

In case pods were already running before the Cilium DaemonSet was deployed, these pods will still be connected using the previous networking plugin according to the CNI configuration. A typical example for this is the `kube-dns` service which runs in the `kube-system` namespace by default.

A simple way to change networking for such existing pods is to rely on the fact that Kubernetes automatically restarts pods in a Deployment if they are deleted, so we can simply delete the original kube-dns pod and the replacment pod started immediately after will have networking managed by Cilium. In a production deployment, this step could be performed as a rolling update of kube-dns pods to avoid downtime of the DNS service.

```
$ kubectl --namespace kube-system delete pods -l k8s-app=kube-dns
pod "kube-dns-268032401-t57r2" deleted
```

Running `kubectl get pods` will show you that Kubernetes started a new set of `kube-dns` pods while at the same time terminating the old pods:

```
$ kubectl --namespace kube-system get pods
NAME                          READY     STATUS        RESTARTS   AGE
cilium-5074s                  1/1       Running       0          58m
kube-addon-manager-minikube   1/1       Running       0          59m
kube-dns-268032401-j0vml      3/3       Running       0          9s
kube-dns-268032401-t57r2      3/3       Terminating   0          57m
```

Removing the cilium daemon

All cilium agents are managed as a DaemonSet which means that deleting the DaemonSet will automatically stop and remove all pods which run Cilium on each worker node:

```
$ kubectl --namespace kube-system delete ds cilium
```

Migrating Cilium TPR to CRD

Prior to Kubernetes 1.7, Cilium Network Policy (CNP) objects were imported as a Kubernetes ThirdPartyResource (TPRs). In Kubernetes `>=1.7.0`, TPRs are now deprecated, and will be removed in Kubernetes 1.8. TPRs are replaced by Custom Resource Definitions (CRDs). Thus, as part of the upgrade process to Kubernetes 1.7, Kubernetes has provided documentation for migrating TPRs to CRDS.

The following instructions document how to migrate CiliumNetworkPolicies existing as TPRs from a Kubernetes cluster which was previously running versions `< 1.7.0` to CRDs on a Kubernetes cluster running versions `>= 1.7.0`. This is meant to correspond to steps 4-6 of the aforementioned guide.

Cilium adds the CNP CRD automatically; check to see that the CNP CRD has been added by Cilium:

```
$ kubectl get customresourcedefinition
NAME                             KIND
ciliumnetworkpolicies.cilium.io   CustomResourceDefinition.v1beta1.apiextensions.k8s.
↪io
```

Save your existing CNPs which were previously added as TPRs:

```
$ kubectl get ciliumnetworkpolicies --all-namespaces -o yaml > cnps.yaml
```

Change the version of the Cilium API from v1 to v2 in the YAML file to which you just saved your old CNPs. The Cilium API is versioned to account for the change from TPR to CRD:

```
$ cp cnps.yaml cnps.yaml.new
$ # Edit the version
$ vi cnps.yaml.new
$ # The diff of the old vs. new YAML file should be similar to the output below.
$ diff cnps.yaml cnps.yaml.new
3c3
< - apiVersion: cilium.io/v1
---
> - apiVersion: cilium.io/v2
10c10
<     selfLink: /apis/cilium.io/v1/namespaces/default/ciliumnetworkpolicies/guestbook-
↪web-deprecated
---
>     selfLink: /apis/cilium.io/v2/namespaces/default/ciliumnetworkpolicies/guestbook-
↪web-deprecated
```

Delete your old CNPs:

```
$ kubectl delete ciliumnetworkpolicies --all
$ kubectl delete thirdpartyresource cilium-network-policy.cilium.io
```

Add the changed CNPs back as CRDs:

```
$ kubectl create -f cnps.yaml.new
```

Check that your CNPs are added:

```
$ kubectl get ciliumnetworkpolicies
NAME                         KIND
guestbook-web-deprecated   CiliumNetworkPolicy.v2.cilium.io
multi-rules-deprecated     CiliumNetworkPolicy.v2.cilium.io   Policy to test multiple␣
↪rules in a single file   2 item(s)
```

Now if you try to create a CNP as a TPR, you will get an error:

```
$ Error from server (BadRequest): error when creating "cilium-tpr.yaml": the API␣
↪version in the data (cilium.io/v1) does not match the expected API version (cilium.
↪io/v2)
```

Troubleshooting

See the section *Kubernetes* for Kubernetes specific instructions on how to troubleshoot a setup.

Docker Compose Installation Guide

This section describes how to install & run the Cilium container image using Docker compose.

Note: for multi-host deployments using a key-value store, you would want to update this template to point cilium to a central key-value store.

```
$ wget https://raw.githubusercontent.com/cilium/cilium/master/examples/docker-compose/
↪docker-compose.yml
$ IFACE=eth1 docker-compose up
[...]
```

```
$ docker network create --ipv6 --subnet ::1/112 --ipam-driver cilium --driver cilium␣
↪cilium
$ docker run -d --name foo --net cilium --label id.foo tgraf/nettools sleep 30000
$ docker run -d --name bar --net cilium --label id.bar tgraf/nettools sleep 30000
```

```
$ docker exec -ti foo ping6 -c 4 bar
PING f00d::c0a8:66:0:f236(f00d::c0a8:66:0:f236) 56 data bytes
64 bytes from f00d::c0a8:66:0:f236: icmp_seq=1 ttl=63 time=0.086 ms
64 bytes from f00d::c0a8:66:0:f236: icmp_seq=2 ttl=63 time=0.062 ms
64 bytes from f00d::c0a8:66:0:f236: icmp_seq=3 ttl=63 time=0.061 ms
64 bytes from f00d::c0a8:66:0:f236: icmp_seq=4 ttl=63 time=0.064 ms

--- f00d::c0a8:66:0:f236 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3066ms
rtt min/avg/max/mdev = 0.061/0.068/0.086/0.011 ms
```

CoreOS Installation Guide

This document serves as a guide to get Cilium up-and-running on CoreOS.

If you haven't read the *Introduction to Cilium* yet, we encourage you to do that first.

For an introduction about how to setup Kubernetes with CoreOS, we recommend following along with the Core OS guide. The CoreOS guide serves as a reference to setting up Kubernetes components (e.g., kube-apiserver, kube-scheduler, certificates, etc.), while this page focuses specifically on the modifications needed to get Cilium functioning with CoreOS.

The best way to get help if you get stuck is to ask a question on the Cilium Slack channel. With Cilium contributors across the globe, there is almost always someone available to help.

Minimum Requirements

- Make sure you understand the various deployment options.
- ***etcd***
  - `etcd` must be at version `>= 3.1.0`. To set up etcd, follow the CoreOS instructions for setting up an etcd cluster.
- ***kubectl***
  - It is also required that you install `kubectl` version `>= 1.6.4` as described in the Kubernetes Docs.

Setup Master Nodes

Step 1: Create TLS Assets

Follow the CoreOS instructions for creating TLS assets on master nodes and install the TLS assets needed for the master nodes for secure communication with the kube-apiserver.

Step 2: Setup Kubelet

On CoreOS, Kubelet (the Kubernetes agent that runs on each node, more info here) runs as a container. In order to securely communicate with the API server, kubelet uses the TLS assets we generated as part of Step 1.

Master nodes are usually not scheduled to run workloads, so we provide the `-register-schedulable=false` in the example YAML excerpt below. This ensures that workloads are only scheduled on worker nodes.

Since we are setting up Kubelet to use Cilium, we want to configure its networking to utilize CNI (Container Networking Interface). This ensures that each pod that is created can communicate with one another within the cluster with Cilium networking configured.

- Replace ${ADVERTISE_IP} with this node's publicly routable IP.
- Replace ${DNS_SERVICE_IP}. For more information about what this IP is, refer to the CoreOS documentation for Kubernetes deployment options.
- Replace ${KUBE_VERSION} with a version `>= 1.6.4`.
- Cilium-specific configuration
  - Mount the CNI configuration directory you created in step 1 so Kubelet can pick up the CNI configuration from the host filesystem:

    ```
    --volume etc-cni,kind=host,source=/etc/cni/net.d \
    --mount volume=etc-cni,target=/etc/cni/net.d
    ```

  - Mount the directory where CNI plugins are installed:

    ```
    --volume cni-bin,kind=host,source=/opt/cni/bin \
    --mount volume=cni-bin,target=/opt/cni/bin
    ```

  - Mount the BPF filesystem so that the information stored there persists across Cilium restarts:

    ```
    ExecStartPre=/bin/bash -c ' \\
      if [[ \$(/bin/mount | /bin/grep /sys/fs/bpf -c) -eq 0 ]]; then \\
        /bin/mount bpffs /sys/fs/bpf -t bpf; \\
      fi'
    ```

  - Also ensure that you specify that the network plugin is CNI:

    ```
    --network-plugin=cni
    ```

  - Specify the CNI directory to correspond to the mount you provided earlier where the CNI configuration is located:

    ```
    --cni-conf-dir=/etc/cni/net.d
    ```

/etc/systemd/system/kubelet.service

```
[Service]
Environment=KUBELET_IMAGE_TAG=v'"${KUBE_VERSION}"'_coreos.0
Environment="RKT_RUN_ARGS=--uuid-file-save=/var/run/kubelet-pod.uuid \
  --volume var-log,kind=host,source=/var/log \
  --mount volume=var-log,target=/var/log \
  --volume dns,kind=host,source=/etc/resolv.conf \
  --mount volume=dns,target=/etc/resolv.conf \
  --volume cni-bin,kind=host,source=/opt/cni/bin \
  --mount volume=cni-bin,target=/opt/cni/bin \
  --volume etc-cni,kind=host,source=/etc/cni/net.d \
  --mount volume=etc-cni,target=/etc/cni/net.d"
ExecStartPre=/usr/bin/mkdir -p /etc/cni/net.d
ExecStartPre=/bin/bash -c ' \\
  if [[ \$(/bin/mount | /bin/grep /sys/fs/bpf -c) -eq 0 ]]; then \\
    /bin/mount bpffs /sys/fs/bpf -t bpf; \\
  fi'
ExecStartPre=/usr/bin/mkdir -p /opt/cni/bin
ExecStartPre=/usr/bin/mkdir -p /etc/kubernetes/manifests
ExecStartPre=/usr/bin/mkdir -p /var/log/containers
ExecStartPre=-/usr/bin/rkt rm --uuid-file=/var/run/kubelet-pod.uuid
ExecStart=/usr/lib/coreos/kubelet-wrapper \
  --api-servers=http://127.0.0.1:8080 \
  --register-schedulable=false \
  --cni-conf-dir=/etc/cni/net.d \
  --network-plugin=cni \
  --container-runtime=docker \
  --allow-privileged=true \
  --pod-manifest-path=/etc/kubernetes/manifests \
  --hostname-override=${ADVERTISE_IP} \
  --cluster-dns=${DNS_SERVICE_IP} \
  --cluster-domain=cluster.local
ExecStop=-/usr/bin/rkt stop --uuid-file=/var/run/kubelet-pod.uuid
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
```

Step 3: Setup kube-apiserver on the master nodes

Follow the CoreOS instructions for setting up the API server.

Make sure that you set the version of the kube-apiserver to whatever version you are using for the Kubelet as well, e.g.:

```
quay.io/coreos/hyperkube:v${KUBE_VERSION}_coreos.0
```

Step 4: Setup kube-controller-manager on the master nodes

Per the CoreOS guide, “the controller manager is responsible for reconciling any required actions based on changes to Replication Controllers. For example, if you increased the replica count, the controller manager would generate a scale up event, which would cause a new Pod to get scheduled in the cluster. The controller manager communicates with the API to submit these events.

Create */etc/kubernetes/manifests/kube-controller-manager.yaml*. It will use the TLS certificate placed on disk earlier.”

- Add `--allocate-node-cidrs` to ensure that the kube-controller-manager allocates unique pod CIDR blocks for each node in the cluster.
- Substitute ${CLUSTER_CIDR} with the CIDR range for pods in your cluster.
- Substitute ${SERVICE_CLUSTER_IP_RANGE} with the IP range used for service IPs in your cluster.
- Set NODE_CIDR_MASK_SIZE to a size that you want for each CIDR block on each node.

/etc/kubernetes/manifests/kube-controller-manager.yaml.

```
apiVersion: v1
kind: Pod
metadata:
  name: kube-controller-manager
  namespace: kube-system
spec:
  hostNetwork: true
  containers:
  - name: kube-controller-manager
    image: quay.io/coreos/hyperkube:v'"${KUBE_VERSION}"'_coreos.0
    command:
    - /hyperkube
    - controller-manager
    - --allocate-node-cidrs
    - --cluster-cidr=${CLUSTER_CIDR}
    - --service-cluster-ip-range=${SERVICE_CLUSTER_IP_RANGE}
    - --node-cidr-mask-size=${NODE_CIDR_MASK_SIZE}
    - --master=http://127.0.0.1:8080
    - --leader-elect=true
    - --service-account-private-key-file=/etc/kubernetes/ssl/apiserver-key.pem
    - --root-ca-file=/etc/kubernetes/ssl/ca.pem
    resources:
      requests:
        cpu: 200m
    livenessProbe:
      httpGet:
        host: 127.0.0.1
        path: /healthz
        port: 10252
      initialDelaySeconds: 15
      timeoutSeconds: 15
    volumeMounts:
    - mountPath: /etc/kubernetes/ssl
      name: ssl-certs-kubernetes
      readOnly: true
    - mountPath: /etc/ssl/certs
      name: ssl-certs-host
      readOnly: true
  volumes:
  - hostPath:
      path: /etc/kubernetes/ssl
    name: ssl-certs-kubernetes
  - hostPath:
      path: /usr/share/ca-certificates
    name: ssl-certs-host
```

Step 5: Setup kube-scheduler on the master nodes

Cilium has no special requirements for setting up the kube-scheduler on master nodes. Follow the CoreOS instructions for setting up kube-scheduler.

Make sure that you set the version of the kube-apiserver to whatever version you are using for the Kubelet as well, e.g.:

```
quay.io/coreos/hyperkube:v${KUBE_VERSION}_coreos.0
```

Step 6: Start Services on Nodes

Start kubelet on all nodes:

```
sudo systemctl start kubelet
```

To have kubelet start after a reboot, run:

```
sudo systemctl enable kubelet
```

Step 7: Health Check of Kubernetes Services

Follow the CoreOS instructions to health check Kubernetes services.

Step 8: Setup Kubectl to Communicate With Your Cluster

Follow the CoreOS instructions to download kubectl.

- Replace ${MASTER_HOST} with the master node address or name used in previous steps
- Replace ${CA_CERT} with the absolute path to the ca.pem created in previous steps
- Replace ${ADMIN_KEY} with the absolute path to the admin-key.pem created in previous steps
- Replace ${ADMIN_CERT} with the absolute path to the admin.pem created in previous steps

```
kubectl config set-cluster default-cluster --server=https://${MASTER_IP} --
↪certificate-authority=${CA_CERT} --embed-certs=true
kubectl config set-credentials default-admin --certificate-authority=${CA_CERT} --
↪client-key=${ADMIN_KEY} --client-certificate=${ADMIN_CERT} --embed-certs=true
kubectl config set-context default-system --cluster=default-cluster --user=default-
↪admin
kubectl config use-context default-system
```

This will populate the Kubeconfig file with the contents of the certificates, which is needed for Cilium to authenticate against the Kubernetes API when it is launched in the next step.

Alternatively, you can run the above commands without `--embed-certs=true`, and then mount the paths to the certificates and keys from the host filesystem in *cilium-ds.yaml*.

Follow the CoreOS instructions to validate that kubectl has been configured correctly.

Step 9: Deploy Cilium DaemonSet

- If your cluster is using RBAC, refer to *RBAC integration*.
- **Follow the instructions for *Configuring the DaemonSet* and *Deploying the DaemonSet*. We recommend using the etcd clust**
  - NOTE: before you deploy the cilium DaemonSet, make sure you change the image for cilium to be "latest" instead of "stable". Once Cilium 0.10 is released, this is not necessary.

Setup Worker Nodes

Step 1: Create TLS Assets

Cilium has no special requirements for setting up the TLS assets on worker nodes. Follow the CoreOS instructions for creating TLS assets on worker nodes for secure communication with the `kube-apiserver`.

Step 2: Setup Kubelet

On CoreOS, Kubelet (the Kubernetes agent that runs on each node, more info here) runs as a container. In order to securely communicate with the API server, kubelet uses the TLS assets we generated as part of Step 1.

Since we are setting up Kubelet to use Cilium, we want to configure its networking to utilize CNI (Container Networking Interface). This ensures that each pod that is created can communicate with one another within the cluster with Cilium networking configured.

- Replace ${MASTER_HOST}
- Replace ${ADVERTISE_IP} with this node's publicly routable IP.
- Replace ${DNS_SERVICE_IP}. For more information about what this IP is, refer to the CoreOS documentation for Kubernetes deployment options.
- Replace ${KUBE_VERSION} with a version `>= 1.6.4`.
- Cilium-specific configuration
  - Mount the CNI configuration directory you created in step 1 so Kubelet can pick up the CNI configuration from the host filesystem:

    ```
    --volume etc-cni,kind=host,source=/etc/cni/net.d \
    --mount volume=etc-cni,target=/etc/cni/net.d
    ```

  - Mount the directory where CNI plugins are installed:

    ```
    --volume cni-bin,kind=host,source=/opt/cni/bin \
    --mount volume=cni-bin,target=/opt/cni/bin
    ```

  - Mount the BPF filesystem so that the information stored there persists across Cilium restarts:

    ```
    ExecStartPre=/bin/bash -c ' \\
      if [[ \$(/bin/mount | /bin/grep /sys/fs/bpf -c) -eq 0 ]]; then \\
        /bin/mount bpffs /sys/fs/bpf -t bpf; \\
      fi'
    ```

  - Also ensure that you specify that the network plugin is CNI:

```
--network-plugin=cni
```

- Specify the CNI directory to correspond to the mount you provided earlier where the CNI configuration is located:

```
--cni-conf-dir=/etc/cni/net.d
```

/etc/systemd/system/kubelet.service

```
[Service]
Environment=KUBELET_IMAGE_TAG=v'"${KUBE_VERSION}"'_coreos.0
Environment="RKT_RUN_ARGS=--uuid-file-save=/var/run/kubelet-pod.uuid \
 --volume var-log,kind=host,source=/var/log \
 --mount volume=var-log,target=/var/log \
 --volume dns,kind=host,source=/etc/resolv.conf \
 --mount volume=dns,target=/etc/resolv.conf \
 --volume cni-bin,kind=host,source=/opt/cni/bin \
 --mount volume=cni-bin,target=/opt/cni/bin \
 --volume etc-cni,kind=host,source=/etc/cni/net.d \
 --mount volume=etc-cni,target=/etc/cni/net.d"
ExecStartPre=/bin/bash -c ' \\
  if [[ \$(/bin/mount | /bin/grep /sys/fs/bpf -c) -eq 0 ]]; then \\
    /bin/mount bpffs /sys/fs/bpf -t bpf; \\
  fi'
ExecStartPre=/usr/bin/mkdir -p /etc/cni/net.d
ExecStartPre=/usr/bin/mkdir -p /opt/cni/bin
ExecStartPre=/usr/bin/mkdir -p /etc/kubernetes/manifests
ExecStartPre=/usr/bin/mkdir -p /var/log/containers
ExecStartPre=-/usr/bin/rkt rm --uuid-file=/var/run/kubelet-pod.uuid
ExecStart=/usr/lib/coreos/kubelet-wrapper \
 --api-servers=https://{MASTER_HOST} \
 --register-node=true \
 --cni-conf-dir=/etc/cni/net.d \
 --network-plugin=cni \
 --container-runtime=docker \
 --allow-privileged=true \
 --pod-manifest-path=/etc/kubernetes/manifests \
 --hostname-override=${ADVERTISE_IP} \
 --cluster-dns=${DNS_SERVICE_IP} \
 --kubeconfig=/etc/kubernetes/worker-kubeconfig.yaml \
 --tls-cert-file=/etc/kubernetes/ssl/worker.pem \
 --tls-private-key-file=/etc/kubernetes/ssl/worker-key.pem \
 --cluster-domain=cluster.local \
 --make-iptables-util-chains=false
ExecStop=-/usr/bin/rkt stop --uuid-file=/var/run/kubelet-pod.uuid
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
```

Step 3: Setup Worker kubeconfig

Cilium has no special requirements for setting up the `kubeconfig` for `kubelet` on worker nodes. Please follow the CoreOS instructions to setup the worker-kubeconfig.

Step 4: Start Services

Start kubelet on all nodes:

```
sudo systemctl start kubelet
```

To have kubelet start after a reboot, run:

```
sudo systemctl enable kubelet
```

Step 5: Make Sure Cilium Runs On Worker Nodes

When we deployed Cilium as part of *Step 9: Deploy Cilium DaemonSet*, the Daemon Set expects the Kubeconfig to be located at the same location on each node in the cluster. So, you need to make sure that the location and contents of the kubeconfig for the worker node is the same as that which Cilium is using on the master nodes, e.g., `~/.kube/config`.

Step 6: Setup kubectl and deploy add-ons

Follow the CoreOS instructions for setting up kube-dns and kube-dashboard.

Instsallation From Source

If for some reason you do not want to run Cilium as a contaimer image. Installing it from source is possible as well. It does come with additional dependencies described in *System Requirements*.

1. Download & extract the latest Cilium release from the ReleasesPage

```
$ wget https://github.com/cilium/cilium/archive/v0.10.0.tar.gz
$ tar xzvf v0.10.0.tar.gz
$ cd cilium-0.10.0
```

2. Build & install the Cilium binaries to `bindir`

```
$ make
$ sudo make install
```

3. Optional: Install systemd/upstart init files:

```
sudo cp contrib/upstart/* /etc/init/
service cilium start
```

Agent Command Reference

Command Line Options

Option	Description	Default
config	config file	$HOME/ciliumd.yaml
debug	Enable debug messages	false
device	Ethernet device to snoop on	
disable-conntrack	Disable connection tracking	false
enable-policy	Enable policy enforcement (default, false, true)	default
docker	Docker socket endpoint	
enable-tracing	enable policy tracing	
nat46-range	IPv6 range to map IPv4 addresses to	
k8s-api-server	Kubernetes api address server	
k8s-kubeconfig-path	Absolute path to the kubeconfig file	
keep-config	When restoring state, keeps containers' configuration in place	false
kvstore	Key Value Store Type: (consul, etcd)	
kvstore-opt	**Etcd:** • etcd.address: Etcd agent address. • etcd.config: Absolute path to the etcd configuration file. **Consul:** • consul.address: Consul agent agent address.	
label-prefix-file	file with label prefixes cilium Cilium should use for policy	
labels	list of label prefixes Cilium should use for policy	
logstash	enable logstash integration	false
logstash-agent	logstash agent address and port	127.0.0.1:8080
node-address	IPv6 address of the node	
restore	Restore state from previously running version of the agent	false
keep-templates	do not restore templates from binary	false
state-dir	path to store runtime state	
lib-dir	path to store runtime build env	
socket-path	path for agent unix socket	
lb	enables load-balancing mode on interface 'device'	
disable-ipv4	disable IPv4 mode	false
ipv4-range	IPv4 prefix	
tunnel	Overlay/tunnel mode (vxlan/geneve)	vxlan
bpf-root	Path to mounted BPF filesystem	
access-log	Path to HTTP access log	

Key-Value Store

Option	Description	Default
–kvstore TYPE	Key Value Store Type: (consul, etcd)	
–kvstore-opt OPTS		

consul

When using consul, the consul agent address needs to be provided with the `consul.address`:

Option	Type	Description
consul.address	Address	Address of consul agent

etcd

When using etcd, one of the following options need to be provided to configure the etcd endpoints:

Option	Type	Description
etcd.address	Address	Address of etcd endpoint
etcd.config	Path	Path to an etcd configuration file.

Example of the etcd configuration file:

```
---
endpoints:
- https://192.168.0.1:2379
- https://192.168.0.2:2379
ca-file: '/var/lib/cilium/etcd-ca.pem'
# In case you want client to server authentication
key-file: '/var/lib/cilium/etcd-client.key'
cert-file: '/var/lib/cilium/etcd-client.crt'
```

CHAPTER 5

Network Policy

This chapter specifies the policy language used to configure network policies in Cilium. Security policies can be specified and imported via the following mechanisms:

- Using Kubernetes `NetworkPolicy` and `CiliumNetworkPolicy` resources. See the section *k8s_policy* for more details. In this mode, Kubernetes will automatically distribute the policies to all agents.
- Directly imported via the *API Reference* of the agent. This method does automatically distribute policies to all agents. It is in the responsibility of the API user to import the policy in all required agents.

New in version future: Use of the KVstore to distribute security policies is on the roadmap but has not been implemented yet.

Policy Language

All policy rules share a common base type which specifies what endpoints the rule applies to and also carries common metadata to identify the rule.

Rule

Each rule is split into an ingress section which contains the rules which must be applied at ingress and egress of all endpoints matching the endpoint selector. Either ingress, egress, or both can be provided. If both ingress and egress are omitted, the rule has no effect.

```
type Rule struct {
        // EndpointSelector selects all endpoints which should be subject to
        // this rule. Cannot be empty.
        EndpointSelector EndpointSelector `json:"endpointSelector"`

        // Ingress is a list of IngressRule which are enforced at ingress.
        // If omitted or empty, this rule does not apply at ingress.
        //
        // +optional
        Ingress []IngressRule `json:"ingress,omitempty"`

        // Egress is a list of EgressRule which are enforced at egress.
        // If omitted or empty, this rule does not apply at egress.
        //
        // +optional
        Egress []EgressRule `json:"egress,omitempty"`

        // Labels is a list of optional strings which can be used to
        // re-identify the rule or to store metadata. It is possible to lookup
        // or delete strings based on labels. Labels are not required to be
        // unique, multiple rules can have overlapping or identical labels.
        //
        // +optional
        Labels labels.LabelArray `json:"labels,omitempty"`

        // Description is a free form string, it can be used by the creator of
        // the rule to store human readable explanation of the purpose of this
        // rule. Rules cannot be identified by comment.
        //
        // +optional
        Description string `json:"description,omitempty"`
}
```

endpointSelector Selects the endpoints to which the policy rules contained must be applied to. All endpoints which match the labels specified in the `endpointSelector` will have the policy rules applied to. See the LabelSelector section in the Kubernetes documentation for the definition of an `endpointSelector`.

ingress List of rules which must apply at ingress of the endpoint, i.e. to all network packets which are entering the endpoint.

egress List of rules which must apply at egress of the endpoint, i.e. to all network packets which are leaving the endpoint.

labels Labels are used to identify the rule. Rules can be listed and deleted by labels. Policy rules which are imported via *Kubernetes* automatically get the label `io.cilium.k8s-policy-name=NAME` assigned where `NAME` corresponds to the name specified in the `NetworkPolicy` or `CiliumNetworkPolicy` resource.

description Description is a string which is not interpreted by Cilium. It can be used to describe the intent and scope of the rule in a human readable form.

Layer 3: Labels-Based

The L3 policy specifies which endpoints can talk to each other. L3 policies can be specified using *Labels* or CIDR. For CIDR, refer to the *Layer 3: IP/CIDR based* section below.

Label-based L3 policy is used to establish policy between endpoints inside the cluster. An endpoint is allowed to talk to another endpoint if at least one rule exists which selects the destination endpoint with the `endpointSelector` while also selecting the source endpoint in the `fromEndpoints` field. Like `endpointSelector`, the `fromEndpoints` is specified as a LabelSelector.

For more complex label combinations, the `fromRequires` field can be used to establish label requirements which apply to multiple destinations. `fromRequires` is a list of additional constraints which must be met in order for the selected endpoints to be reachable. These additional constraints do no by itself grant access privileges and must always be accompanied with at least one matching fromEndpoints.

```
type IngressRule struct {
        // FromEndpoints is a list of endpoints identified by an
        // EndpointSelector which are allowed to communicate with the endpoint
        // subject to the rule.
        //
        // Example:
        // Any endpoint with the label "role=backend" can be consumed by any
        // endpoint carrying the label "role=frontend".
        //
        // +optional
        FromEndpoints []EndpointSelector `json:"fromEndpoints,omitempty"`

        // FromRequires is a list of additional constraints which must be met
        // in order for the selected endpoints to be reachable. These
        // additional constraints do no by itself grant access privileges and
        // must always be accompanied with at least one matching FromEndpoints.
        //
        // Example:
        // Any Endpoint with the label "team=A" requires consuming endpoint
        // to also carry the label "team=A".
        //
        // +optional
        FromRequires []EndpointSelector `json:"fromRequires,omitempty"`

        // [...]
}
```

Example (Basic)

This example shows to enable all endpoints with the label `role=frontend` to communicate with all endpoints with the label `role=backend`:

```
[{
    "endpointSelector": {"matchLabels": {"role":"backend"}},
    "ingress": [{
        "fromEndpoints": [
          {"matchLabels":{"role":"frontend"}}
        ]
    }]
}]
```

Example (Requires)

The following example builds on top of the previous one but requires that *all* endpoints which carry the label `env=prod` require the consumer to also carry the label `env=prod` in order for access to be granted:

```
[{
    "endpointSelector": {"matchLabels": {"env":"prod"}},
    "ingress": [{
        "fromRequires": [
          {"matchLabels":{"env":"prod"}}
        ]
    }]
},{
    "endpointSelector": {"matchLabels": {"role":"backend"}},
    "ingress": [{
        "fromEndpoints": [
          {"matchLabels":{"role":"frontend"}}
        ]
    }]
}]
```

Layer 3: IP/CIDR based

CIDR policies are used to define policies to and from endpoints which are not managed by a container orchestration system and thus do not have labels associated with them. These are typically VMs or bare metal machines with static IP addresses. CIDR policy can also be used to limit access to external services, for example to limit external access to a particular IP range.

CIDR policies can be applied at ingress and egress:

```
type IngressRule struct {
        // FromCIDR is a list of IP blocks which the endpoint subject to the
        // rule is allowed to receive connections from in addition to FromEndpoints.
        // This will match on the source IP address of incoming connections.
        //
        // Example:
        // Any endpoint with the label "app=my-legacy-pet" is allowed to receive
        // connections from 10.3.9.1
        //
        // +optional
        FromCIDR []CIDR `json:"fromCIDR,omitempty"`

        // [...]
}

type EgressRule struct {
        // ToCIDR is a list of IP blocks which the endpoint subject to the rule
        // is allowed to initiate connections to in addition to connections
        // which are allowed via FromEndpoints. This will match on the
        // destination IP address of outgoing connections.
        //
        // Example:
        // Any endpoint with the label "app=database-proxy" is allowed to
        // initiate connections to 10.2.3.0/24
        //
        // +optional
        ToCIDR []CIDR `json:"toCIDR,omitempty"`

        // [...]
}
```

fromCIDR List of source prefixes/CIDRs that are allowed to talk to all endpoint selected by the `endpointSelector`. Note that this list is **in addition** to the `fromEndpoints` specified. It is not required to allow the IPs of endpoints if the endpoints are already allowed to communicate based on `fromEndpoints` rules.

toCIDR: List of destination prefixes/CIDRs that endpoints selected by `endpointSelector` are allowed to talk to. Note that endpoints which are selected by a `fromEndpoints` are automatically allowed to talk to their respective destination endpoints. It is not required to list the IP of destination endpoints.

Example

This example shows how to allow all endpoints with the label `app=myService` to talk to the external IP `20.1.1.1`

```
[{
    "endpointSelector": {"matchLabels":{"app":"myService"}},
    "egress": [{
        "toCIDR": [
            "20.1.1.1/32"
        ]
    }]
}]
```

Layer 4: Ports

L4 policy can be specified in addition to L3 policies. It restricts the ability of an endpoint to emit and/or receive packets on a particular port using a particular protocol. If no L4 policy is specified for an endpoint, that endpoint is allowed to send and receive on all L4 ports and protocols.

L4 policy can be specified at both ingress and egress using the *toPorts* field:

```
type IngressRule struct {
        // ToPorts is a list of destination ports identified by port number and
        // protocol which the endpoint subject to the rule is allowed to
        // receive connections on.
        //
        // Example:
        // Any endpoint with the label "app=httpd" can only accept incoming
        // connections on port 80/tcp.
        //
        // +optional
        ToPorts []PortRule `json:"toPorts,omitempty"`

        // [...]
}

type EgressRule struct {
        // ToPorts is a list of destination ports identified by port number and
        // protocol which the endpoint subject to the rule is allowed to
        // connect to.
        //
        // Example:
        // Any endpoint with the label "role=frontend" is allowed to initiate
        // connections to destination port 8080/tcp
        //
        // +optional
        ToPorts []PortRule `json:"toPorts,omitempty"`

        // [...]
}
```

The `toPorts` field takes a `PortProtocol` structure which is defined as follows:

```
// PortProtocol specifies an L4 port with an optional transport protocol
type PortProtocol struct {
        // Port is an L4 port number. For now the string will be strictly
        // parsed as a single uint16. In the future, this field may support
        // ranges in the form "1024-2048
        Port string `json:"port"`

        // Protocol is the L4 protocol. If omitted or empty, any protocol
        // matches. Accepted values: "tcp", "udp", ""/"any"
        //
        // Matching on ICMP is not supported.
        //
        // +optional
        Protocol string `json:"protocol,omitempty"`
}
```

Example (L4)

The following rule limits all endpoints with the label `app=myService` to only be able to emit packets using TCP on port 80:

```
{
    "endpointSelector": {"matchLabels":{"app":"myService"}},
    "egress": [{
        "toPorts": [
            {"port": "80", "protocol": "tcp"}
        ]
    }]
}
```

Layer 7 - HTTP

Layer 7 policy can be specified embedded into *policy_l4* rules. The `L7Rules` structure is a base type containing an enumeration of protocol specific fields which will be extended as Cilium starts supporting additional layer 7 protocols. Only one field can be specified at the time.

Layer 7 policies can be specified for ingress and egress *policy_l4* rules:

```
// L7Rules is a union of port level rule types. Mixing of different port
// level rule types is disallowed, so exactly one of the following must be set.
// If none are specified, then no additional port level rules are applied.
type L7Rules struct {
        // HTTP specific rules.
        //
        // +optional
        HTTP []PortRuleHTTP `json:"http,omitempty"`
}
```

HTTP If specified, will restrict all HTTP requests which are sent or received on the `PortProtocol` to which the `L7Rules` to the list of specified request patterns.

HTTP Policy

Unlike L3 and L4 policies, violation of Layer 7 rules does not result in packet drops. Instead, if possible, an access denied message such as an *HTTP 403 access denied* is sent back to the sending endpoint.

```
// PortRuleHTTP is a list of HTTP protocol constraints. All fields are
// optional, if all fields are empty or missing, the rule does not have any
// effect.
//
// All fields of this type are extended POSIX regex as defined by IEEE Std
// 1003.1, (i.e this follows the egrep/unix syntax, not the perl syntax)
// matched against the path of an incoming request. Currently it can contain
// characters disallowed from the conventional "path" part of a URL as defined
// by RFC 3986.
type PortRuleHTTP struct {
        // Path is an extended POSIX regex matched against the path of a
        // request. Currently it can contain characters disallowed from the
        // conventional "path" part of a URL as defined by RFC 3986. Paths must
        // begin with a '/'.
        //
        // If omitted or empty, all paths are all allowed.
        //
        // +optional
        Path string `json:"path,omitempty" protobuf:"bytes,1,opt,name=path"`

        // Method is an extended POSIX regex matched against the method of a
        // request, e.g. "GET", "POST", "PUT", "PATCH", "DELETE", ...
        //
        // If omitted or empty, all methods are allowed.
        //
        // +optional
        Method string `json:"method,omitempty" protobuf:"bytes,1,opt,name=method"`

        // Host is an extended POSIX regex matched against the host header of a
        // request, e.g. "foo.com"
        //
        // If omitted or empty, the value of the host header is ignored.
        //
        // +optional
        Host string `json:"host,omitempty" protobuf:"bytes,1,opt,name=method"`

        // Headers is a list of HTTP headers which must be present in the
        // request. If omitted or empty, requests are allowed regardless of
        // headers present.
        //
        // +optional
        Headers []string `json:"headers,omitempty"`
}
```

Path When specified, the path field of the request must match the regular expression specified.

Method When specified, the method name of the request must match the regular expression specified.

Host When specified, the host field of the request must match the regular expression specified.

Headers When specified, the request must contain all the headers specified in the list.

Example (HTTP)

The following example limits all endpoints which carry the labels `app=myService` to only be able to receive packets on port 80 using TCP. While communicating on this port, the only API endpoints allowed will be `GET /path1` and `PUT /path2` with the HTTP header `X-My_header` set to `true`:

```
{
    "endpointSelector": {"matchLabels":{"app":"myService"}},
    "ingress": [{
        "toPorts": [{
            "ports": [
                {"port": "80", "protocol": "tcp"}
            ],
            "rules": {
                "HTTP": [
                    {
                        "method": "GET",
                        "path": "/path1$"
                    },{
                        "method": "PUT",
                        "path": "/path2$",
                        "headers": ["X-My-Header: true"]
                    }
                ]
            }
        }]
    }]
}
```

Integrations

Kubernetes

If you are running Cilium on Kubernetes, you can benefit from the native support of network security policies. In this mode, Kubernetes is responsible for distributing the policies across all nodes and Cilium will automatically apply the policies. Two formats are available to configure network policies natively with Kubernetes:

- The standard NetworkPolicy resource which at the time of this writing, supports to specify L3/L4 ingress policies. See the official Kubernetes documentation on NetworkPolicy for details on how to specify such policies.
- The extended `CiliumNetworkPolicy` format which is available as a ThirdPartyResource and CustomResourceDefinition which supports to specify L3/L4/L7 policies at both ingress and egress.

JSON specification for `CiliumNetworkPolicy`:

```
// CiliumNetworkPolicy is a Kubernetes third-party resource with an
// extended version of NetworkPolicy
type CiliumNetworkPolicy struct {
        metav1.TypeMeta `json:",inline"`
        // +optional
        Metadata metav1.ObjectMeta `json:"metadata"`

        // Spec is the desired Cilium specific rule specification.
        Spec *api.Rule `json:"spec,omitempty"`

        // Specs is a list of desired Cilium specific rule specification.
        Specs api.Rules `json:"specs,omitempty"`
}
```

A `CiliumNetworkPolicy` can contain either a single rule (`Spec`) or a list of rules (`Specs`). If a list of rules are specified, all rules will be added and removed atomically. This is useful if multiple rules depend on each other and may not be applied independently.

The `Spec` respectively `Specs` field refers to a standard Cilium Policy Rule.

Tracing

If Cilium is denying connections which it shouldn't. There is an easy way to verify if and why Cilium is denying connectivity in between particular endpoints. The following example shows how to use `cilium policy trace` to simulate a policy decision from an endpoint with the label `id.curl` to an endpoint with the label `id.http` on port 80:

```
$ cilium policy trace -s id.curl -d id.httpd --dport 80
Tracing From: [container:id.curl] => To: [container:id.httpd] Ports: [80/any]
* Rule 2 {"matchLabels":{"any:id.httpd":""}}: match
    Allows from labels {"matchLabels":{"any:id.curl":""}}
+     Found all required labels
1 rules matched
Result: ALLOWED
L3 verdict: allowed

Resolving egress port policy for [container:id.curl]
* Rule 0 {"matchLabels":{"any:id.curl":""}}: match
  Allows Egress port [{80 tcp}]
1 rules matched
L4 egress verdict: allowed

Resolving ingress port policy for [container:id.httpd]
* Rule 2 {"matchLabels":{"any:id.httpd":""}}: match
  Allows Ingress port [{80 tcp}]
1 rules matched
L4 ingress verdict: allowed

Verdict: allowed
```

CHAPTER 6

Troubleshooting

This document describes how to troubleshoot Cilium in different deployment modes. It focuses on a full deployment of Cilium within a datacenter or public cloud. If you are just looking for a simple way to experiment, we highly recommend trying out the *Getting Started* instead.

This guide assumes that you have read the *Concepts* which explains all the components and concepts.

Kubernetes

Check the status of the DaemonSet and verify that all desired instances are in "ready" state:

```
$ kubectl --namespace kube-system get ds
NAME      DESIRED   CURRENT   READY     NODE-SELECTOR   AGE
cilium    1         1         0         <none>          3s
```

In this example, we see a desired state of 1 with 0 being ready. This indicates a problem. The next step is to list all cilium pods by matching on the label `k8s-app=cilium` and also sort the list by the restart count of each pod to easily identify the failing pods:

```
$ kubectl --namespace kube-system get pods --selector k8s-app=cilium \
          --sort-by='.status.containerStatuses[0].restartCount'
NAME           READY     STATUS             RESTARTS   AGE
cilium-813gf   0/1       CrashLoopBackOff   2          44s
```

Pod `cilium-813gf` is failing and has already been restarted 2 times. Let's print the logfile of that pod to investigate the cause:

```
$ kubectl --namespace kube-system logs cilium-813gf
INFO      _ _ _
INFO  ___|_| |_|_ _ _____
INFO |  _| | | | | |     |
INFO |___|_|_|_|___|_|_|_|
INFO Cilium 0.8.90 f022e2f Thu, 27 Apr 2017 23:17:56 -0700 go version go1.7.5 linux/
↪amd64
CRIT kernel version: NOT OK: minimal supported kernel version is >= 4.8
```

In this example, the cause for the failure is a Linux kernel running on the worker node which is not meeting *System Requirements*.

If the cause for the problem is not apparent based on these simple steps, please come and seek help on our Slack channel.

Monitoring Packet Drops

When connectivity is not as it should be. A main cause can be unwanted packet drops on the networking level. There can be various causes for this. The tool `cilium monitor` allows you to quickly inspect and see if and where packet drops happen.

```
$ cilium monitor
Listening for events on 2 CPUs with 64x4096 of shared memory
Press Ctrl-C to quit

CPU 00: MARK 0x14126c56 FROM 56326 Packet dropped 159 (Policy denied (L4)) 94 bytes␣
↪ifindex=18
00000000  02 fd 7f 53 22 c8 66 56  da 2e fb 84 86 dd 60 0c  |...S".fV......`.|
00000010  12 14 00 28 06 3f f0 0d  00 00 00 00 00 00 0a 00  |...(.?..........|
00000020  02 0f 00 00 00 ad f0 0d  00 00 00 00 00 00 0a 00  |................|
00000030  02 0f 00 00 dc 06 ca 5c  00 50 70 28 32 21 00 00  |.......\.Pp(2!..|
00000040  00 00 a0 02 6c 98 d5 1b  00 00 02 04 05 6e 04 02  |....l........n..|
00000050  08 0a 01 5f 07 80 00 00  00 00 01 03 03 07 00 00  |..._............|
00000060  00 00 00 00                                       |....|
```

The above indicates that a packet from endpoint ID *56326* has been dropped due to violation of the Layer 4 policy.

Policy Tracing

See section policy_tracing_ for details and examples on how to use the policy tracing feature.

Debugging the datapath

The tool `cilium monitor` can also be used to retrieve debugging information from the BPF based datapath. Debugging messages are sent if either the `cilium-agent` itself or the respective endpoint is in debug mode. The debug mode of the agent can be enabled by starting `cilium-agent` with the option `--debug` enabled or by running `cilium config debug=true` for an already running agent. Debugging of an individual endpoint can be enabled by running `cilium endpoint config ID Debug=true`

```
$ cilium endpoint config 29381 Debug=true
Endpoint 29381 configuration updated successfully
$ cilium monitor
CPU 01: MARK 0x3c7a42a5 FROM 13949 DEBUG: 118 bytes Incoming packet from container␣
↪ifindex 20
00000000  3a f3 07 b3 c6 7f 4e 76  63 5c 53 4e 86 dd 60 02  |:.....Nvc\SN..`.|
00000010  7a 3c 00 40 3a 40 f0 0d  00 00 00 00 00 00 0a 00  |z<.@:@..........|
00000020  02 0f 00 00 36 7d f0 0d  00 00 00 00 00 00 0a 00  |....6}..........|
00000030  02 0f 00 00 ff ff 81 00  c7 05 4a 32 00 05 29 98  |..........J2..).|
00000040  2c 59 00 00 00 00 1d cd  0c 00 00 00 00 00 10 11  |,Y..............|
00000050  12 13 14 15 16 17 18 19  1a 1b 1c 1d 1e 1f 20 21  |.............. !|
00000060  22 23 24 25 26 27 28 29  2a 2b 2c 2d 2e 2f 30 31  |"#$%&'()*+,-./01|
00000070  32 33 34 35 36 37 00 00                           |234567..|

CPU 01: MARK 0x3c7a42a5 FROM 13949 DEBUG: Handling ICMPv6 type=129
CPU 01: MARK 0x3c7a42a5 FROM 13949 DEBUG: CT reverse lookup: sport=0 dport=32768␣
↪nexthdr=58 flags=1
CPU 01: MARK 0x3c7a42a5 FROM 13949 DEBUG: CT entry found lifetime=24026, proxy_port=0␣
↪revnat=0
CPU 01: MARK 0x3c7a42a5 FROM 13949 DEBUG: CT verdict: Reply, proxy_port=0 revnat=0
CPU 01: MARK 0x3c7a42a5 FROM 13949 DEBUG: Going to host, policy-skip=1
CPU 00: MARK 0x4010f7f3 FROM 13949 DEBUG: CT reverse lookup: sport=2048 dport=0␣
↪nexthdr=1 flags=0
CPU 00: MARK 0x4010f7f3 FROM 13949 DEBUG: CT lookup address: 10.15.0.1
CPU 00: MARK 0x4010f7f3 FROM 13949 DEBUG: CT lookup: sport=0 dport=2048 nexthdr=1␣
↪flags=1
CPU 00: MARK 0x4010f7f3 FROM 13949 DEBUG: CT verdict: New, proxy_port=0 revnat=0
CPU 00: MARK 0x4010f7f3 FROM 13949 DEBUG: CT created 1/2: sport=0 dport=2048␣
↪nexthdr=1 flags=1 proxy_port=0 revnat=0
CPU 00: MARK 0x4010f7f3 FROM 13949 DEBUG: CT created 2/2: 10.15.42.252 revnat=0
CPU 00: MARK 0x4010f7f3 FROM 13949 DEBUG: CT created 1/2: sport=0 dport=0 nexthdr=1␣
↪flags=3 proxy_port=0 revnat=0
CPU 00: MARK 0x4010f7f3 FROM 13949 DEBUG: 98 bytes Delivery to ifindex 20
00000000  4e 76 63 5c 53 4e 3a f3  07 b3 c6 7f 08 00 45 00  |Nvc\SN:.......E.|
00000010  00 54 d8 41 40 00 3f 01  24 4d 0a 0f 00 01 0a 0f  |.T.A@.?.$M......|
00000020  2a fc 08 00 67 03 4a 4f  00 01 2a 98 2c 59 00 00  |*...g.JO..*.,Y..|
00000030  00 00 24 e8 0c 00 00 00  00 00 10 11 12 13 14 15  |..$.............|
00000040  16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25  |.......... !"#$%|
00000050  26 27 28 29 2a 2b 2c 2d  2e 2f 30 31 32 33 34 35  |&'()*+,-./012345|
00000060  36 37 00 00 00 00 00 00                           |67......|
```

CHAPTER 7

BPF and XDP Reference Guide

Note: This documentation section is targeted at developers and users who want to understand BPF and XDP in great technical depth. While reading this reference guide may help broaden your understanding of Cilium, it is not a requirement to use Cilium. Please refer to the *Getting Started* and *Concepts* for a higher level introduction.

BPF is a highly flexible and efficient virtual machine-like construct in the Linux kernel allowing to execute bytecode at various hook points in a safe manner. It is used in a number of Linux kernel subsystems, most prominently networking, tracing and security (e.g. sandboxing).

Although BPF exists since 1992, this document covers the extended Berkley Paket Filter (eBPF) version which has first appeared in Kernel 3.18 and renders the original version which is being referred to as "classic" BPF (cBPF) these days mostly obsolete. cBPF is known to many as being the packet filter language used by tcpdump. Nowadays, the Linux kernel runs eBPF only and loaded cBPF bytecode is transparently translated into an eBPF representation in the kernel before program execution. This documentation will generally refer to the term BPF unless explicit differences between eBPF and cBPF are being pointed out.

Even though the name Berkley Packet Filter hints at a packet filtering specific purpose, the instruction set is generic and flexible enough these days that there are many use cases for BPF apart from networking. See *Further Reading* for a list of projects which use BPF.

Cilium uses BPF heavily in its data path, see *Concepts* for further information. The goal of this chapter is to provide a BPF reference guide in oder to gain understanding of BPF, its networking specific use including loading BPF programs with tc (traffic control) and XDP (eXpress Data Path), and to aid with developing Cilium's BPF templates.

BPF Architecture

BPF does not define itself by only providing its instruction set, but also by offering further infrastructure around it such as maps which act as efficient key / value stores, helper functions to interact with and leverage kernel functionality, tail calls for calling into other BPF programs, security hardening primitives, a pseudo file system for pinning objects (maps, programs), and infrastructure for allowing BPF to be offloaded, for example, to a network card.

LLVM provides a BPF back end, so that tools like clang can be used to compile C into a BPF object file, which can then be loaded into the kernel. BPF is deeply tied to the Linux kernel and allows for full programmability without sacrificing native kernel performance.

Last but not least, also the kernel subsystems making use of BPF are part of BPF's infrastructure. The two main subsystems discussed throughout this document are tc and XDP where BPF programs can be attached to. XDP BPF programs are attached at the earliest networking driver stage and trigger a run of the BPF program upon packet reception. By definition, this achieves the best possible packet processing performance since packets cannot get processed at an even earlier point in software. Driver support is necessary in order to use XDP BPF programs, though. However, tc BPF programs don't need any driver support and can be attached to receive and transmit paths of any networking device, including virtual ones such as `veth` devices since they hook later in the kernel stack compared to XDP. Apart from tc and XDP programs, there are various other kernel subsystems as well which use BPF such as tracing (kprobes, uprobes, tracepoints, etc).

The following subsections provide further details on individual aspects of the BPF architecture.

Instruction Set

BPF is a general purpose RISC instruction set and was originally designed for the purpose of writing programs in a subset of C which can be compiled into BPF instructions through a compiler back end (e.g. LLVM), so that the kernel can later on map them through an in-kernel JIT compiler into native opcodes for optimal execution performance inside the kernel.

The advantages for pushing these instructions into the kernel include:

- Making the kernel programmable without having to cross kernel / user space boundaries. For example, BPF programs related to networking, as in the case of Cilium, can implement flexible container policies, load balancing and other means without having to move packets to user space and back into the kernel. State between BPF programs and kernel / user space can still be shared through maps whenever needed.
- Given the flexibility of a programmable data path, programs can be heavily optimized for performance also by compiling out features that are not required for the use cases the program solves. For example, if a container does not require IPv4, then the BPF program can be built to only deal with IPv6 in order to save resources in the fast-path.
- In case of networking (e.g. tc and XDP), BPF programs can be updated atomically without having to restart the kernel, system services or containers, and without traffic interruptions. Furthermore, any program state can also be maintained throughout updates via BPF maps.
- BPF provides a stable ABI towards user space, and does not require any third party kernel modules. BPF is a core part of the Linux kernel that is shipped everywhere, and guarantees that existing BPF programs keep running with newer kernel versions. This guarantee is the same guarantee that the kernel provides for system calls with regard to user space applications.
- BPF programs work in concert with the kernel, they make use of existing kernel infrastructure (e.g. drivers, netdevices, tunnels, protocol stack, sockets) and tooling (e.g. iproute2) as well as the safety guarantees which the kernel provides. Unlike kernel modules, BPF programs are verified through an in-kernel verifier in order to ensure that they cannot crash the kernel, always terminate, etc. XDP programs, for example, reuse the existing in-kernel drivers and operate on the provided DMA buffers containing the packet frames without exposing them or an entire driver to user space as in other models. Moreover, XDP programs reuse the existing stack instead of bypassing it. BPF can be considered a generic "glue code" to kernel facilities for crafting programs to solve specific use cases.

The execution of a BPF program inside the kernel is always event driven! For example, a networking device which has a BPF program attached on its ingress path will trigger the execution of the program once a packet is received, a kernel address which has a kprobes with a BPF program attached will trap once the code at that address gets executed, then invoke the kprobes callback function for instrumentation which subsequently triggers the execution of the BPF program attached to it.

BPF consists of eleven 64 bit registers with 32 bit subregisters, a program counter and a 512 byte large BPF stack space. Registers are named `r0` - `r10`. The operating mode is 64 bit by default, the 32 bit subregisters can only be accessed through special ALU (arithmetic logic unit) operations. The 32 bit lower subregisters zero-extend into 64 bit when they are being written to.

Register `r10` is the only register which is read-only and contains the frame pointer address in order to access the BPF stack space. The remaining `r0` - `r9` registers are general purpose and of read/write nature.

A BPF program can call into a predefined helper function, which is defined by the core kernel (never by modules). The BPF calling convention is defined as follows:

- `r0` contains the return value of a helper function call.
- `r1` - `r5` hold arguments from the BPF program to the kernel helper function.
- `r6` - `r9` are callee saved registers that will be preserved on helper function call.

The BPF calling convention is generic enough to map directly to `x86_64`, `arm64` and other ABIs, thus all BPF registers map one to one to HW CPU registers, so that a JIT only needs to issue a call instruction, but no additional extra moves for placing function arguments. This calling convention was modeled to cover common call situations without having a performance penalty. Calls with 6 or more arguments are currently not supported. The helper functions in the kernel which are dedicated to BPF (`BPF_CALL_0()` to `BPF_CALL_5()` functions) are specifically designed with this convention in mind.

Register `r0` is also the register containing the exit value for the BPF program. The semantics of the exit value are defined by the type of program. Furthermore, when handing execution back to the kernel, the exit value is passed as a 32 bit value.

Registers `r1` - `r5` are scratch registers, meaning the BPF program needs to either spill them to the BPF stack or move them to callee saved registers if these arguments are to be reused across multiple helper function calls. Spilling means that the variable in the register is moved to the BPF stack. The reverse operation of moving the variable from the BPF stack to the register is called filling. The reason for spilling/filling is due to the limited number of registers.

Upon entering execution of a BPF program, register `r1` initially contains the context for the program. The context is the input argument for the program (similar to `argc/argv` pair for a typical C program). BPF is restricted to work on a single context. The context is defined by the program type, for example, a networking program can have a kernel representation of the network packet (`skb`) as the input argument.

The general operation of BPF is 64 bit to follow the natural model of 64 bit architectures in order to perform pointer arithmetics, pass pointers but also pass 64 bit values into helper functions, and to allow for 64 bit atomic operations.

The maximum instruction limit per program is restricted to 4096 BPF instructions, which, by design, means that any program will terminate quickly. Although the instruction set contains forward as well as backward jumps, the in-kernel BPF verifier will forbid loops so that termination is always guaranteed. Since BPF programs run inside the kernel, the verifier's job is to make sure that these are safe to run, not affecting the system's stability. This means that from an instruction set point of view, loops can be implemented, but the verifier will restrict that. However, there is also a concept of tail calls that allows for one BPF program to jump into another one. This, too, comes with an upper nesting limit of 32 calls, and is usually used to decouple parts of the program logic, for example, into stages.

The instruction format is modeled as two operand instructions, which helps mapping BPF instructions to native instructions during JIT phase. The instruction set is of fixed size, meaning every instruction has 64 bit encoding. Currently, 87 instructions have been implemented and the encoding also allows to extend the set with further instructions when needed. The instruction encoding of a single 64 bit instruction is defined as a bit sequence from most significant bit (MSB) to least significant bit (LSB) of `op:8`, `dst_reg:4`, `src_reg:4`, `off:16`, `imm:32`. `off` and `imm` is of signed type. The encodings are part of the kernel headers and defined in `linux/bpf.h` header, which also includes `linux/bpf_common.h`.

`op` defines the actual operation to be performed. Most of the encoding for `op` has been reused from cBPF. The operation can be based on register or immediate operands. The encoding of `op` itself provides information on which mode to use (`BPF_X` for denoting register-based operations, and `BPF_K` for immediate-based operations respectively).

In the latter case, the destination operand is always a register. Both dst_reg and src_reg provide additional information about the register operands to be used (e.g. r0 - r9) for the operation. off is used in some instructions to provide a relative offset, for example, for addressing the stack or other buffers available to BPF (e.g. map values, packet data, etc), or jump targets in jump instructions. imm contains a constant / immediate value.

The available op instructions can be categorized into various instruction classes. These classes are also encoded inside the op field. The op field is divided into (from MSB to LSB) code:4, source:1 and class:3. class is the more generic instruction class, code denotes a specific operational code inside that class, and source tells whether the source operand is a register or an immediate value. Possible instruction classes include:

- BPF_LD, BPF_LDX: Both classes are for load operations. BPF_LD is used for loading a double word as a special instruction spanning two instructions due to the imm:32 split, and for byte / half-word / word loads of packet data. The latter was carried over from cBPF mainly in order to keep cBPF to BPF translations efficient, since they have optimized JIT code. For native BPF these packet load instructions are less relevant nowadays. BPF_LDX class holds instructions for byte / half-word / word / double-word loads out of memory. Memory in this context is generic and could be stack memory, map value data, packet data, etc.
- BPF_ST, BPF_STX: Both classes are for store operations. Similar to BPF_LDX the BPF_STX is the store counterpart and is used to store the data from a register into memory, which, again, can be stack memory, map value, packet data, etc. BPF_STX also holds special instructions for performing word and double-word based atomic add operations, which can be used for counters, for example. The BPF_ST class is similar to BPF_STX by providing instructions for storing data into memory only that the source operand is an immediate value.
- BPF_ALU, BPF_ALU64: Both classes contain ALU operations. Generally, BPF_ALU operations are in 32 bit mode and BPF_ALU64 in 64 bit mode. Both ALU classes have basic operations with source operand which is register-based and an immediate-based counterpart. Supported by both are add (+), sub (-), and (&), or (|), left shift (<<), right shift (>>), xor (^), mul (*), div (/), mod (%), neg (~) operations. Also mov (<X> := <Y>) was added as a special ALU operation for both classes in both operand modes. BPF_ALU64 also contains a signed right shift. BPF_ALU additionally contains endianness conversion instructions for half-word / word / double-word on a given source register.
- BPF_JMP: This class is dedicated to jump operations. Jumps can be unconditional and conditional. Unconditional jumps simply move the program counter forward, so that the next instruction to be executed relative to the current instruction is off + 1, where off is the constant offset encoded in the instruction. Since off is signed, the jump can also be performed backwards as long as it does not create a loop and is within program bounds. Conditional jumps operate on both, register-based and immediate-based source operands. If the condition in the jump operations results in true, then a relative jump to off + 1 is performed, otherwise the next instruction (0 + 1) is performed. This fall-through jump logic differs compared to cBPF and allows for better branch prediction as it fits the CPU branch predictor logic more naturally. Available conditions are jeq (==), jne (!=), jgt (>), jge (>=), jsgt (signed >), jsge (signed >=), jset (jump if DST & SRC). Apart from that, there are three special jump operations within this class: the exit instruction which will leave the BPF program and return the current value in r0 as a return code, the call instruction, which will issue a function call into one of the available BPF helper functions, and a hidden tail call instruction, which will jump into a different BPF program.

The Linux kernel is shipped with a BPF interpreter which executes programs assembled in BPF instructions. Even cBPF programs are translated into eBPF programs transparently in the kernel, except for architectures that still ship with a cBPF JIT and have not yet migrated to an eBPF JIT.

Currently x86_64, arm64, ppc64, s390x and sparc64 architectures come with an in-kernel eBPF JIT compiler.

All BPF handling such as loading of programs into the kernel or creation of BPF maps is managed through a central bpf() system call. It is also used for managing map entries (lookup / update / delete), and making programs as well as maps persistent in the BPF file system through pinning.

Helper Functions

Helper functions are a concept which enables BPF programs to consult a core kernel defined set of function calls in order to retrieve / push data from / to the kernel. Available helper functions may differ for each BPF program type, for example, BPF programs attached to sockets are only allowed to call into a subset of helpers compared to BPF programs attached to the tc layer. Encapsulation and decapsulation helpers for lightweight tunneling constitute an example of functions which are only available to lower tc layers, whereas event output helpers for pushing notifications to user space are available to tc and XDP programs.

Each helper function is implemented with a commonly shared function signature similar to system calls. The signature is defined as:

```
u64 fn(u64 r1, u64 r2, u64 r3, u64 r4, u64 r5)
```

The calling convention as described in the previous section applies to all BPF helper functions.

The kernel abstracts helper functions into macros `BPF_CALL_0()` to `BPF_CALL_5()` which are similar to those of system calls. The following example is an extract from a helper function which updates map elements by calling into the corresponding map implementation callbacks:

```
BPF_CALL_4(bpf_map_update_elem, struct bpf_map *, map, void *, key,
           void *, value, u64, flags)
{
    WARN_ON_ONCE(!rcu_read_lock_held());
    return map->ops->map_update_elem(map, key, value, flags);
}

const struct bpf_func_proto bpf_map_update_elem_proto = {
    .func           = bpf_map_update_elem,
    .gpl_only       = false,
    .ret_type       = RET_INTEGER,
    .arg1_type      = ARG_CONST_MAP_PTR,
    .arg2_type      = ARG_PTR_TO_MAP_KEY,
    .arg3_type      = ARG_PTR_TO_MAP_VALUE,
    .arg4_type      = ARG_ANYTHING,
};
```

There are various advantages of this approach: while cBPF overloaded its load instructions in order to fetch data at an impossible packet offset to invoke auxiliary helper functions, each cBPF JIT needed to implement support for such a cBPF extension. In case of eBPF, each newly added helper function will be JIT compiled in a transparent and efficient way, meaning that the JIT compiler only needs to emit a call instruction since the register mapping is made in such a way that BPF register assignments already match the underlying architecture's calling convention. This allows for easily extending the core kernel with new helper functionality.

The aforementioned function signature also allows the verifier to perform type checks. The above `struct bpf_func_proto` is used to hand all the necessary information which need to be known about the helper to the verifier, so that the verifier can make sure that the expected types from the helper match the current contents of the BPF program's analyzed registers.

Argument types can range from passing in any kind of value up to restricted contents such as a pointer / size pair for the BPF stack buffer, which the helper should read from or write to. In the latter case, the verifier can also perform additional checks, for example, whether the buffer was previously initialized.

Maps

Maps are efficient key / value stores that reside in kernel space. They can be accessed from a BPF program in order to keep state among multiple BPF program invocations. They can also be accessed through file descriptors from user space and can be arbitrarily shared with other BPF programs or user space applications.

BPF programs which share maps with each other are not required to be of the same program type, for example, tracing programs can share maps with networking programs. A single BPF program can currently access up to 64 different maps directly.

Map implementations are provided by the core kernel. There are generic maps with per-CPU and non-per-CPU flavor that can read / write arbitrary data, but there are also a few non-generic maps that are used along with helper functions.

Generic maps currently available:

- `BPF_MAP_TYPE_HASH`
- `BPF_MAP_TYPE_ARRAY`
- `BPF_MAP_TYPE_PERCPU_HASH`
- `BPF_MAP_TYPE_PERCPU_ARRAY`
- `BPF_MAP_TYPE_LRU_HASH`
- `BPF_MAP_TYPE_LRU_PERCPU_HASH`
- `BPF_MAP_TYPE_LPM_TRIE`

Non-generic maps currently in the kernel:

- `BPF_MAP_TYPE_PROG_ARRAY`
- `BPF_MAP_TYPE_PERF_EVENT_ARRAY`
- `BPF_MAP_TYPE_CGROUP_ARRAY`
- `BPF_MAP_TYPE_STACK_TRACE`
- `BPF_MAP_TYPE_ARRAY_OF_MAPS`
- `BPF_MAP_TYPE_HASH_OF_MAPS`

TODO: further coverage of maps and their purpose

Object Pinning

BPF maps and programs act as a kernel resource and can only be accessed through file descriptors, backed by anonymous inodes in the kernel. Advantages, but also a number of disadvantages come along with them:

User space applications can make use of most file descriptor related APIs, file descriptor passing for Unix domain sockets work transparently, etc, but at the same time, file descriptors are limited to a processes' lifetime, which makes options like map sharing rather cumbersome to carry out.

Thus, it brings a number of complications for certain use cases such as iproute2, where tc or XDP sets up and loads the program into the kernel and terminates itself eventually. With that, also access to maps is unavailable from user space side, where it could otherwise be useful, for example, when maps are shared between ingress and egress locations of the data path. Also, third party applications may wish to monitor or update map contents during BPF program runtime.

To overcome this limitation, a minimal kernel space BPF file system has been implemented, where BPF map and programs can be pinned to, a process called object pinning. The BPF system call has therefore been extended with two new commands which can pin (`BPF_OBJ_PIN`) or retrieve (`BPF_OBJ_GET`) a previously pinned object.

For instance, tools such as tc make use of this infrastructure for sharing maps on ingress and egress. The BPF related file system is not a singleton, it does support multiple mount instances, hard and soft links, etc.

Tail Calls

Another concept that can be used with BPF is called tail calls. Tail calls can be seen as a mechanism that allows one BPF program to call another, without returning back to the old program. Such a call has minimal overhead as unlike function calls, it is implemented as a long jump, reusing the same stack frame.

Such programs are verified independently of each other, thus for transferring state, either per-CPU maps as scratch buffers or in case of tc programs, `skb` fields such as the `cb[]` area must be used.

Only programs of the same type can be tail called, and they also need to match in terms of JIT compilation, thus either JIT compiled or only interpreted programs can be invoked, but not mixed together.

There are two components involved for carrying out tail calls: the first part needs to setup a specialized map called program array (`BPF_MAP_TYPE_PROG_ARRAY`) that can be populated by user space with key / values, where values are the file descriptors of the tail called BPF programs, the second part is a `bpf_tail_call()` helper where the context, a reference to the program array and the lookup key is passed to. Then the kernel inlines this helper call directly into a specialized BPF instruction. Such a program array is currently write-only from user space side.

The kernel looks up the related BPF program from the passed file descriptor and atomically replaces program pointers at the given map slot. When no map entry has been found at the provided key, the kernel will just "fall through" and continue execution of the old program with the instructions following after the `bpf_tail_call()`. Tail calls are a powerful utility, for example, parsing network headers could be structured through tail calls. During runtime, functionality can be added or replaced atomically, and thus altering the BPF program's execution behaviour.

JIT

The 64 bit `x86_64`, `arm64`, `ppc64`, `s390x` and `sparc64` architectures are all shipped with an in-kernel eBPF JIT compiler (`mips64` is work in progress at this time), also all of them are feature equivalent and can be enabled through:

```
# echo 1 > /proc/sys/net/core/bpf_jit_enable
```

The 32 bit `arm`, `mips`, `ppc` and `sparc` architectures currently have a cBPF JIT compiler. The mentioned architectures still having a cBPF JIT as well as all remaining architectures supported by the Linux kernel which do not have a BPF JIT compiler at all need to run eBPF programs through the in-kernel interpreter.

In the kernel's source tree, eBPF JIT support can be easily determined through issuing a grep for `HAVE_EBPF_JIT`:

```
# git grep HAVE_EBPF_JIT arch/
arch/arm64/Kconfig:      select HAVE_EBPF_JIT
arch/powerpc/Kconfig:    select HAVE_EBPF_JIT   if PPC64
arch/s390/Kconfig:       select HAVE_EBPF_JIT   if PACK_STACK && HAVE_MARCH_Z196_
↪FEATURES
arch/sparc/Kconfig:      select HAVE_EBPF_JIT   if SPARC64
arch/x86/Kconfig:        select HAVE_EBPF_JIT   if X86_64
```

Hardening

BPF locks the entire BPF interpreter image (`struct bpf_prog`) as well as the JIT compiled image (`struct bpf_binary_header`) in the kernel as read-only during the program's lifetime in order to prevent the code from potential corruptions. Any corruption happening at that point, for example, due to some kernel bugs will result in a general protection fault and thus crash the kernel instead of allowing the corruption to happen silently.

Architectures that support setting the image memory as read-only can be determined through:

```
$ git grep ARCH_HAS_SET_MEMORY | grep select
arch/arm/Kconfig:    select ARCH_HAS_SET_MEMORY
arch/arm64/Kconfig:  select ARCH_HAS_SET_MEMORY
arch/s390/Kconfig:   select ARCH_HAS_SET_MEMORY
arch/x86/Kconfig:    select ARCH_HAS_SET_MEMORY
```

The option `CONFIG_ARCH_HAS_SET_MEMORY` is not configurable, thanks to which this protection is always built-in. Other architectures might follow in the future.

In case of `/proc/sys/net/core/bpf_jit_harden` set to `1` additional hardening steps for the JIT compilation take effect for unprivileged users. This effectively trades off their performance slightly by decreasing a (potential) attack surface in case of untrusted users operating on the system. The decrease in program execution still results in better performance compared to switching to interpreter entirely.

Currently, enabling hardening will blind all user provided 32 bit and 64 bit constants from the BPF program when it gets JIT compiled in order to prevent JIT spraying attacks which inject native opcodes as immediate values. This is problematic as these immediate values reside in executable kernel memory, therefore a jump that could be triggered from some kernel bug would jump to the start of the immediate value and then execute these as native instructions.

JIT constant blinding prevents this due to randomizing the actual instruction, which means the operation is transformed from an immediate based source operand to a register based one through rewriting the instruction by splitting the actual load of the value into two steps: 1) load of a blinded immediate value `rnd ^ imm` into a register, 2) xoring that register with `rnd` such that the original `imm` immediate then resides in the register and can be used for the actual operation. The example was provided for a load operation, but really all generic operations are blinded.

Example of JITing a program with hardening disabled:

```
# echo 0 > /proc/sys/net/core/bpf_jit_harden

  ffffffffa034f5e9 + <x>:
  [...]
  39:   mov    $0xa8909090,%eax
  3e:   mov    $0xa8909090,%eax
  43:   mov    $0xa8ff3148,%eax
  48:   mov    $0xa89081b4,%eax
  4d:   mov    $0xa8900bb0,%eax
  52:   mov    $0xa810e0c1,%eax
  57:   mov    $0xa8908eb4,%eax
  5c:   mov    $0xa89020b0,%eax
  [...]
```

The same program gets constant blinded when loaded through BPF as an unprivileged user in the case hardening is enabled:

```
# echo 1 > /proc/sys/net/core/bpf_jit_harden

  ffffffffa034f1e5 + <x>:
  [...]
  39:   mov    $0xe1192563,%r10d
  3f:   xor    $0x4989b5f3,%r10d
  46:   mov    %r10d,%eax
  49:   mov    $0xb8296d93,%r10d
  4f:   xor    $0x10b9fd03,%r10d
  56:   mov    %r10d,%eax
  59:   mov    $0x8c381146,%r10d
  5f:   xor    $0x24c7200e,%r10d
  66:   mov    %r10d,%eax
  69:   mov    $0xeb2a830e,%r10d
  6f:   xor    $0x43ba02ba,%r10d
  76:   mov    %r10d,%eax
  79:   mov    $0xd9730af,%r10d
  7f:   xor    $0xa5073b1f,%r10d
  86:   mov    %r10d,%eax
  89:   mov    $0x9a45662b,%r10d
  8f:   xor    $0x325586ea,%r10d
  96:   mov    %r10d,%eax
  [...]
```

Both programs are semantically the same, only that none of the original immediate values are visible anymore in the disassembly of the second program.

At the same time, hardening also disables any JIT kallsyms exposure for privileged users, preventing that JIT image addresses are not exposed to `/proc/kallsyms` anymore.

Offloads

Networking programs in BPF, in particular for tc and XDP do have an offload-interface to hardware in the kernel in order to execute BPF code directly on the NIC.

Currently, the `nfp` driver from Netronome has support for offloading BPF through a JIT compiler which translates BPF instructions to an instruction set implemented against the NIC.

Toolchain

Current user space tooling, introspection facilities and kernel control knobs around BPF are discussed in this section. Note, the tooling and infrastructure around BPF is still rapidly evolving and thus may not provide a complete picture of all available tools.

Development Environment

A step by step guide for setting up a development environment for BPF can be found below for both Fedora and Ubuntu. This will guide you through building, installing and testing a development kernel as well as building and installing iproute2.

The step of building your own iproute2 and Linux kernel is usually not necessary given that major distributions already ship recent enough kernels by default, but would be needed for testing bleeding edge versions or contributing BPF patches to iproute2 and to the Linux kernel, respectively.

Fedora

The following applies to Fedora 25 or later:

```
$ sudo dnf install -y git gcc ncurses-devel elfutils-libelf-devel bc \
  openssl-devel libcap-devel clang llvm
```

Note: If you are running some other Fedora derivative and `dnf` is missing, try using `yum` instead.

Ubuntu

The following applies to Ubuntu 17.04 or later:

```
$ sudo apt-get install -y make gcc libssl-dev bc libelf-dev libcap-dev \
  clang gcc-multilib llvm libncurses5-dev git
```

Compiling the Kernel

Development of new BPF features for the Linux kernel happens inside the `net-next` git tree, latest BPF fixes in the `net` tree. The following command will obtain the kernel source for the `net-next` tree through git:

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/davem/net-next.git
```

If the git commit history is not of interest, then `--depth 1` will clone the tree much faster by truncating the git history only to the most recent commit.

In case the `net` tree is of interest, it can be cloned from this url:

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/davem/net.git
```

There are dozens of tutorials in the Internet on how to build Linux kernels, one good resource is the Kernel Newbies website (https://kernelnewbies.org/KernelBuild) that can be followed with one of the two git trees mentioned above.

Make sure that the generated `.config` file contains the following `CONFIG_*` entries for running BPF. These entries are also needed for Cilium.

```
CONFIG_CGROUP_BPF=y
CONFIG_BPF=y
CONFIG_BPF_SYSCALL=y
CONFIG_NET_SCH_INGRESS=m
CONFIG_NET_CLS_BPF=m
CONFIG_NET_CLS_ACT=y
CONFIG_BPF_JIT=y
CONFIG_LWTUNNEL_BPF=y
CONFIG_HAVE_EBPF_JIT=y
CONFIG_BPF_EVENTS=y
CONFIG_TEST_BPF=m
```

Some of the entries cannot be adjusted through `make menuconfig`. For example, `CONFIG_HAVE_EBPF_JIT` is selected automatically if a given architecture does come with an eBPF JIT. In this specific case, `CONFIG_HAVE_EBPF_JIT` is optional but highly recommended. An architecture not having an eBPF JIT compiler will need to fall back to the in-kernel interpreter with the cost of being less efficient executing BPF instructions.

Verifying the Setup

After you have booted into the newly compiled kernel, navigate to the BPF selftest suite in order to test BPF functionality (current working directory points to the root of the cloned git tree):

```
$ cd tools/testing/selftests/bpf/
$ make
$ sudo ./test_verifier
```

The verifier tests print out all the current checks being performed. The summary at the end of running all tests will dump information of test successes and failures:

```
Summary: 418 PASSED, 0 FAILED
```

In order to run through all BPF selftests, the following command is needed:

```
$ sudo make run_tests
```

If you see any failures, please contact us on Slack with the full test output.

Compiling iproute2

Similar to the `net` (fixes only) and `net-next` (new features) kernel trees, the iproute2 git tree has two branches, namely `master` and `net-next`. The `master` branch is based on the `net` tree and the `net-next` branch is based against the `net-next` kernel tree. This is necessary, so that changes in header files can be synchronized in the iproute2 tree.

In order to clone the iproute2 `master` branch, the following command can be used:

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/shemminger/iproute2.git
```

Similarly, to clone into mentioned `net-next` branch of iproute2, run the following:

```
$ git clone -b net-next git://git.kernel.org/pub/scm/linux/kernel/git/shemminger/
↪iproute2.git
```

After that, proceed with the build and installation:

```
$ cd iproute2/
$ ./configure --prefix=/usr
TC schedulers
 ATM    no

libc has setns: yes
SELinux support: yes
ELF support: yes
libmnl support: no
Berkeley DB: no

docs: latex: no
 WARNING: no docs can be built from LaTeX files
 sgml2html: no
 WARNING: no HTML docs can be built from SGML
$ make
[...]
$ sudo make install
```

Ensure that the `configure` script shows `ELF support:  yes`, so that iproute2 can process ELF files from LLVM's BPF back end. libelf was listed in the instructions for installing the dependencies in case of Fedora and Ubuntu earlier.

LLVM

LLVM is currently the only compiler suite providing a BPF back end. gcc does not support BPF at this point.

The BPF back end was merged into LLVM's 3.7 release. Major distributions enable the BPF back end by default when they package LLVM, therefore installing clang and llvm is sufficient on most recent distributions to start compiling C into BPF object files.

The typical workflow is that BPF programs are written in C, compiled by LLVM into object / ELF files, which are parsed by user space BPF ELF loaders (such as iproute2 or others), and pushed into the kernel through the BPF system call. The kernel verifies the BPF instructions and JITs them, returning a new file descriptor for the program, which then can be attached to a subsystem (e.g. networking). If supported, the subsystem could then further offload the BPF program to hardware (e.g. NIC).

For LLVM, BPF target support can be checked, for example, through the following:

```
$ llc --version
LLVM (http://llvm.org/):
LLVM version 3.8.1
Optimized build.
Default target: x86_64-unknown-linux-gnu
Host CPU: skylake

Registered Targets:
  [...]
  bpf        - BPF (host endian)
  bpfeb      - BPF (big endian)
  bpfel      - BPF (little endian)
  [...]
```

By default, the bpf target uses the endianness of the CPU it compiles on, meaning that if the CPU's endianness is little endian, the program is represented in little endian format as well, and if the CPU's endianness is big endian, the program is represented in big endian. This also matches the runtime behavior of BPF, which is generic and uses the CPU's endianness it runs on in order to not disadvantage architectures in any of the format.

For cross-compilation, the two targets bpfeb and bpfel were introduced, thanks to that BPF programs can be compiled on a node running in one endianness (e.g. little endian on x86) and run on a node in another endianness format (e.g. big endian on arm). Note that the front end (clang) needs to run in the target endianness as well.

Using bpf as a target is the preferred way in situations where no mixture of endianness applies. For example, compilation on x86_64 results in the same output for the targets bpf and bpfel due to being little endian, therefore scripts triggering a compilation also do not have to be endian aware.

A minimal, stand-alone XDP drop program might look like the following example (xdp-example.c):

```
#include <linux/bpf.h>

#ifndef __section
# define __section(NAME)                  \
   __attribute__((section(NAME), used))
#endif

__section("prog")
int xdp_drop(struct xdp_md *ctx)
{
    return XDP_DROP;
}

char __license[] __section("license") = "GPL";
```

It can then be compiled and loaded into the kernel as follows:

```
$ clang -O2 -Wall -target bpf -c xdp-example.c -o xdp-example.o
# ip link set dev em1 xdp obj xdp-example.o
```

For the generated object file LLVM (>= 3.9) uses the official BPF machine value, that is, EM_BPF (decimal: 247 / hex: 0xf7). In this example, the program has been compiled with bpf target under x86_64, therefore LSB (as opposed to MSB) is shown regarding endianness:

```
$ file xdp-example.o
xdp-example.o: ELF 64-bit LSB relocatable, *unknown arch 0xf7* version 1 (SYSV), not␣
↪stripped
```

`readelf -a xdp-example.o` will dump further information about the ELF file, which can sometimes be useful for introspecting generated section headers, relocation entries and the symbol table.

In the unlikely case where clang and LLVM need to be compiled from scratch, the following commands can be used:

```
$ git clone http://llvm.org/git/llvm.git
$ cd llvm/tools
$ git clone --depth 1 http://llvm.org/git/clang.git
$ cd ..; mkdir build; cd build
$ cmake .. -DLLVM_TARGETS_TO_BUILD="BPF;X86" -DBUILD_SHARED_LIBS=OFF -DCMAKE_BUILD_
↪TYPE=Release -DLLVM_BUILD_RUNTIME=OFF
$ make -j $(getconf _NPROCESSORS_ONLN)

$ ./bin/llc --version
LLVM (http://llvm.org/):
LLVM version x.y.zsvn
Optimized build.
Default target: x86_64-unknown-linux-gnu
Host CPU: skylake

Registered Targets:
  bpf    - BPF (host endian)
  bpfeb  - BPF (big endian)
  bpfel  - BPF (little endian)
  x86    - 32-bit X86: Pentium-Pro and above
  x86-64 - 64-bit X86: EM64T and AMD64

$ export PATH=$PWD/bin:$PATH   # add to ~/.bashrc
```

Make sure that `--version` mentions `Optimized build.`, otherwise the compilation time for programs when having LLVM in debugging mode will significantly increase (e.g. by 10x or more).

For debugging, clang can generate the assembler output as follows:

```
$ clang -O2 -S -Wall -target bpf -c xdp-example.c -o xdp-example.S
$ cat xdp-example.S
    .text
    .section    prog,"ax",@progbits
    .globl      xdp_drop
    .p2align    3
xdp_drop:                             # @xdp_drop
# BB#0:
    r0 = 1
    exit

    .section    license,"aw",@progbits
    .globl    __license               # @__license
__license:
    .asciz    "GPL"
```

Furthermore, more recent LLVM versions (>= 4.0) can also store debugging information in dwarf format into the object file. This can be done through the usual workflow by adding `-g` for compilation.

```
$ clang -O2 -g -Wall -target bpf -c xdp-example.c -o xdp-example.o
$ llvm-objdump -S -no-show-raw-insn xdp-example.o

xdp-example.o:        file format ELF64-BPF

Disassembly of section prog:
xdp_drop:
; {
    0:        r0 = 1
; return XDP_DROP;
    1:        exit
```

The `llvm-objdump` tool can then annotate the assembler output with the original C code used in the compilation. The trivial example in this case does not contain much C code, however, the line numbers shown as `0:` and `1:` correspond directly to the kernel's verifier log.

This means that in case BPF programs get rejected by the verifier, `llvm-objdump` can help to correlate the instructions back to the original C code, which is highly useful for analysis.

```
# ip link set dev em1 xdp obj xdp-example.o verb

Prog section 'prog' loaded (5)!
 - Type:         6
 - Instructions: 2 (0 over limit)
 - License:      GPL

Verifier analysis:

0: (b7) r0 = 1
1: (95) exit
processed 2 insns
```

As it can be seen in the verifier analysis, the `llvm-objdump` output dumps the same BPF assembler code as the kernel.

Leaving out the `-no-show-raw-insn` option will also dump the raw `struct bpf_insn` as hex in front of the assembly:

```
$ llvm-objdump -S xdp-example.o

xdp-example.o:        file format ELF64-BPF

Disassembly of section prog:
xdp_drop:
; {
   0:       b7 00 00 00 01 00 00 00     r0 = 1
; return foo();
   1:       95 00 00 00 00 00 00 00     exit
```

For LLVM IR debugging, the compilation process for BPF can be split into two steps, generating a binary LLVM IR intermediate file `xdp-example.bc`, which can later on be passed to llc:

```
$ clang -O2 -Wall -emit-llvm -c xdp-example.c -o xdp-example.bc
$ llc xdp-example.bc -march=bpf -filetype=obj -o xdp-example.o
```

The generated LLVM IR can also be dumped in human readable format through:

```
$ clang -O2 -Wall -emit-llvm -S -c xdp-example.c -o -
```

Note that LLVM's BPF back end currently does not support generating code that makes use of BPF's 32 bit subregisters. Inline assembly for BPF is currently unsupported, too.

Furthermore, compilation from BPF assembly (e.g. `llvm-mc xdp-example.S -arch bpf -filetype=obj -o xdp-example.o`) is currently not supported either due to missing BPF assembly parser.

When writing C programs for BPF, there are a couple of pitfalls to be aware of, compared to usual application development with C. The following items describe some of the differences for the BPF model:

1. **Everything needs to be inlined, there are no function or shared library calls available.**

   Shared libraries, etc cannot be used with BPF. However, common library code used in BPF programs can be placed into header files and included in the main programs. For example, Cilium makes heavy use of it (see `bpf/lib/`). However, this still allows for including header files, for example, from the kernel or other libraries and reuse their static inline functions or macros / definitions.

   Eventually LLVM needs to compile the entire code into a flat sequence of BPF instructions for a given program section. Best practice is to use an annotation like `__inline` for every library function as shown below. The use of `always_inline` is recommended, since the compiler could still decide to uninline large functions that are only annotated as `inline`.

   In case the latter happens, LLVM will generate a relocation entry into the ELF file, which BPF ELF loaders such as iproute2 cannot resolve and will thus produce an error since only BPF maps are valid relocation entries which loaders can process.

   ```
   #include <linux/bpf.h>

   #ifndef __section
   # define __section(NAME)                  \
      __attribute__((section(NAME), used))
   #endif

   #ifndef __inline
   # define __inline                         \
      inline __attribute__((always_inline))
   #endif

   static __inline int foo(void)
   {
       return XDP_DROP;
   }

   __section("prog")
   int xdp_drop(struct xdp_md *ctx)
   {
       return foo();
   }

   char __license[] __section("license") = "GPL";
   ```

2. **Multiple programs can reside inside a single C file in different sections.**

   C programs for BPF make heavy use of section annotations. A C file is typically structured into 3 or more sections. BPF ELF loaders use these names to extract and prepare the relevant information in order to load the programs and maps through the bpf system call. For example, iproute2 uses `maps` and `license` as default section name to find metadata needed for map creation and the license for the BPF program, respectively.

On program creation time the latter is pushed into the kernel as well, and enables some of the helper functions which are exposed as GPL only in case the program also holds a GPL compatible license, for example `bpf_ktime_get_ns()`, `bpf_probe_read()` and others.

The remaining section names are specific for BPF program code, for example, the below code has been modified to contain two program sections, `ingress` and `egress`. The toy example code demonstrates that both can share a map and common static inline helpers such as the `account_data()` function.

The `xdp-example.c` example has been modified to a `tc-example.c` example that can be loaded with tc and attached to a netdevice's ingress and egress hook. It accounts the transferred bytes into a map called `acc_map`, which has two map slots, one for traffic accounted on the ingress hook, one on the egress hook.

```
#include <linux/bpf.h>
#include <linux/pkt_cls.h>
#include <stdint.h>
#include <iproute2/bpf_elf.h>

#ifndef __section
# define __section(NAME)                  \
   __attribute__((section(NAME), used))
#endif

#ifndef __inline
# define __inline                         \
   inline __attribute__((always_inline))
#endif

#ifndef lock_xadd
# define lock_xadd(ptr, val)              \
   ((void)__sync_fetch_and_add(ptr, val))
#endif

#ifndef BPF_FUNC
# define BPF_FUNC(NAME, ...)              \
   (*NAME)(__VA_ARGS__) = (void *)BPF_FUNC_##NAME
#endif

static void *BPF_FUNC(map_lookup_elem, void *map, const void *key);

struct bpf_elf_map acc_map __section("maps") = {
    .type           = BPF_MAP_TYPE_ARRAY,
    .size_key       = sizeof(uint32_t),
    .size_value     = sizeof(uint32_t),
    .pinning        = PIN_GLOBAL_NS,
    .max_elem       = 2,
};

static __inline int account_data(struct __sk_buff *skb, uint32_t dir)
{
    uint32_t *bytes;

    bytes = map_lookup_elem(&acc_map, &dir);
    if (bytes)
            lock_xadd(bytes, skb->len);

    return TC_ACT_OK;
}

__section("ingress")
```

```
int tc_ingress(struct __sk_buff *skb)
{
    return account_data(skb, 0);
}

__section("egress")
int tc_egress(struct __sk_buff *skb)
{
    return account_data(skb, 1);
}

char __license[] __section("license") = "GPL";
```

The example also demonstrates a couple of other things which are useful to be aware of when developing programs. The code includes kernel headers, standard C headers and an iproute2 specific header containing the definition of `struct bpf_elf_map`. iproute2 has a common BPF ELF loader and as such the definition of `struct bpf_elf_map` is the very same for XDP and tc typed programs.

A `struct bpf_elf_map` entry defines a map in the program and contains all relevant information (such as key / value size, etc) needed to generate a map which is used from the two BPF programs. The structure must be placed into the `maps` section, so that the loader can find it. There can be multiple map declarations of this type with different variable names, but all must be annotated with `__section("maps")`.

The `struct bpf_elf_map` is specific to iproute2. Different BPF ELF loaders can have different formats, for example, the libbpf in the kernel source tree, which is mainly used by `perf`, has a different specification. iproute2 guarantees backwards compatibility for `struct bpf_elf_map`. Cilium follows the iproute2 model.

The example also demonstrates how BPF helper functions are mapped into the C code and being used. Here, `map_lookup_elem()` is defined by mapping this function into the `BPF_FUNC_map_lookup_elem` enum value which is exposed as a helper in `uapi/linux/bpf.h`. When the program is later loaded into the kernel, the verifier checks whether the passed arguments are of the expected type and re-points the helper call into a real function call. Moreover, `map_lookup_elem()` also demonstrates how maps can be passed to BPF helper functions. Here, `&acc_map` from the `maps` section is passed as the first argument to `map_lookup_elem()`.

Since the defined array map is global, the accounting needs to use an atomic operation, which is defined as `lock_xadd()`. LLVM maps `__sync_fetch_and_add()` as a built-in function to the BPF atomic add instruction, that is, `BPF_STX | BPF_XADD | BPF_W` for word sizes.

Last but not least, the `struct bpf_elf_map` tells that the map is to be pinned as `PIN_GLOBAL_NS`. This means that tc will pin the map into the BPF pseudo file system as a node. By default, it will be pinned to `/sys/fs/bpf/tc/globals/acc_map` for the given example. Due to the `PIN_GLOBAL_NS`, the map will be placed under `/sys/fs/bpf/tc/globals/`. `globals` acts as a global namespace that spans across object files. If the example used `PIN_OBJECT_NS`, then tc would create a directory that is local to the object file. For example, different C files with BPF code could have the same `acc_map` definition as above with a `PIN_GLOBAL_NS` pinning. In that case, the map will be shared among BPF programs originating from various object files. `PIN_NONE` would mean that the map is not placed into the BPF file system as a node, and as a result will not be accessible from user space after tc quits. It would also mean that tc creates two separate map instances for each program, since it cannot retrieve a previously pinned map under that name. The `acc_map` part from the mentioned path is the name of the map as specified in the source code.

Thus, upon loading of the `ingress` program, tc will find that no such map exists in the BPF file system and creates a new one. On success, the map will also be pinned, so that when the `egress` program is loaded through tc, it will find that such map already exists in the BPF file system and will reuse that for the egress program. The loader also makes sure in case maps exist with the same name that also their properties (key / value size, etc) match.

Just like tc can retrieve the same map, also third party applications can use the BPF_OBJ_GET command from the bpf system call in order to create a new file descriptor pointing to the same map instance, which can then be used to lookup / update / delete map elements.

The code can be compiled and loaded via iproute2 as follows:

```
$ clang -O2 -Wall -target bpf -c tc-example.c -o tc-example.o

# tc qdisc add dev em1 clsact
# tc filter add dev em1 ingress bpf da obj tc-example.o sec ingress
# tc filter add dev em1 egress bpf da obj tc-example.o sec egress

# tc filter show dev em1 ingress
filter protocol all pref 49152 bpf
filter protocol all pref 49152 bpf handle 0x1 tc-example.o:[ingress] direct-
↪action tag c5f7825e5dac396f

# tc filter show dev em1 egress
filter protocol all pref 49152 bpf
filter protocol all pref 49152 bpf handle 0x1 tc-example.o:[egress] direct-
↪action tag b2fd5adc0f262714

# mount | grep bpf
sysfs on /sys/fs/bpf type sysfs (rw,nosuid,nodev,noexec,relatime,seclabel)
bpf on /sys/fs/bpf type bpf (rw,relatime,mode=0700)

# tree /sys/fs/bpf/
/sys/fs/bpf/
+-- ip -> /sys/fs/bpf/tc/
+-- tc
|   +-- globals
|       +-- acc_map
+-- xdp -> /sys/fs/bpf/tc/

4 directories, 1 file
```

As soon as packets pass the em1 device, counters from the BPF map will be increased.

3. **There are no global variables allowed.**

   For the reasons already mentioned in point 1, BPF cannot have global variables as often used in normal C programs.

   However, there is a work-around in that the program can simply use a BPF map of type BPF_MAP_TYPE_PERCPU_ARRAY with just a single slot of arbitrary value size. This works, because during execution, BPF programs are guaranteed to never get preempted by the kernel and therefore can use the single map entry as a scratch buffer for temporary data, for example, to extend beyond the stack limitation. This also functions across tail calls, since it has the same guarantees with regards to preemption.

   Otherwise, for holding state across multiple BPF program runs, normal BPF maps can be used.

4. **There are no const strings or arrays allowed.**

   Defining const strings or other arrays in the BPF C program does not work for the same reasons as pointed out in sections 1 and 3, which is, that relocation entries will be generated in the ELF file which will be rejected by loaders due to not being part of the ABI towards loaders (loaders also cannot fix up such entries as it would require large rewrites of the already compiled BPF sequence).

In the future, LLVM might detect these occurrences and early throw an error to the user.

Helper functions such as `trace_printk()` can be worked around as follows:

```
static void BPF_FUNC(trace_printk, const char *fmt, int fmt_size, ...);

#ifndef printk
# define printk(fmt, ...)                                      \
    ({                                                         \
        char ____fmt[] = fmt;                                  \
        trace_printk(____fmt, sizeof(____fmt), ##__VA_ARGS__); \
    })
#endif
```

The program can then use the macro naturally like `printk("skb len:%u\n", skb->len);`. The output will then be written to the trace pipe. `tc exec bpf dbg` can be used to retrieve the messages from there.

The use of the `trace_printk()` helper function has a couple of disadvantages and thus is not recommended for production usage. Constant strings like the `"skb len:%u\n"` need to be loaded into the BPF stack each time the helper function is called, but also BPF helper functions are limited to a maximum of 5 arguments. This leaves room for only 3 additional variables which can be passed for dumping.

Therefore, despite being helpful for quick debugging, it is recommended (for networking programs) to use the `skb_event_output()` or the `xdp_event_output()` helper, respectively. They allow for passing custom structs from the BPF program to the perf event ring buffer along with an optional packet sample. For example, Cilium's monitor makes use of these helpers in order to implement a debugging framework, notifications for network policy violations, etc. These helpers pass the data through a lockless memory mapped per-CPU `perf` ring buffer, and is thus significantly faster than `trace_printk()`.

5. **Use of LLVM built-in functions for memset()/memcpy()/memmove()/memcmp().**

   Since BPF programs cannot perform any function calls other than those to BPF helpers, common library code needs to be implemented as inline functions. In addition, also LLVM provides some built-ins that the programs can use for constant sizes (here: `n`) which will then always get inlined:

   ```
   #ifndef memset
   # define memset(dest, chr, n)   __builtin_memset((dest), (chr), (n))
   #endif

   #ifndef memcpy
   # define memcpy(dest, src, n)   __builtin_memcpy((dest), (src), (n))
   #endif

   #ifndef memmove
   # define memmove(dest, src, n)  __builtin_memmove((dest), (src), (n))
   #endif
   ```

   The `memcmp()` built-in had some corner cases where inlining did not take place due to an LLVM issue in the back end, and is therefore not recommended to be used until the issue is fixed.

6. **There are no loops available.**

   The BPF verifier in the kernel checks that a BPF program does not contain loops by performing a depth first search of all possible program paths besides other control flow graph validations. The purpose is to make sure that the program is always guaranteed to terminate.

   A very limited form of looping is available for constant upper loop bounds by using `#pragma unroll` directive. Example code that is compiled to BPF:

```
#pragma unroll
    for (i = 0; i < IPV6_MAX_HEADERS; i++) {
        switch (nh) {
        case NEXTHDR_NONE:
            return DROP_INVALID_EXTHDR;
        case NEXTHDR_FRAGMENT:
            return DROP_FRAG_NOSUPPORT;
        case NEXTHDR_HOP:
        case NEXTHDR_ROUTING:
        case NEXTHDR_AUTH:
        case NEXTHDR_DEST:
            if (skb_load_bytes(skb, l3_off + len, &opthdr, sizeof(opthdr)) <␣
↪0)
                return DROP_INVALID;

            nh = opthdr.nexthdr;
            if (nh == NEXTHDR_AUTH)
                len += ipv6_authlen(&opthdr);
            else
                len += ipv6_optlen(&opthdr);
            break;
        default:
            *nexthdr = nh;
            return len;
        }
    }
```

Another possibility is to use tail calls by calling into the same program again and using a `BPF_MAP_TYPE_PERCPU_ARRAY` map for having a local scratch space. While being dynamic, this form of looping however is limited to a maximum of 32 iterations.

In the future, BPF may have some native, but limited form of implementing loops.

7. **Partitioning programs with tail calls.**

   Tail calls provide the flexibility to atomically alter program behavior during runtime by jumping from one BPF program into another. In order to select the next program, tail calls make use of program array maps (`BPF_MAP_TYPE_PROG_ARRAY`), and pass the map as well as the index to the next program to jump to. There is no return to the old program after the jump has been performed, and in case there was no program present at the given map index, then execution continues on the original program.

   For example, this can be used to implement various stages of a parser, where such stages could be updated with new parsing features during runtime.

   Another use case are event notifications, for example, Cilium can opt in packet drop notifications during runtime, where the `skb_event_output()` call is located inside the tail called program. Thus, during normal operations, the fall-through path will always be executed unless a program is added to the related map index, where the program then prepares the metadata and triggers the event notification to a user space daemon.

   Program array maps are quite flexible, enabling also individual actions to be implemented for programs located in each map index. For example, the root program attached to XDP or tc could perform an initial tail call to index 0 of the program array map, performing traffic sampling, then jumping to index 1 of the program array map, where firewalling policy is applied and the packet either dropped or further processed in index 2 of the program array map, where it is mangled and sent out of an interface again. Jumps in the program array map can, of course, be arbitrary. The kernel will eventually execute the fall-through path when the maximum tail call limit has been reached.

   Minimal example extract of using tail calls:

```
[...]

#ifndef __stringify
# define __stringify(X)   #X
#endif

#ifndef __section
# define __section(NAME)                  \
   __attribute__((section(NAME), used))
#endif

#ifndef __section_tail
# define __section_tail(ID, KEY)          \
   __section(__stringify(ID) "/" __stringify(KEY))
#endif

#ifndef BPF_FUNC
# define BPF_FUNC(NAME, ...)              \
   (*NAME)(__VA_ARGS__) = (void *)BPF_FUNC_##NAME
#endif

#define BPF_JMP_MAP_ID   1

static void BPF_FUNC(tail_call, struct __sk_buff *skb, void *map,
                     uint32_t index);

struct bpf_elf_map jmp_map __section("maps") = {
    .type           = BPF_MAP_TYPE_PROG_ARRAY,
    .id             = BPF_JMP_MAP_ID,
    .size_key       = sizeof(uint32_t),
    .size_value     = sizeof(uint32_t),
    .pinning        = PIN_GLOBAL_NS,
    .max_elem       = 1,
};

__section_tail(JMP_MAP_ID, 0)
int looper(struct __sk_buff *skb)
{
    printk("skb cb: %u\n", skb->cb[0]++);
    tail_call(skb, &jmp_map, 0);
    return TC_ACT_OK;
}

__section("prog")
int entry(struct __sk_buff *skb)
{
    skb->cb[0] = 0;
    tail_call(skb, &jmp_map, 0);
    return TC_ACT_OK;
}

char __license[] __section("license") = "GPL";
```

When loading this toy program, tc will create the program array and pin it to the BPF file system in the global namespace under `jmp_map`. Also, the BPF ELF loader in iproute2 will also recognize sections that are marked as `__section_tail()`. The provided `id` in `struct bpf_elf_map` will be matched against the id marker in the `__section_tail()`, that is, `JMP_MAP_ID`, and the program therefore loaded at the user specified program array map index, which is `0` in this example. As a result, all provided tail call sections will be populated by the iproute2 loader to the corresponding maps. This mechanism is not specific to tc, but can be applied with any other BPF program type that iproute2 supports (such as XDP, lwt).

The pinned map can be retrieved by a user space applications (e.g. Cilium daemon), but also by tc itself in order to update the map with new programs. Updates happen atomically, the initial entry programs that are triggered first from the various subsystems are also updated atomically.

Example for tc to perform tail call map updates:

```
# tc exec bpf graft m:globals/jmp_map key 0 obj new.o sec foo
```

In case iproute2 would update the pinned program array, the `graft` command can be used. By pointing it to `globals/jmp_map`, tc will update the map at index / key `0` with a new program residing in the object file `new.o` under section `foo`.

8. **Limited stack space of 512 bytes.**

   Stack space in BPF programs is limited to only 512 bytes, which needs to be taken into careful consideration when implementing BPF programs in C. However, as mentioned earlier in point 3, a `BPF_MAP_TYPE_PERCPU_ARRAY` map with a single entry can be used in order to enlarge scratch buffer space.

iproute2

There are various front ends for loading BPF programs into the kernel such as bcc, perf, iproute2 and others. The Linux kernel source tree also provides a user space library under `tools/lib/bpf/`, which is mainly used and driven by perf for loading BPF tracing programs into the kernel. However, the library itself is generic and not limited to perf only. bcc is a toolkit providing many useful BPF programs mainly for tracing that are loaded ad-hoc through a Python interface embedding the BPF C code. Syntax and semantics for implementing BPF programs slightly differ among front ends in general, though. Additionally, there are also BPF samples in the kernel source tree (`samples/bpf/`) which parse the generated object files and load the code directly through the system call interface.

This and previous sections mainly focus on the iproute2 suite's BPF front end for loading networking programs of XDP, tc or lwt type, since Cilium's programs are implemented against this BPF loader. In future, Cilium will be equipped with a native BPF loader, but programs will still be compatible to be loaded through iproute2 suite in order to facilitate development and debugging.

All BPF program types supported by iproute2 share the same BPF loader logic due to having a common loader back end implemented as a library (`lib/bpf.c` in iproute2 source tree).

The previous section on LLVM also covered some iproute2 parts related to writing BPF C programs, and later sections in this document are related to tc and XDP specific aspects when writing programs. Therefore, this section will rather focus on usage examples for loading object files with iproute2 as well as some of the generic mechanics of the loader. It does not try to provide a complete coverage of all details, but enough for getting started.

1. Loading of XDP BPF object files.

Given a BPF object file `prog.o` has been compiled for XDP, it can be loaded through `ip` to a XDP-supported netdevice called `em1` with the following command:

```
# ip link set dev em1 xdp obj prog.o
```

The above command assumes that the program code resides in the default section which is called `prog` in XDP case. Should this not be the case, and the section is named differently, for example, `foobar`, then the program needs to be loaded as:

```
# ip link set dev em1 xdp obj prog.o sec foobar
```

By default, ip will throw an error in case a XDP program is already attached to the networking interface, to prevent it from being overridden by accident. In order to replace the currently running XDP program with a new one, the -force option must be used:

```
# ip -force link set dev em1 xdp obj prog.o
```

Most XDP-enabled drivers today support an atomic replacement of the existing program with a new one without traffic interruption. There is always only a single program attached to an XDP-enabled driver due to performance reasons, hence a chain of programs is not supported. However, as described in the previous section, partitioning of programs can be performed through tail calls to achieve a similar use case when necessary.

The ip link command will display an xdp flag if the interface has an XDP program attached. ip link | grep xdp can thus be used to find all interfaces that have XDP running. Further introspection facilities will be provided through the detailed view with ip -d link once the kernel API gains support for dumping additional attributes.

In order to remove the existing XDP program from the interface, the following command must be issued:

```
# ip link set dev em1 xdp off
```

2. Loading of tc BPF object files.

Given a BPF object file prog.o has been compiled for tc, it can be loaded through the tc command to a netdevice. Unlike XDP, there is no driver dependency for supporting attaching BPF programs to the device. Here, the netdevice is called em1, and with the following command the program can be attached to the networking ingress path of em1:

```
# tc qdisc add dev em1 clsact
# tc filter add dev em1 ingress bpf da obj prog.o
```

The first step is to set up a clsact qdisc (Linux queueing discipline). clsact is a dummy qdisc similar to the ingress qdisc, which can only hold classifier and actions, but does not perform actual queueing. It is needed in order to attach the bpf classifier. The clsact qdisc provides two special hooks called ingress and egress, where the classifier can be attached to. Both ingress and egress hooks are located in central receive and transmit locations in the networking data path, where every packet on the device passes through. The ingress hook is called from __netif_receive_skb_core() -> sch_handle_ingress() in the kernel and the egress hook from __dev_queue_xmit() -> sch_handle_egress().

The equivalent for attaching the program to the egress hook looks as follows:

```
# tc filter add dev em1 egress bpf da obj prog.o
```

The clsact qdisc is processed lockless from ingress and egress direction and can also be attached to virtual, queue-less devices such as veth devices connecting containers.

Next to the hook, the tc filter command selects bpf to be used in da (direct-action) mode. da mode is recommended and should always be specified. It basically means that the bpf classifier does not need to call into external tc action modules, which are not necessary for bpf anyway, since all packet mangling, forwarding or other kind of actions can already be performed inside the single BPF program which is to be attached, and is therefore significantly faster.

At this point, the program has been attached and is executed once packets traverse the device. Like in XDP, should the default section name not be used, then it can be specified during load, for example, in case of section foobar:

```
# tc filter add dev em1 egress bpf da obj prog.o sec foobar
```

iproute2's BPF loader allows for using the same command line syntax across program types, hence the `obj prog.o sec foobar` is the same syntax as with XDP mentioned earlier.

The attached programs can be listed through the following commands:

```
# tc filter show dev em1 ingress
filter protocol all pref 49152 bpf
filter protocol all pref 49152 bpf handle 0x1 prog.o:[ingress] direct-action␣
↪tag c5f7825e5dac396f

# tc filter show dev em1 egress
filter protocol all pref 49152 bpf
filter protocol all pref 49152 bpf handle 0x1 prog.o:[egress] direct-action␣
↪tag b2fd5adc0f262714
```

The output of `prog.o:[ingress]` tells that program section `ingress` was loaded from the file `prog.o`, and `bpf` operates in `direct-action` mode. The program tags are appended for each, which denotes a hash over the instruction stream which can be used for debugging / introspection.

tc can attach more than just a single BPF program, it provides various other classifiers which can be chained together. However, attaching a single BPF program is fully sufficient since all packet operations can be contained in the program itself thanks to `da` (`direct-action`) mode. For optimal performance and flexibility, this is the recommended usage.

In the above `show` command, tc also displays `pref 49152` and `handle 0x1` next to the BPF related output. Both are auto-generated in case they are not explicitly provided through the command line. `pref` denotes a priority number, which means that in case multiple classifiers are attached, they will be executed based on ascending priority, and `handle` represents an identifier in case multiple instances of the same classifier have been loaded under the same `pref`. Since in case of BPF, a single program is fully sufficient, `pref` and `handle` can typically be ignored.

Only in the case where it is planned to atomically replace the attached BPF programs, it would be recommended to explicitly specify `pref` and `handle` a priori on initial load, so that they do not have to be queried at a later point in time for the `replace` operation. Thus, creation becomes:

```
# tc filter add dev em1 ingress pref 1 handle 1 bpf da obj prog.o sec foobar

# tc filter show dev em1 ingress
filter protocol all pref 1 bpf
filter protocol all pref 1 bpf handle 0x1 prog.o:[foobar] direct-action tag␣
↪c5f7825e5dac396f
```

And for the atomic replacement, the following can be issued for updating the existing program at `ingress` hook with the new BPF program from the file `prog.o` in section `foobar`:

```
# tc filter replace dev em1 ingress pref 1 handle 1 bpf da obj prog.o sec␣
↪foobar
```

Last but not least, in order to remove all attached programs from the `ingress` respectively `egress` hook, the following can be used:

```
# tc filter del dev em1 ingress
# tc filter del dev em1 egress
```

For removing the entire `clsact` qdisc from the netdevice, which implicitly also removes all attached programs from the `ingress` and `egress` hooks, the below command is provided:

```
# tc qdisc del dev em1 clsact
```

These two workflows are the basic operations to load XDP BPF respectively tc BPF programs with iproute2.

There are other various advanced options for the BPF loader that apply both to XDP and tc, some of them are listed here. In the examples only XDP is presented for simplicity.

1. Verbose log output even on success.

The option `verb` can be appended for loading programs in order to dump the verifier log, even if no error occurred:

```
# ip link set dev em1 xdp obj xdp-example.o verb

Prog section 'prog' loaded (5)!
 - Type:         6
 - Instructions: 2 (0 over limit)
 - License:      GPL

Verifier analysis:

0: (b7) r0 = 1
1: (95) exit
processed 2 insns
```

2. Load program that is already pinned in BPF file system.

Instead of loading a program from an object file, iproute2 can also retrieve the program from the BPF file system in case some external entity pinned it there and attach it to the device:

```
# ip link set dev em1 xdp pinned /sys/fs/bpf/prog
```

iproute2 can also use the short form that is relative to the detected mount point of the BPF file system:

```
# ip link set dev em1 xdp pinned m:prog
```

When loading BPF programs, iproute2 will automatically detect the mounted file system instance in order to perform pinning of nodes. In case no mounted BPF file system instance was found, then tc will automatically mount it to the default location under `/sys/fs/bpf/`.

In case an instance has already been found, then it will be used and no additional mount will be performed:

```
# mkdir /var/run/bpf
# mount --bind /var/run/bpf /var/run/bpf
# mount -t bpf bpf /var/run/bpf
# tc filter add dev em1 ingress bpf da obj tc-example.o sec prog
# tree /var/run/bpf
/var/run/bpf
+-- ip -> /run/bpf/tc/
+-- tc
|   +-- globals
|       +-- jmp_map
+-- xdp -> /run/bpf/tc/

4 directories, 1 file
```

By default tc will create an initial directory structure as shown above, where all subsystem users will point to the same location through symbolic links for the `globals` namespace, so that pinned BPF maps can be reused among various BPF program types in iproute2. In case the file system instance has already been mounted and an existing structure already exists, then tc will not override it. This could be the case for separating lwt, tc and xdp maps in order to not share globals among all.

As briefly covered in the previous LLVM section, iproute2 will install a header file upon installation which can be included through the standard include path by BPF programs:

```
#include <iproute2/bpf_elf.h>
```

The purpose of this header file is to provide an API for maps and default section names used by programs. It's a stable contract between iproute2 and BPF programs.

The map definition for iproute2 is struct bpf_elf_map. Its members have been covered earlier in the LLVM section of this document.

When parsing the BPF object file, the iproute2 loader will walk through all ELF sections. It initially fetches ancillary sections like maps and license. For maps, the struct bpf_elf_map array will be checked for validity and whenever needed, compatibility workarounds are performed. Subsequently all maps are created with the user provided information, either retrieved as a pinned object, or newly created and then pinned into the BPF file system. Next the loader will handle all program sections that contain ELF relocation entries for maps, meaning that BPF instructions loading map file descriptors into registers are rewritten so that the corresponding map file descriptors are encoded into the instructions immediate value, in order for the kernel to be able to convert them later on into map kernel pointers. After that all the programs themselves are created through the BPF system call, and tail called maps, if present, updated with the program's file descriptors.

BPF sysctls

The Linux kernel provides few sysctls that are BPF related and covered in this section.

- /proc/sys/net/core/bpf_jit_enable: Enables or disables the BPF JIT compiler.

Value	Description
0	Disable the JIT and use only interpreter (kernel's default value)
1	Enable the JIT compiler
2	Enable the JIT and emit debugging traces to the kernel log

  As described in subsequent sections, bpf_jit_disasm tool can be used to process debugging traces when the JIT compiler is set to debugging mode (option 2).

- /proc/sys/net/core/bpf_jit_harden: Enables or disables BPF JIT hardening. Note that enabling hardening trades off performance, but can mitigate JIT spraying by blinding out the BPF program's immediate values. For programs processed through the interpreter, blinding of immediate values is not needed / performed.

Value	Description
0	Disable JIT hardening (kernel's default value)
1	Enable JIT hardening for unprivileged users only
2	Enable JIT hardening for all users

- /proc/sys/net/core/bpf_jit_kallsyms: Enables or disables export of JITed programs as kernel symbols to /proc/kallsyms so that they can be used together with perf tooling as well as making these addresses aware to the kernel for stack unwinding, for example, used in dumping stack traces. The symbol names contain the BPF program tag (bpf_prog_<tag>). If bpf_jit_harden is enabled, then this feature is disabled.

Value	Description
0	Disable JIT kallsyms export (kernel's default value)
1	Enable JIT kallsyms export for privileged users only

Kernel Testing

The Linux kernel ships a BPF selftest suite, which can be found in the kernel source tree under `tools/testing/selftests/bpf/`.

```
$ cd tools/testing/selftests/bpf/
$ make
# make run_tests
```

The test suite contains test cases against the BPF verifier, program tags, various tests against the BPF map interface and map types. It contains various runtime tests from C code for checking LLVM back end, and eBPF as well as cBPF asm code that is run in the kernel for testing the interpreter and JITs.

JIT Debugging

For JIT developers performing audits or writing extensions, each compile run can output the generated JIT image into the kernel log through:

```
# echo 2 > /proc/sys/net/core/bpf_jit_enable
```

Whenever a new BPF program is loaded, the JIT compiler will dump the output, which can then be inspected with `dmesg`, for example:

```
[ 3389.935842] flen=6 proglen=70 pass=3 image=ffffffffa0069c8f from=tcpdump pid=20583
[ 3389.935847] JIT code: 00000000: 55 48 89 e5 48 83 ec 60 48 89 5d f8 44 8b 4f 68
[ 3389.935849] JIT code: 00000010: 44 2b 4f 6c 4c 8b 87 d8 00 00 00 be 0c 00 00 00
[ 3389.935850] JIT code: 00000020: e8 1d 94 ff e0 3d 00 08 00 00 75 16 be 17 00 00
[ 3389.935851] JIT code: 00000030: 00 e8 28 94 ff e0 83 f8 01 75 07 b8 ff ff 00 00
[ 3389.935852] JIT code: 00000040: eb 02 31 c0 c9 c3
```

`flen` is the length of the BPF program (here, 6 BPF instructions), and `proglen` tells the number of bytes generated by the JIT for the opcode image (here, 70 bytes in size). `pass` means that the image was generated in 3 compiler passes, for example, `x86_64` can have various optimization passes to further reduce the image size when possible. `image` contains the address of the generated JIT image, `from` and `pid` the user space application name and PID respectively, which triggered the compilation process. The dump output for eBPF and cBPF JITs is the same format.

In the kernel tree under `tools/net/`, there is a tool called `bpf_jit_disasm`. It reads out the latest dump and prints the disassembly for further inspection:

```
# ./bpf_jit_disasm
70 bytes emitted from JIT compiler (pass:3, flen:6)
ffffffffa0069c8f + <x>:
   0:       push   %rbp
   1:       mov    %rsp,%rbp
   4:       sub    $0x60,%rsp
   8:       mov    %rbx,-0x8(%rbp)
   c:       mov    0x68(%rdi),%r9d
  10:       sub    0x6c(%rdi),%r9d
  14:       mov    0xd8(%rdi),%r8
  1b:       mov    $0xc,%esi
  20:       callq  0xffffffffe0ff9442
  25:       cmp    $0x800,%eax
  2a:       jne    0x0000000000000042
  2c:       mov    $0x17,%esi
  31:       callq  0xffffffffe0ff945e
  36:       cmp    $0x1,%eax
  39:       jne    0x0000000000000042
  3b:       mov    $0xffff,%eax
  40:       jmp    0x0000000000000044
  42:       xor    %eax,%eax
  44:       leaveq
  45:       retq
```

Alternatively, the tool can also dump related opcodes along with the disassembly.

```
# ./bpf_jit_disasm -o
70 bytes emitted from JIT compiler (pass:3, flen:6)
ffffffffa0069c8f + <x>:
   0:       push   %rbp
    55
   1:       mov    %rsp,%rbp
    48 89 e5
   4:       sub    $0x60,%rsp
    48 83 ec 60
   8:       mov    %rbx,-0x8(%rbp)
    48 89 5d f8
   c:       mov    0x68(%rdi),%r9d
    44 8b 4f 68
  10:       sub    0x6c(%rdi),%r9d
    44 2b 4f 6c
  14:       mov    0xd8(%rdi),%r8
    4c 8b 87 d8 00 00 00
  1b:       mov    $0xc,%esi
    be 0c 00 00 00
  20:       callq  0xffffffffe0ff9442
    e8 1d 94 ff e0
  25:       cmp    $0x800,%eax
    3d 00 08 00 00
  2a:       jne    0x0000000000000042
    75 16
  2c:       mov    $0x17,%esi
    be 17 00 00 00
  31:       callq  0xffffffffe0ff945e
    e8 28 94 ff e0
  36:       cmp    $0x1,%eax
    83 f8 01
  39:       jne    0x0000000000000042
    75 07
  3b:       mov    $0xffff,%eax
    b8 ff ff 00 00
  40:       jmp    0x0000000000000044
    eb 02
  42:       xor    %eax,%eax
    31 c0
  44:       leaveq
    c9
  45:       retq
    c3
```

For performance analysis of JITed BPF programs, `perf` can be used as usual. As a prerequisite, JITed programs need to be exported through kallsyms infrastructure.

```
# echo 1 > /proc/sys/net/core/bpf_jit_enable
# echo 1 > /proc/sys/net/core/bpf_jit_kallsyms
```

Enabling or disabling `bpf_jit_kallsyms` does not require a reload of the related BPF programs. Next, a small workflow example is provided for profiling BPF programs. A crafted tc BPF program is used for demonstration purposes, where perf records a failed allocation inside `bpf_clone_redirect()` helper. Due to the use of direct write, `bpf_try_make_head_writable()` failed, which would then release the cloned `skb` again and return with an error message. `perf` thus records all `kfree_skb` events.

```
# tc qdisc add dev em1 clsact
# tc filter add dev em1 ingress bpf da obj prog.o sec main
# tc filter show dev em1 ingress
filter protocol all pref 49152 bpf
filter protocol all pref 49152 bpf handle 0x1 prog.o:[main] direct-action tag␣
↪8227addf251b7543

# cat /proc/kallsyms
[...]
ffffffffc00349e0 t fjes_hw_init_command_registers    [fjes]
ffffffffc003e2e0 d __tracepoint_fjes_hw_stop_debug_err    [fjes]
ffffffffc0036190 t fjes_hw_epbuf_tx_pkt_send    [fjes]
ffffffffc004b000 t bpf_prog_8227addf251b7543

# perf record -a -g -e skb:kfree_skb sleep 60
# perf script --kallsyms=/proc/kallsyms
[...]
ksoftirqd/0     6 [000]  1004.578402:    skb:kfree_skb: skbaddr=0xffff9d4161f20a00␣
↪protocol=2048 location=0xffffffffc004b52c
   7fffb8745961 bpf_clone_redirect (/lib/modules/4.10.0+/build/vmlinux)
   7fffc004e52c bpf_prog_8227addf251b7543 (/lib/modules/4.10.0+/build/vmlinux)
   7fffc05b6283 cls_bpf_classify (/lib/modules/4.10.0+/build/vmlinux)
   7fffb875957a tc_classify (/lib/modules/4.10.0+/build/vmlinux)
   7fffb8729840 __netif_receive_skb_core (/lib/modules/4.10.0+/build/vmlinux)
   7fffb8729e38 __netif_receive_skb (/lib/modules/4.10.0+/build/vmlinux)
   7fffb872ae05 process_backlog (/lib/modules/4.10.0+/build/vmlinux)
   7fffb872a43e net_rx_action (/lib/modules/4.10.0+/build/vmlinux)
   7fffb886176c __do_softirq (/lib/modules/4.10.0+/build/vmlinux)
   7fffb80ac5b9 run_ksoftirqd (/lib/modules/4.10.0+/build/vmlinux)
   7fffb80ca7fa smpboot_thread_fn (/lib/modules/4.10.0+/build/vmlinux)
   7fffb80c6831 kthread (/lib/modules/4.10.0+/build/vmlinux)
   7fffb885e09c ret_from_fork (/lib/modules/4.10.0+/build/vmlinux)
```

The stack trace recorded by `perf` will then show the `bpf_prog_8227addf251b7543()` symbol as part of the call trace, meaning that the BPF program with the tag `8227addf251b7543` was related to the `kfree_skb` event, and such program was attached to netdevice `em1` on the ingress hook as shown by tc.

Introspection

The Linux kernel provides various tracepoints around BPF and XDP which can be used for additional introspection, for example, to trace interactions of user space programs with the bpf system call.

Tracepoints for BPF:

```
# perf list | grep bpf:
bpf:bpf_map_create                                 [Tracepoint event]
bpf:bpf_map_delete_elem                            [Tracepoint event]
bpf:bpf_map_lookup_elem                            [Tracepoint event]
bpf:bpf_map_next_key                               [Tracepoint event]
bpf:bpf_map_update_elem                            [Tracepoint event]
bpf:bpf_obj_get_map                                [Tracepoint event]
```

```
bpf:bpf_obj_get_prog                               [Tracepoint event]
bpf:bpf_obj_pin_map                                [Tracepoint event]
bpf:bpf_obj_pin_prog                               [Tracepoint event]
bpf:bpf_prog_get_type                              [Tracepoint event]
bpf:bpf_prog_load                                  [Tracepoint event]
bpf:bpf_prog_put_rcu                               [Tracepoint event]
```

Example usage with `perf` (alternatively to `sleep` example used here, a specific application like `tc` could be used here instead, of course):

```
# perf record -a -e bpf:* sleep 10
# perf script
sock_example  6197 [005]   283.980322:      bpf:bpf_map_create: map type=ARRAY ufd=4␣
↪key=4 val=8 max=256 flags=0
sock_example  6197 [005]   283.980721:       bpf:bpf_prog_load: prog=a5ea8fa30ea6849c␣
↪type=SOCKET_FILTER ufd=5
sock_example  6197 [005]   283.988423:   bpf:bpf_prog_get_type: prog=a5ea8fa30ea6849c␣
↪type=SOCKET_FILTER
sock_example  6197 [005]   283.988443: bpf:bpf_map_lookup_elem: map type=ARRAY ufd=4␣
↪key=[06 00 00 00] val=[00 00 00 00 00 00 00 00]
[...]
sock_example  6197 [005]   288.990868: bpf:bpf_map_lookup_elem: map type=ARRAY ufd=4␣
↪key=[01 00 00 00] val=[14 00 00 00 00 00 00 00]
     swapper     0 [005]   289.338243:    bpf:bpf_prog_put_rcu: prog=a5ea8fa30ea6849c␣
↪type=SOCKET_FILTER
```

For the BPF programs, their individual program tag is displayed.

For debugging, XDP also has a tracepoint that is triggered when exceptions are raised:

```
# perf list | grep xdp:
xdp:xdp_exception                                  [Tracepoint event]
```

Exceptions are triggered in the following scenarios:

- The BPF program returned an invalid / unknown XDP action code.
- The BPF program returned with `XDP_ABORTED` indicating a non-graceful exit.
- The BPF program returned with `XDP_TX`, but there was an error on transmit, for example, due to the port not being up, due to the transmit ring being full, due to allocation failures, etc.

Both tracepoint classes can also be inspected with a BPF program itself attached to one or more tracepoints, collecting further information in a map or punting such events to a user space collector through the `bpf_perf_event_output()` helper, for example.

Miscellaneous

BPF programs and maps are memory accounted against `RLIMIT_MEMLOCK` similar to `perf`. The currently available size in unit of system pages which may be locked into memory can be inspected through `ulimit -l`. The setrlimit system call man page provides further details.

The default limit is usually insufficient to load more complex programs or larger BPF maps, so that the BPF system call will return with `errno` of `EPERM`. In such situations a workaround with `ulimit -l unlimited` or with a sufficiently large limit could be performed. The `RLIMIT_MEMLOCK` is mainly enforcing limits for unprivileged users. Depending on the setup, setting a higher limit for privileged users is often acceptable.

tc (traffic control)

TODO

XDP

TODO

Further Reading

Mentioned lists of projects, talks, papers, and further reading material are likely not complete. Thus, feel free to open pull requests to complete the list.

Projects using BPF

The following list includes some open source projects making use of BPF:

- BCC - tools for BPF-based Linux IO analysis, networking, monitoring, and more (https://github.com/iovisor/bcc)
- Cilium (https://github.com/cilium/cilium)
- iproute2 (ip and tc tools) (https://wiki.linuxfoundation.org/networking/iproute2)
- perf tool (https://perf.wiki.kernel.org/index.php/Main_Page)
- ply - a dynamic tracer for Linux (https://wkz.github.io/ply)
- Go bindings for creating BPF programs (https://github.com/iovisor/gobpf)
- Suricata IDS (https://suricata-ids.org)

XDP Newbies

There are a couple of walk-through posts by David S. Miller to the xdp-newbies mailing list (http://vger.kernel.org/vger-lists.html#xdp-newbies), which explain various parts of XDP and BPF:

4. **May 2017,** BPF Verifier Overview, David S. Miller, https://www.spinics.net/lists/xdp-newbies/msg00185.html
3. **May 2017,** Contextually speaking..., David S. Miller, https://www.spinics.net/lists/xdp-newbies/msg00181.html
2. **May 2017,** bpf.h and you..., David S. Miller, https://www.spinics.net/lists/xdp-newbies/msg00179.html
1. **Apr 2017,** XDP example of the day, David S. Miller, https://www.spinics.net/lists/xdp-newbies/msg00009.html

BPF Newsletter

Alexander Alemayhu initiated a newsletter around BPF that appears roughly once per week covering latest developments around BPF in Linux kernel land and its surrounding ecosystem in user space:

5. **May 2017,** BPF Updates 05, Alexander Alemayhu, https://www.cilium.io/blog/2017/5/31/bpf-updates-05
4. **May 2017,** BPF Updates 04, Alexander Alemayhu, https://www.cilium.io/blog/2017/5/24/bpf-updates-04

3. **May 2017,** BPF Updates 03, Alexander Alemayhu, https://www.cilium.io/blog/2017/5/17/bpf-updates-03
2. **May 2017,** BPF Updates 02, Alexander Alemayhu, https://www.cilium.io/blog/2017/5/10/bpf-updates-02
1. **May 2017,** BPF Updates 01, Alexander Alemayhu, https://www.cilium.io/blog/2017/5/2/bpf-updates-01-2017-05-02

Podcasts

There have been a number of technical podcasts partially covering BPF. Incomplete list:

5. **Feb 2017,** Linux Networking Update from Netdev Conference, Thomas Graf, Software Gone Wild, Show 71, http://blog.ipspace.net/2017/02/linux-networking-update-from-netdev.html http://media.blubrry.com/ipspace/stream.ipspace.net/nuggets/podcast/Show_71-NetDev_Update.mp3
4. **Jan 2017,** The IO Visor Project, Brenden Blanco, OVS Orbit, Episode 23, https://ovsorbit.org/#e23 https://ovsorbit.org/episode-23.mp3
3. **Oct 2016,** Fast Linux Packet Forwarding, Thomas Graf, Software Gone Wild, Show 64, http://blog.ipspace.net/2016/10/fast-linux-packet-forwarding-with.html http://media.blubrry.com/ipspace/stream.ipspace.net/nuggets/podcast/Show_64-Cilium_with_Thomas_Graf.mp3
2. **Aug 2016,** P4 on the Edge, John Fastabend, OVS Orbit, Episode 11, https://ovsorbit.org/#e11 https://ovsorbit.org/episode-11.mp3
1. **May 2016,** Cilium, Thomas Graf, OVS Orbit, Episode 4, https://ovsorbit.org/#e4 https://ovsorbit.benpfaff.org/episode-4.mp3

Blog posts

The following (incomplete) list includes blog posts around BPF, XDP and related projects:

34. **May 2017,** An entertaining eBPF XDP adventure, Suchakra Sharma, https://suchakra.wordpress.com/2017/05/23/an-entertaining-ebpf-xdp-adventure/
33. **May 2017,** eBPF, part 2: Syscall and Map Types, Ferris Ellis, https://ferrisellis.com/posts/ebpf_syscall_and_maps/
32. **May 2017,** Monitoring the Control Plane, Gary Berger, http://firstclassfunc.com/2017/05/monitoring-the-control-plane/
31. **Apr 2017,** USENIX/LISA 2016 Linux bcc/BPF Tools, Brendan Gregg, http://www.brendangregg.com/blog/2017-04-29/usenix-lisa-2016-bcc-bpf-tools.html
30. **Apr 2017,** Liveblog: Cilium for Network and Application Security with BPF and XDP, Scott Lowe, http://blog.scottlowe.org//2017/04/18/black-belt-cilium/
29. **Apr 2017,** eBPF, part 1: Past, Present, and Future, Ferris Ellis, https://ferrisellis.com/posts/ebpf_past_present_future/
28. **Mar 2017,** Analyzing KVM Hypercalls with eBPF Tracing, Suchakra Sharma, https://suchakra.wordpress.com/2017/03/31/analyzing-kvm-hypercalls-with-ebpf-tracing/
27. **Jan 2017,** Golang bcc/BPF Function Tracing, Brendan Gregg, http://www.brendangregg.com/blog/2017-01-31/golang-bcc-bpf-function-tracing.html
26. **Dec 2016,** Give me 15 minutes and I'll change your view of Linux tracing, Brendan Gregg, http://www.brendangregg.com/blog/2016-12-27/linux-tracing-in-15-minutes.html
25. **Nov 2016,** Cilium: Networking and security for containers with BPF and XDP, Daniel Borkmann, https://opensource.googleblog.com/2016/11/cilium-networking-and-security.html

24. **Nov 2016,** Linux bcc/BPF tcplife: TCP Lifespans, Brendan Gregg, http://www.brendangregg.com/blog/2016-11-30/linux-bcc-tcplife.html
23. **Oct 2016,** DTrace for Linux 2016, Brendan Gregg, http://www.brendangregg.com/blog/2016-10-27/dtrace-for-linux-2016.html
22. **Oct 2016,** Linux 4.9's Efficient BPF-based Profiler, Brendan Gregg, http://www.brendangregg.com/blog/2016-10-21/linux-efficient-profiler.html
21. **Oct 2016,** Linux bcc tcptop, Brendan Gregg, http://www.brendangregg.com/blog/2016-10-15/linux-bcc-tcptop.html
20. **Oct 2016,** Linux bcc/BPF Node.js USDT Tracing, Brendan Gregg, http://www.brendangregg.com/blog/2016-10-12/linux-bcc-nodejs-usdt.html
19. **Oct 2016,** Linux bcc/BPF Run Queue (Scheduler) Latency, Brendan Gregg, http://www.brendangregg.com/blog/2016-10-08/linux-bcc-runqlat.html
18. **Oct 2016,** Linux bcc ext4 Latency Tracing, Brendan Gregg, http://www.brendangregg.com/blog/2016-10-06/linux-bcc-ext4dist-ext4slower.html
17. **Oct 2016,** Linux MySQL Slow Query Tracing with bcc/BPF, Brendan Gregg, http://www.brendangregg.com/blog/2016-10-04/linux-bcc-mysqld-qslower.html
16. **Oct 2016,** Linux bcc Tracing Security Capabilities, Brendan Gregg, http://www.brendangregg.com/blog/2016-10-01/linux-bcc-security-capabilities.html
15. **Sep 2016,** Suricata bypass feature, Eric Leblond, https://www.stamus-networks.com/2016/09/28/suricata-bypass-feature/
14. **Aug 2016,** Introducing the p0f BPF compiler, Gilberto Bertin, https://blog.cloudflare.com/introducing-the-p0f-bpf-compiler/
13. **Jun 2016,** Ubuntu Xenial bcc/BPF, Brendan Gregg, http://www.brendangregg.com/blog/2016-06-14/ubuntu-xenial-bcc-bpf.html
12. **Mar 2016,** Linux BPF/bcc Road Ahead, March 2016, Brendan Gregg, http://www.brendangregg.com/blog/2016-03-28/linux-bpf-bcc-road-ahead-2016.html
11. **Mar 2016,** Linux BPF Superpowers, Brendan Gregg, http://www.brendangregg.com/blog/2016-03-05/linux-bpf-superpowers.html
10. **Feb 2016,** Linux eBPF/bcc uprobes, Brendan Gregg, http://www.brendangregg.com/blog/2016-02-08/linux-ebpf-bcc-uprobes.html
9. **Feb 2016,** Who is waking the waker? (Linux chain graph prototype), Brendan Gregg, http://www.brendangregg.com/blog/2016-02-05/ebpf-chaingraph-prototype.html
8. **Feb 2016,** Linux Wakeup and Off-Wake Profiling, Brendan Gregg, http://www.brendangregg.com/blog/2016-02-01/linux-wakeup-offwake-profiling.html
7. **Jan 2016,** Linux eBPF Off-CPU Flame Graph, Brendan Gregg, http://www.brendangregg.com/blog/2016-01-20/ebpf-offcpu-flame-graph.html
6. **Jan 2016,** Linux eBPF Stack Trace Hack, Brendan Gregg, http://www.brendangregg.com/blog/2016-01-18/ebpf-stack-trace-hack.html
1. **Sep 2015,** Linux Networking, Tracing and IO Visor, a New Systems Performance Tool for a Distributed World, Suchakra Sharma, https://thenewstack.io/comparing-dtrace-iovisor-new-systems-performance-platform-advance-linux-networking-virtualization/
5. **Aug 2015,** BPF Internals - II, Suchakra Sharma, https://suchakra.wordpress.com/2015/08/12/bpf-internals-ii/

4. **May 2015,** eBPF: One Small Step, Brendan Gregg, http://www.brendangregg.com/blog/2015-05-15/ebpf-one-small-step.html
3. **May 2015,** BPF Internals - I, Suchakra Sharma, https://suchakra.wordpress.com/2015/05/18/bpf-internals-i/
2. **Jul 2014,** Introducing the BPF Tools, Marek Majkowski, https://blog.cloudflare.com/introducing-the-bpf-tools/
1. **May 2014,** BPF - the forgotten bytecode, Marek Majkowski, https://blog.cloudflare.com/bpf-the-forgotten-bytecode/

Talks

The following (incomplete) list includes talks and conference papers related to BPF and XDP:

44. **May 2017,** PyCon 2017, Portland, Executing python functions in the linux kernel by transpiling to bpf, Alex Gartrell, https://www.youtube.com/watch?v=CpqMroMBGP4
43. **May 2017,** gluecon 2017, Denver, Cilium + BPF: Least Privilege Security on API Call Level for Microservices, Dan Wendlandt, http://gluecon.com/#agenda
42. **May 2017,** Lund Linux Con, Lund, XDP - eXpress Data Path, Jesper Dangaard Brouer, http://people.netfilter.org/hawk/presentations/LLC2017/XDP_DDoS_protecting_LLC2017.pdf
41. **May 2017,** Polytechnique Montreal, Trace Aggregation and Collection with eBPF, Suchakra Sharma, http://step.polymtl.ca/~suchakra/eBPF-5May2017.pdf
40. **Apr 2017,** DockerCon, Austin, Cilium - Network and Application Security with BPF and XDP, Thomas Graf, https://www.slideshare.net/ThomasGraf5/dockercon-2017-cilium-network-and-application-security-with-bpf-and-xdp
39. **Apr 2017,** NetDev 2.1, Montreal, XDP Mythbusters, David S. Miller, https://www.netdevconf.org/2.1/slides/apr7/miller-XDP-MythBusters.pdf
38. **Apr 2017,** NetDev 2.1, Montreal, Droplet: DDoS countermeasures powered by BPF + XDP, Huapeng Zhou, Doug Porter, Ryan Tierney, Nikita Shirokov, https://www.netdevconf.org/2.1/slides/apr6/zhou-netdev-xdp-2017.pdf
37. **Apr 2017,** NetDev 2.1, Montreal, XDP in practice: integrating XDP in our DDoS mitigation pipeline, Gilberto Bertin, https://www.netdevconf.org/2.1/slides/apr6/bertin_Netdev-XDP.pdf
36. **Apr 2017,** NetDev 2.1, Montreal, XDP for the Rest of Us, Andy Gospodarek, Jesper Dangaard Brouer, https://www.netdevconf.org/2.1/slides/apr7/gospodarek-Netdev2.1-XDP-for-the-Rest-of-Us_Final.pdf
35. **Mar 2017,** SCALE15x, Pasadena, Linux 4.x Tracing: Performance Analysis with bcc/BPF, Brendan Gregg, https://www.slideshare.net/brendangregg/linux-4x-tracing-performance-analysis-with-bccbpf
34. **Mar 2017,** XDP Inside and Out, David S. Miller, https://github.com/iovisor/bpf-docs/raw/master/XDP_Inside_and_Out.pdf
33. **Mar 2017,** OpenSourceDays, Copenhagen, XDP - eXpress Data Path, Used for DDoS protection, Jesper Dangaard Brouer, https://github.com/iovisor/bpf-docs/raw/master/XDP_Inside_and_Out.pdf
32. **Mar 2017,** source{d}, Infrastructure 2017, Madrid, High-performance Linux monitoring with eBPF, Alfonso Acosta, https://www.youtube.com/watch?v=k4jqTLtdrxQ
31. **Feb 2017,** FOSDEM 2017, Brussels, Stateful packet processing with eBPF, an implementation of OpenState interface, Quentin Monnet, https://fosdem.org/2017/schedule/event/stateful_ebpf/
30. **Feb 2017,** FOSDEM 2017, Brussels, eBPF and XDP walkthrough and recent updates, Daniel Borkmann, http://borkmann.ch/talks/2017_fosdem.pdf

29. **Feb 2017,** FOSDEM 2017, Brussels, Cilium - BPF & XDP for containers, Thomas Graf, https://fosdem.org/2017/schedule/event/cilium/
28. **Jan 2017,** linuxconf.au, Hobart, BPF: Tracing and more, Brendan Gregg, https://www.slideshare.net/brendangregg/bpf-tracing-and-more
27. **Dec 2016,** USENIX LISA 2016, Boston, Linux 4.x Tracing Tools: Using BPF Superpowers, Brendan Gregg, https://www.slideshare.net/brendangregg/linux-4x-tracing-tools-using-bpf-superpowers
26. **Nov 2016,** Linux Plumbers, Santa Fe, Cilium: Networking & Security for Containers with BPF & XDP, Thomas Graf, http://www.slideshare.net/ThomasGraf5/clium-container-networking-with-bpf-xdp
25. **Nov 2016,** OVS Conference, Santa Clara, Offloading OVS Flow Processing using eBPF, William (Cheng-Chun) Tu, http://openvswitch.org/support/ovscon2016/7/1120-tu.pdf
24. **Oct 2016,** One.com, Copenhagen, XDP - eXpress Data Path, Intro and future use-cases, Jesper Dangaard Brouer, http://people.netfilter.org/hawk/presentations/xdp2016/xdp_intro_and_use_cases_sep2016.pdf
23. **Oct 2016,** Docker Distributed Systems Summit, Berlin, Cilium: Networking & Security for Containers with BPF & XDP, Thomas Graf, http://www.slideshare.net/Docker/cilium-bpf-xdp-for-containers-66969823
22. **Oct 2016,** NetDev 1.2, Tokyo, Data center networking stack, Tom Herbert, http://netdevconf.org/1.2/session.html?tom-herbert
21. **Oct 2016,** NetDev 1.2, Tokyo, Fast Programmable Networks & Encapsulated Protocols, David S. Miller, http://netdevconf.org/1.2/session.html?david-miller-keynote
20. **Oct 2016,** NetDev 1.2, Tokyo, XDP workshop - Introduction, experience, and future development, Tom Herbert, http://netdevconf.org/1.2/session.html?herbert-xdp-workshop
19. **Oct 2016,** NetDev1.2, Tokyo, The adventures of a Suricate in eBPF land, Eric Leblond, http://netdevconf.org/1.2/slides/oct6/10_suricata_ebpf.pdf
18. **Oct 2016,** NetDev1.2, Tokyo, cls_bpf/eBPF updates since netdev 1.1, Daniel Borkmann, http://borkmann.ch/talks/2016_tcws.pdf
17. **Oct 2016,** NetDev1.2, Tokyo, Advanced programmability and recent updates with tc's cls_bpf, Daniel Borkmann, http://borkmann.ch/talks/2016_netdev2.pdf http://www.netdevconf.org/1.2/papers/borkmann.pdf
16. **Oct 2016,** NetDev 1.2, Tokyo, eBPF/XDP hardware offload to SmartNICs, Jakub Kicinski, Nic Viljoen, http://netdevconf.org/1.2/papers/eBPF_HW_OFFLOAD.pdf
15. **Aug 2016,** LinuxCon, Toronto, What Can BPF Do For You?, Brenden Blanco, https://events.linuxfoundation.org/sites/events/files/slides/iovisor-lc-bof-2016.pdf
14. **Aug 2016,** LinuxCon, Toronto, Cilium - Fast IPv6 Container Networking with BPF and XDP, Thomas Graf, https://www.slideshare.net/ThomasGraf5/cilium-fast-ipv6-container-networking-with-bpf-and-xdp
13. **Aug 2016,** P4, EBPF and Linux TC Offload, Dinan Gunawardena, Jakub Kicinski, https://de.slideshare.net/Open-NFP/p4-epbf-and-linux-tc-offload
12. **Jul 2016,** Linux Meetup, Santa Clara, eXpress Data Path, Brenden Blanco, http://www.slideshare.net/IOVisor/express-data-path-linux-meetup-santa-clara-july-2016
11. **Jul 2016,** Linux Meetup, Santa Clara, CETH for XDP, Yan Chan, Yunsong Lu, http://www.slideshare.net/IOVisor/ceth-for-xdp-linux-meetup-santa-clara-july-2016
10. **May 2016,** P4 workshop, Stanford, P4 on the Edge, John Fastabend, https://schd.ws/hosted_files/2016p4workshop/1d/Intel%20Fastabend-P4%20on%20the%20Edge.pdf
9. **Mar 2016,** Performance @Scale 2016, Menlo Park, Linux BPF Superpowers, Brendan Gregg, https://www.slideshare.net/brendangregg/linux-bpf-superpowers

8. **Mar 2016,** eXpress Data Path, Tom Herbert, Alexei Starovoitov, https://github.com/iovisor/bpf-docs/raw/master/Express_Data_Path.pdf

7. **Feb 2016,** NetDev1.1, Seville, On getting tc classifier fully programmable with cls_bpf, Daniel Borkmann, http://borkmann.ch/talks/2016_netdev.pdf http://www.netdevconf.org/1.1/proceedings/papers/On-getting-tc-classifier-fully-programmable-with-cls-bpf.pdf

6. **Jan 2016,** FOSDEM 2016, Brussels, Linux tc and eBPF, Daniel Borkmann, http://borkmann.ch/talks/2016_fosdem.pdf

5. **Oct 2015,** LinuxCon Europe, Dublin, eBPF on the Mainframe, Michael Holzheu, https://events.linuxfoundation.org/sites/events/files/slides/ebpf_on_the_mainframe_lcon_2015.pdf

4. **Aug 2015,** Tracing Summit, Seattle, LLTng's Trace Filtering and beyond (with some eBPF goodness, of course!), Suchakra Sharma, https://github.com/iovisor/bpf-docs/raw/master/ebpf_excerpt_20Aug2015.pdf

3. **Jun 2015,** LinuxCon Japan, Tokyo, Exciting Developments in Linux Tracing, Elena Zannoni, https://events.linuxfoundation.org/sites/events/files/slides/tracing-linux-ezannoni-linuxcon-ja-2015_0.pdf

2. **Feb 2015,** Collaboration Summit, Santa Rosa, BPF: In-kernel Virtual Machine, Alexei Starovoitov, https://events.linuxfoundation.org/sites/events/files/slides/bpf_collabsummit_2015feb20.pdf

1. **Feb 2015,** NetDev 0.1, Ottawa, BPF: In-kernel Virtual Machine, Alexei Starovoitov, http://netdevconf.org/0.1/sessions/15.html

0. **Feb 2014,** DevConf.cz, Brno, tc and cls_bpf: lightweight packet classifying with BPF, Daniel Borkmann, http://borkmann.ch/talks/2014_devconf.pdf

Further Documents

- Dive into BPF: a list of reading material, Quentin Monnet (https://qmonnet.github.io/whirl-offload/2016/09/01/dive-into-bpf/)
- XDP - eXpress Data Path, Jesper Dangaard Brouer (https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/index.html)

CHAPTER 8

API Reference

GET /healthz

Get health of Cilium daemon

Returns health and status information of the Cilium daemon and related components such as the local container runtime, connected datastore, Kubernetes integration.

Status Codes

- 200 OK – Success

GET /config

Get configuration of Cilium daemon

Returns the configuration of the Cilium daemon.

Status Codes

- 200 OK – Success

PATCH /config

Modify daemon configuration

Updates the daemon configuration by applying the provided ConfigurationMap and regenerates & recompiles all required datapath components.

Status Codes

- 200 OK – Success
- 400 Bad Request – Bad configuration parameters
- 500 Internal Server Error – Recompilation failed

GET /endpoint/{id}

Get endpoint by endpoint ID

Returns endpoint information

Parameters

- **id** (*string*) – String describing an endpoint with the format *[prefix:]id*. If no prefix is specified, a prefix of *cilium-local:* is assumed. Not all endpoints will be addressable by all endpoint ID prefixes with the exception of the local Cilium UUID which is assigned to all endpoints.

  Supported endpoint id prefixes:

  - cilium-local: Local Cilium endpoint UUID, e.g. cilium-local:3389595
  - cilium-global: Global Cilium endpoint UUID, e.g. cilium-global:cluster1:nodeX:452343
  - container-id: Container runtime ID, e.g. container-id:22222
  - container-name: Container name, e.g. container-name:foobar
  - pod-name: pod name for this container if K8s is enabled, e.g. pod-name:default:foobar
  - docker-net-endpoint: Docker libnetwork endpoint ID, e.g. docker-net-endpoint:4444

Status Codes

- 200 OK – Success
- 400 Bad Request – Invalid endpoint ID format for specified type
- 404 Not Found – Endpoint not found

`PUT /endpoint/{id}`

Create endpoint

Updates an existing endpoint

Parameters

- **id** (*string*) – String describing an endpoint with the format *[prefix:]id*. If no prefix is specified, a prefix of *cilium-local:* is assumed. Not all endpoints will be addressable by all endpoint ID prefixes with the exception of the local Cilium UUID which is assigned to all endpoints.

  Supported endpoint id prefixes:

  - cilium-local: Local Cilium endpoint UUID, e.g. cilium-local:3389595
  - cilium-global: Global Cilium endpoint UUID, e.g. cilium-global:cluster1:nodeX:452343
  - container-id: Container runtime ID, e.g. container-id:22222
  - container-name: Container name, e.g. container-name:foobar
  - pod-name: pod name for this container if K8s is enabled, e.g. pod-name:default:foobar
  - docker-net-endpoint: Docker libnetwork endpoint ID, e.g. docker-net-endpoint:4444

Status Codes

- 201 Created – Created
- 400 Bad Request – Invalid endpoint in request
- 409 Conflict – Endpoint already exists
- 500 Internal Server Error – Endpoint creation failed

`PATCH /endpoint/{id}`

Modify existing endpoint

Applies the endpoint change request to an existing endpoint

Parameters

- **id** (*string*) – String describing an endpoint with the format *[prefix:]id*. If no prefix is specified, a prefix of *cilium-local:* is assumed. Not all endpoints will be addressable by all endpoint ID prefixes with the exception of the local Cilium UUID which is assigned to all endpoints.

  Supported endpoint id prefixes:

  - cilium-local: Local Cilium endpoint UUID, e.g. cilium-local:3389595
  - cilium-global: Global Cilium endpoint UUID, e.g. cilium-global:cluster1:nodeX:452343
  - container-id: Container runtime ID, e.g. container-id:22222
  - container-name: Container name, e.g. container-name:foobar
  - pod-name: pod name for this container if K8s is enabled, e.g. pod-name:default:foobar
  - docker-net-endpoint: Docker libnetwork endpoint ID, e.g. docker-net-endpoint:4444

Status Codes

- 200 OK – Success
- 400 Bad Request – Invalid modify endpoint request
- 404 Not Found – Endpoint does not exist
- 500 Internal Server Error – Endpoint update failed

DELETE /endpoint/{id}

Delete endpoint

Deletes the endpoint specified by the ID. Deletion is imminent and atomic, if the deletion request is valid and the endpoint exists, deletion will occur even if errors are encountered in the process. If errors have been encountered, the code 202 will be returned, otherwise 200 on success.

All resources associated with the endpoint will be freed and the workload represented by the endpoint will be disconnected.It will no longer be able to initiate or receive communications of any sort.

Parameters

- **id** (*string*) – String describing an endpoint with the format *[prefix:]id*. If no prefix is specified, a prefix of *cilium-local:* is assumed. Not all endpoints will be addressable by all endpoint ID prefixes with the exception of the local Cilium UUID which is assigned to all endpoints.

  Supported endpoint id prefixes:

  - cilium-local: Local Cilium endpoint UUID, e.g. cilium-local:3389595
  - cilium-global: Global Cilium endpoint UUID, e.g. cilium-global:cluster1:nodeX:452343
  - container-id: Container runtime ID, e.g. container-id:22222
  - container-name: Container name, e.g. container-name:foobar
  - pod-name: pod name for this container if K8s is enabled, e.g. pod-name:default:foobar
  - docker-net-endpoint: Docker libnetwork endpoint ID, e.g. docker-net-endpoint:4444

Status Codes

- 200 OK – Success
- 206 Partial Content – Deleted with a number of errors encountered
- 400 Bad Request – Invalid endpoint ID format for specified type. Details in error message
- 404 Not Found – Endpoint not found

GET /endpoint

Retrieves a list of endpoints that have metadata matching the provided parameters.

Retrieves a list of endpoints that have metadata matching the provided parameters, or all endpoints if no parameters provided.

Status Codes

- 200 OK – Success
- 404 Not Found – Endpoints with provided parameters not found

GET /endpoint/{id}/config

Retrieve endpoint configuration

Retrieves the configuration of the specified endpoint.

Parameters

- **id** (*string*) – String describing an endpoint with the format *[prefix:]id*. If no prefix is specified, a prefix of *cilium-local:* is assumed. Not all endpoints will be addressable by all endpoint ID prefixes with the exception of the local Cilium UUID which is assigned to all endpoints.

  Supported endpoint id prefixes:

  - cilium-local: Local Cilium endpoint UUID, e.g. cilium-local:3389595
  - cilium-global: Global Cilium endpoint UUID, e.g. cilium-global:cluster1:nodeX:452343
  - container-id: Container runtime ID, e.g. container-id:22222
  - container-name: Container name, e.g. container-name:foobar
  - pod-name: pod name for this container if K8s is enabled, e.g. pod-name:default:foobar
  - docker-net-endpoint: Docker libnetwork endpoint ID, e.g. docker-net-endpoint:4444

Status Codes

- 200 OK – Success
- 404 Not Found – Endpoint not found

PATCH /endpoint/{id}/config

Modify mutable endpoint configuration

Update the configuration of an existing endpoint and regenerates & recompiles the corresponding programs automatically.

Parameters

- **id** (*string*) – String describing an endpoint with the format *[prefix:]id*. If no prefix is specified, a prefix of *cilium-local:* is assumed. Not all endpoints will be addressable by all endpoint ID prefixes with the exception of the local Cilium UUID which is assigned to all endpoints.

  Supported endpoint id prefixes:

- cilium-local: Local Cilium endpoint UUID, e.g. cilium-local:3389595
- cilium-global: Global Cilium endpoint UUID, e.g. cilium-global:cluster1:nodeX:452343
- container-id: Container runtime ID, e.g. container-id:22222
- container-name: Container name, e.g. container-name:foobar
- pod-name: pod name for this container if K8s is enabled, e.g. pod-name:default:foobar
- docker-net-endpoint: Docker libnetwork endpoint ID, e.g. docker-net-endpoint:4444

Status Codes

- 200 OK – Success
- 400 Bad Request – Invalid configuration request
- 404 Not Found – Endpoint not found
- 500 Internal Server Error – Update failed. Details in message.

`GET /endpoint/{id}/labels`

Retrieves the list of labels associated with an endpoint.

Parameters

- **`id`** (*`string`*) – String describing an endpoint with the format *[prefix:]id*. If no prefix is specified, a prefix of *cilium-local:* is assumed. Not all endpoints will be addressable by all endpoint ID prefixes with the exception of the local Cilium UUID which is assigned to all endpoints.

  Supported endpoint id prefixes:

  - cilium-local: Local Cilium endpoint UUID, e.g. cilium-local:3389595
  - cilium-global: Global Cilium endpoint UUID, e.g. cilium-global:cluster1:nodeX:452343
  - container-id: Container runtime ID, e.g. container-id:22222
  - container-name: Container name, e.g. container-name:foobar
  - pod-name: pod name for this container if K8s is enabled, e.g. pod-name:default:foobar
  - docker-net-endpoint: Docker libnetwork endpoint ID, e.g. docker-net-endpoint:4444

Status Codes

- 200 OK – Success
- 404 Not Found – Endpoint not found

`PUT /endpoint/{id}/labels`

Modify label configuration of endpoint

Updates the list of labels associated with an endpoint by applying a label modificator structure to the label configuration of an endpoint.

The label configuration mutation is only executed as a whole, i.e. if any of the labels to be deleted are not either on the list of orchestration system labels, custom labels, or already disabled, then the request will fail. Labels to be added which already exist on either the orchestration list or custom list will be ignored.

Parameters

- **id** (*string*) – String describing an endpoint with the format *[prefix:]id*. If no prefix is specified, a prefix of *cilium-local:* is assumed. Not all endpoints will be addressable by all endpoint ID prefixes with the exception of the local Cilium UUID which is assigned to all endpoints.

  Supported endpoint id prefixes:

  - cilium-local: Local Cilium endpoint UUID, e.g. cilium-local:3389595
  - cilium-global: Global Cilium endpoint UUID, e.g. cilium-global:cluster1:nodeX:452343
  - container-id: Container runtime ID, e.g. container-id:22222
  - container-name: Container name, e.g. container-name:foobar
  - pod-name: pod name for this container if K8s is enabled, e.g. pod-name:default:foobar
  - docker-net-endpoint: Docker libnetwork endpoint ID, e.g. docker-net-endpoint:4444

Status Codes

- 200 OK – Success
- 404 Not Found – Endpoint not found
- **460** – Label to be deleted not found
- 500 Internal Server Error – Error while updating labels

GET /identity

Retrieve identity by labels

Status Codes

- 200 OK – Success
- 404 Not Found – Identity not found
- **520** – Identity storage unreachable. Likely a network problem.
- **521** – Invalid identity format in storage

GET /identity/{id}

Retrieve identity

Parameters

- **id** (*string*) – Cluster wide unique identifier of a security identity.

Status Codes

- 200 OK – Success
- 400 Bad Request – Invalid identity provided
- 404 Not Found – Identity not found
- **520** – Identity storage unreachable. Likely a network problem.
- **521** – Invalid identity format in storage

POST /ipam

Allocate an IP address

Query Parameters

- **family** (*string*) –

Status Codes

- 201 Created – Success
- 502 Bad Gateway – Allocation failure

`POST /ipam/{ip}`

Allocate an IP address

Parameters

- **ip** (*string*) – IP address

Status Codes

- 200 OK – Success
- 400 Bad Request – Invalid IP address
- 409 Conflict – IP already allocated
- 500 Internal Server Error – IP allocation failure. Details in message.
- 501 Not Implemented – Allocation for address family disabled

`DELETE /ipam/{ip}`

Release an allocated IP address

Parameters

- **ip** (*string*) – IP address

Status Codes

- 200 OK – Success
- 400 Bad Request – Invalid IP address
- 404 Not Found – IP address not found
- 500 Internal Server Error – Address release failure
- 501 Not Implemented – Allocation for address family disabled

`GET /policy`

Retrieve entire policy tree

Returns the entire policy tree with all children.

Status Codes

- 200 OK – Success
- 404 Not Found – No policy rules found

`PUT /policy`

Create or update a policy (sub)tree

Status Codes

- 200 OK – Success
- 400 Bad Request – Invalid policy
- **460** – Invalid path
- 500 Internal Server Error – Policy import failed

`DELETE /policy`

Delete a policy (sub)tree

Status Codes

- 200 OK – Success
- 400 Bad Request – Invalid request
- 404 Not Found – Policy not found
- 500 Internal Server Error – Error while deleting policy

`GET /policy/resolve`

Resolve policy for an identity context

Status Codes

- 200 OK – Success

`GET /service`

Retrieve list of all services

Status Codes

- 200 OK – Success

`GET /service/{id}`

Retrieve configuration of a service

Parameters

- **id** (*integer*) – ID of service

Status Codes

- 200 OK – Success
- 404 Not Found – Service not found

`PUT /service/{id}`

Create or update service

Parameters

- **id** (*integer*) – ID of service

Status Codes

- 200 OK – Updated
- 201 Created – Created
- **460** – Invalid frontend in service configuration
- **461** – Invalid backend in service configuration
- 500 Internal Server Error – Error while creating service

`DELETE /service/{id}`

Delete a service

Parameters

- **id** (*integer*) – ID of service

Status Codes

- 200 OK – Success
- 404 Not Found – Service not found
- 500 Internal Server Error – Service deletion failed

CHAPTER 9

Developer / Contributor Guide

We're happy you're interested in contributing to the Cilium project.

This guide will help you make sure you have an environment capable of testing changes to the Cilium source code, and that you understand the workflow of getting these changes reviewed and merged upstream.

Setting up a development environment

Developer requirements

You need to have the following tools available in order to effectively contribute to Cilium:

Dependency	Version / Commit ID	Download Command
git	latest	N/A (OS-specific)
go	1.8.3	N/A (OS-specific)
go-swagger	*fbc64c26*	`go get -u github.com/go-swagger/go-swagger/cmd/swagger`
go-bindata	*a0ff2567cfb*	`go get -u github.com/jteeuwen/go-bindata/...`

To run Cilium locally on VMs, you need:

Dependency	Version / Commit ID	Download Command
Vagrant	>= 1.8.3	Vagrant Install Instructions
VirtualBox (if not using libvirt)	>= 5.1.22	N/A (OS-specific)

Finally, in order to build the documentation, you should have Sphinx installed:

```
$ sudo pip install sphinx
```

You should start with the *Getting Started*, which walks you through the set-up, such as installing Vagrant, getting the Cilium sources, and going through some Cilium basics.

Vagrant Setup

While the the *Getting Started* uses a Vagrantfile tuned for the basic walk through, the setup for the Vagrantfile in the root of the Cilium tree depends on a number of environment variables and network setup that are managed via `contrib/vagrant/start.sh`.

Using the provided Vagrantfile

To bring up a Vagrant VM with Cilium plus dependencies installed, run:

```
$ contrib/vagrant/start.sh
```

This will create and run a vagrant VM based on the base box `cilium/ubuntu-16.10`. The box is currently available for the following providers:

- libvirt
- virtualbox

Options

The following environment variables can be set to customize the VMs brought up by vagrant:

- `NWORKERS=n`: Number of child nodes you want to start with the master, default 0.
- `RELOAD=1`: Issue a `vagrant reload` instead of `vagrant up`
- `NFS=1`: Use NFS for vagrant shared directories instead of rsync
- `K8S=1`: Build & install kubernetes on the nodes
- `IPV4=1`: Run Cilium with IPv4 enabled
- VAGRANT_DEFAULT_PROVIDER={virtualbox | libvirt | ...}

If you want to start the VM with cilium enabled with IPv4, with kubernetes installed and plus a worker, run:

```
$ IPV4=1 K8S=1 NWORKERS=1 contrib/vagrant/start.sh
```

If you have any issue with the provided vagrant box `cilium/ubuntu-16.10` or need a different box format, you may build the box yourself using the packer scripts

Manual Installation

Alternatively you can import the vagrant box `cilium/ubuntu-16.10` directly and manually install Cilium:

```
$ vagrant init cilium/ubuntu-16.10
$ vagrant up
$ vagrant ssh [...]
$ cd go/src/github.com/cilium/cilium/
$ make
$ sudo make install
$ sudo cp contrib/upstart/* /etc/init/
$ sudo usermod -a -G cilium vagrant
$ sudo service cilium restart
```

Notes

Your Cilium tree is mapped to the VM so that you do not need to keep copying files between your host and the VM. The default sync method is rsync, which only syncs when the VM is brought up, or when manually triggered (`vagrant rsync` command in the Cilium tree). You can also use NFS to access your Cilium tree from the VM by setting the environment variable `NFS` (mentioned above) before running the startup script (`export NFS=1`). Note that your host firewall have the NFS UDP ports open, the startup script will give the address and port details for this.

Note: OSX file system is by default case insensitive, which can confuse git. At the writing of this Cilium repo has no file names that would be considered referring to the same file on a case insensitive file system. Regardless, it may be useful to create a disk image with a case sensitive file system for holding your git repos.

Note: VirtualBox for OSX currently (version 5.1.22) always reports host-only networks' prefix length as 64. Cilium needs this prefix to be 16, and the startup script will check for this. This check always fails when using VirtualBox on OSX, but it is safe to let the startup script to reset the prefix length to 16.

If for some reason, running of the provisioning script fails, you should bring the VM down before trying again:

```
$ vagrant halt
```

Development Cycle

The Vagrantfile in the Cilium repo root (hereon just `Vagrantfile`), always provisions Cilium build and install when the VM is started. After the initial build and install you can do further building and testing incrementally inside the VM. `vagrant ssh` takes you to the Cilium source tree directory (`/home/vagrant/go/src/github.com/cilium/cilium`) by default, and the following commands assume that being your current directory.

Build

Assuming you have synced (rsync) the source tree after you have made changes, or the tree is automatically in sync via NFS or guest additions folder sharing, you can issue a build as follows:

```
$ make
```

A successful build should be followed by running the unit tests:

```
$ make tests
```

Install

After a successful build and test you can re-install Cilium by:

```
$ sudo -E make install
```

Restart Cilium service

To run the newly installed version of Cilium, restart the service:

```
$ sudo service cilium restart
```

You can verify the service and cilium-agent status by the following commands, respectively:

```
$ service cilium status
$ cilium status
```

Testsuite

After the new version of Cilium is running, you should run the runtime tests:

```
$ sudo make runtime-tests
```

Building Documentation

Whenever making changes to Cilium documentation you should check that you did not introduce any new warnings or errors, and also check that your changes look as you intended. To do this you can build the docs:

```
$ make -C Documentation html
```

After this you can browse the updated docs as HTML starting at `Documentation\_build\html\index.html`.

Submitting a pull request

Contributions may be submitted in the form of pull requests against the github repository at: https://github.com/cilium/cilium

Before hitting the submit button, please make sure that the following requirements have been met:

- The pull request and all corresponding commits have been equipped with a well written commit message which explains the reasoning and details of the change.
- You have added unit and/or runtime tests where feasible.
- You have tested the changes and checked for regressions by running the existing testsuite against your changes. See the "Testsuite" section for additional details.
- You have signed off on your commits, see the section "Developer's Certificate of Origin" for more details.

Release Process

Cilium schedules a major release every 3 months. Each major release is performed by incrementing the *Y* in the version format *X.Y.0*. The group of committers can decide to increment *X* instead to mark major milestones in which case *Y* is reset to 0.

The following steps are performed to publish a release:

1. The master branch is set to the version *X.Y.90* at all times. This ensures that a development snapshot is considered more recent than a stable release at all times.
2. The committers can agree on a series of release candidates which will be tagged *vX.Y-rcN* in the master branch.

3. The committers declare the master branch ready for the release and fork the master branch into a release branch *vX.Y+1.0*.
4. The first commit in the release branch is to change the version to *X.Y+1.0*.
5. The next commit goes into the master branch and sets the version to *X.Y+1.90* to ensure that the master branch will be considered more recent than any stable release of the major release that is about to be published.

Stable releases

The committers can nominate commits pushed to the master as stable release candidates in which case they will be backported to previous release branches. Upon necessity, stable releases are published with the version *X.Y.Z+1*.

Criteria for the inclusion into stable release branches are:

- Security relevant fixes
- Major bugfixes relevant to the correct operation of Cilium

Developer's Certificate of Origin

To improve tracking of who did what, we've introduced a "sign-off" procedure.

The sign-off is a simple line at the end of the explanation for the commit, which certifies that you wrote it or otherwise have the right to pass it on as open-source work. The rules are pretty simple: if you can certify the below:

```
Developer Certificate of Origin
Version 1.1

Copyright (C) 2004, 2006 The Linux Foundation and its contributors.
1 Letterman Drive
Suite D4700
San Francisco, CA, 94129

Everyone is permitted to copy and distribute verbatim copies of this
license document, but changing it is not allowed.


Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the best
    of my knowledge, is covered under an appropriate open source
    license and I have the right under that license to submit that
    work with modifications, whether created in whole or in part
    by me, under the same open source license (unless I am
    permitted to submit under a different license), as indicated
    in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.
```

```
(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including all
    personal information I submit with it, including my sign-off) is
    maintained indefinitely and may be redistributed consistent with
    this project or the open source license(s) involved.
```

then you just add a line saying:

```
Signed-off-by: Random J Developer <random@developer.example.org>
```

Use your real name (sorry, no pseudonyms or anonymous contributions.)

Cilium Committer Grant/Revocation Policy

A Cilium committer is a participant in the project with the ability to commit code directly to the master repository. Commit access grants a broad ability to affect the progress of the project as presented by its most important artifact, the code and related resources that produce working binaries of Cilium. As such it represents a significant level of trust in an individual's commitment to working with other committers and the community at large for the benefit of the project. It can not be granted lightly and, in the worst case, must be revocable if the trust placed in an individual was inappropriate.

This document suggests guidelines for granting and revoking commit access. It is intended to provide a framework for evaluation of such decisions without specifying deterministic rules that wouldn't be sensitive to the nuance of specific situations. In the end the decision to grant or revoke committer privileges is a judgment call made by the existing set of committers.

Expectations for Developers with commit access

Pre-requisites

Be familiar with the *Developer / Contributor Guide*.

Review

Code (yours or others') must be reviewed publicly (by you or others) before you push it to the repository. With one exception (see below), every change needs at least one review.

If one or more people know an area of code particularly well, code that affects that area should ordinarily get a review from one of them.

The riskier, more subtle, or more complicated the change, the more careful the review required. When a change needs careful review, use good judgment regarding the quality of reviews. If a change adds 1000 lines of new code, and a review posted 5 minutes later says just “Looks good,” then this is probably not a quality review.

(The size of a change is correlated with the amount of care needed in review, but it is not strictly tied to it. A search and replace across many files may not need much review, but one-line optimization changes can have widespread implications.)

Your own small changes to fix a recently broken build (“make”) or tests (“make check”), that you believe to be visible to a large number of developers, may be checked in without review. If you are not sure, ask for review.

Regularly review submitted code in areas where you have expertise. Consider reviewing other code as well.

Git conventions

If you apply a change (yours or another's) then it is your responsibility to handle any resulting problems, especially broken builds and other regressions. If it is someone else's change, then you can ask the original submitter to address it. Regardless, you need to ensure that the problem is fixed in a timely way. The definition of "timely" depends on the severity of the problem.

If a bug is present on master and other branches, fix it on master first, then backport the fix to other branches. Straightforward backports do not require additional review (beyond that for the fix on master).

Feature development should be done only on master. Occasionally it makes sense to add a feature to the most recent release branch, before the first actual release of that branch. These should be handled in the same way as bug fixes, that is, first implemented on master and then backported.

Keep the authorship of a commit clear by maintaining a correct list of "Signed-off-by:"s. If a confusing situation comes up, as it occasionally does, bring it up in the development forums. If you explain the use of "Signed-off-by:" to a new developer, explain not just how but why, since the intended meaning of "Signed-off-by:" is more important than the syntax.

Use Reported-by: and Tested-by: tags in commit messages to indicate the source of a bug report.

Keep the AUTHORS file up to date.

Granting Commit Access

Granting commit access should be considered when a candidate has demonstrated the following in their interaction with the project:

- Contribution of significant new features through the patch submission process where:
- Submissions are free of obvious critical defects
- Submissions do not typically require many iterations of improvement to be accepted
- Consistent participation in code review of other's patches, including existing committers, with comments consistent with the overall project standards
- Assistance to those in the community who are less knowledgeable through active participation in project forums.
- Plans for sustained contribution to the project compatible with the project's direction as viewed by current committers.
- Commitment to meet the expectations described in the "Expectations of Developer's with commit access"

The process to grant commit access to a candidate is simple:

- An existing committer nominates the candidate by sending an email to all existing committers with information substantiating the contributions of the candidate in the areas described above.
- All existing committers discuss the pros and cons of granting commit access to the candidate in the email thread.
- When the discussion has converged or a reasonable time has elapsed without discussion developing (e.g. a few business days) the nominator calls for a final decision on the candidate with a followup email to the thread.
- Each committer may vote yes, no, or abstain by replying to the email thread. A failure to reply is an implicit abstention.
- After votes from all existing committers have been collected or a reasonable time has elapsed for them to be provided (e.g. a couple of business days) the votes are evaluated. To be granted commit access the candidate must receive yes votes from a majority of the existing committers and zero no votes. Since a no vote is effectively a veto of the candidate it should be accompanied by a reason for the vote.

- The nominator summarizes the result of the vote in an email to all existing committers.
- If the vote to grant commit access passed, the candidate is contacted with an invitation to become a committer to the project which asks them to agree to the committer expectations documented on the project web site.
- If the candidate agrees access is granted by setting up commit access to the repos.

Revoking Commit Access

There are two situations in which commit access might be revoked.

The straightforward situation is a committer who is no longer active in the project and has no plans to become active in the near future. The process in this case is:

- Any time after a committer has been inactive for more than 6 months any other committer to the project may identify that committer as a candidate for revocation of commit access due to inactivity.
- The plans of revocation should be sent in a private email to the candidate.
- If the candidate for removal states plans to continue participating no action is taken and this process terminates.
- If the candidate replies they no longer require commit access then commit access is removed and a notification is sent to the candidate and all existing committers.
- If the candidate can not be reached within 1 week of the first attempting to contact this process continues.
- A message proposing removal of commit access is sent to the candidate and all other committers.
- If the candidate for removal states plans to continue participating no action is taken.
- If the candidate replies they no longer require commit access then their access is removed.
- If the candidate can not be reached within 2 months of the second attempting to contact them, access is removed.
- In any case, where access is removed, this fact is published through an email to all existing committers (including the candidate for removal).

The more difficult situation is a committer who is behaving in a manner that is viewed as detrimental to the future of the project by other committers. This is a delicate situation with the potential for the creation of division within the greater community and should be handled with care. The process in this case is:

- Discuss the behavior of concern with the individual privately and explain why you believe it is detrimental to the project. Stick to the facts and keep the email professional. Avoid personal attacks and the temptation to hypothesize about unknowable information such as the other's motivations. Make it clear that you would prefer not to discuss the behavior more widely but will have to raise it with other contributors if it does not change. Ideally the behavior is eliminated and no further action is required. If not,
- Start an email thread with all committers, including the source of the behavior, describing the behavior and the reason it is detrimental to the project. The message should have the same tone as the private discussion and should generally repeat the same points covered in that discussion. The person whose behavior is being questioned should not be surprised by anything presented in this discussion. Ideally the wider discussion provides more perspective to all participants and the issue is resolved. If not,
- Start an email thread with all committers except the source of the detrimental behavior requesting a vote on revocation of commit rights. Cite the discussion among all committers and describe all the reasons why it was not resolved satisfactorily. This email should be carefully written with the knowledge that the reasoning it contains may be published to the larger community to justify the decision.
- Each committer may vote yes, no, or abstain by replying to the email thread. A failure to reply is an implicit abstention.

- After all votes have been collected or a reasonable time has elapsed for them to be provided (e.g. a couple of business days) the votes are evaluated. For the request to revoke commit access for the candidate to pass it must receive yes votes from two thirds of the existing committers.
- anyone that votes no must provide their reasoning, and
- if the proposal passes then counter-arguments for the reasoning in no votes should also be documented along with the initial reasons the revocation was proposed. Ideally there should be no new counter-arguments supplied in a no vote as all concerns should have surfaced in the discussion before the vote.
- The original person to propose revocation summarizes the result of the vote in an email to all existing committers excepting the candidate for removal.
- If the vote to revoke commit access passes, access is removed and the candidate for revocation is informed of that fact and the reasons for it as documented in the email requesting the revocation vote.
- Ideally the revoked committer peacefully leaves the community and no further action is required. However, there is a distinct possibility that he/she will try to generate support for his/her point of view within the larger community. In this case the reasoning for removing commit access as described in the request for a vote will be published to the community.

Changing the Policy

The process for changing the policy is:

- Propose the changes to the policy in an email to all current committers and request discussion.
- After an appropriate period of discussion (a few days) update the proposal based on feedback if required and resend it to all current committers with a request for a formal vote.
- After all votes have been collected or a reasonable time has elapsed for them to be provided (e.g. a couple of business days) the votes are evaluated. For the request to modify the policy to pass it must receive yes votes from two thirds of the existing committers.

Template Emails

Nomination to Grant Commit Access

```
I would like to nominate *[candidate]* for commit access. I believe
*[he/she]* has met the conditions for commit access described in the
committer grant policy on the project web site in the following ways:

*[list of requirements & evidence]*

Please reply to all in this message thread with your comments and
questions. If that discussion concludes favorably I will request a formal
vote on the nomination in a few days.
```

Vote to Grant Commit Access

```
I nominated *[candidate]* for commit access on *[date]*. Having allowed
sufficient time for discussion it's now time to formally vote on the
proposal.

Please reply to all in this thread with your vote of: YES, NO, or ABSTAIN.
```

```
A failure to reply will be counted as an abstention. If you vote NO, by our
policy you must include the reasons for that vote in your reply. The
deadline for votes is *[date and time]*.

If a majority of committers vote YES and there are zero NO votes commit
access will be granted.
```

Vote Results for Grant of Commit Access

```
The voting period for granting to commit access to *[candidate]* initiated
at *[date and time]* is now closed with the following results:

YES: *[count of yes votes]* (*[% of voters]*)

NO: *[count of no votes]* (*[% of voters]*)

ABSTAIN: *[count of abstentions]* (*[% of voters]*)

Based on these results commit access *[is/is NOT]* granted.
```

Invitation to Accepted Committer

```
Due to your sustained contributions to the Cilium project we
would like to provide you with commit access to the project repository.
Developers with commit access must agree to fulfill specific
responsibilities described in the source repository:

    /Documentation/commit-access.rst

Please let us know if you would like to accept commit access and if so that
you agree to fulfill these responsibilities. Once we receive your response
we'll set up access. We're looking forward continuing to work together to
advance the Cilium project.
```

Proposal to Remove Commit Access for Inactivity

```
Committer *[candidate]* has been inactive for *[duration]*. I have
attempted to privately contacted *[him/her]* and *[he/she]* could not be
reached.

Based on this I would like to formally propose removal of commit access.
If a response to this message documenting the reasons to retain commit
access is not received by *[date]* access will be removed.
```

Notification of Commit Removal for Inactivity

```
Committer *[candidate]* has been inactive for *[duration]*. *[He/she]*
*[stated no commit access is required/failed to respond]* to the formal
proposal to remove access on *[date]*. Commit access has now been removed.
```

Proposal to Revoke Commit Access for Detrimental Behavior

```
I regret that I feel compelled to propose revocation of commit access for
*[candidate]*. I have privately discussed with *[him/her]* the following
reasons I believe *[his/her]* actions are detrimental to the project and we
have failed to come to a mutual understanding:

*[List of reasons and supporting evidence]*

Please reply to all in this thread with your thoughts on this proposal.  I
plan to formally propose a vote on the proposal on or after *[date and
time]*.

It is important to get all discussion points both for and against the
proposal on the table during the discussion period prior to the vote.
Please make it a high priority to respond to this proposal with your
thoughts.
```

Vote to Revoke Commit Access

```
I nominated *[candidate]* for revocation of commit access on *[date]*.
Having allowed sufficient time for discussion it's now time to formally
vote on the proposal.

Please reply to all in this thread with your vote of: YES, NO, or ABSTAIN.
A failure to reply will be counted as an abstention. If you vote NO, by our
policy you must include the reasons for that vote in your reply. The
deadline for votes is *[date and time]*.

If 2/3rds of committers vote YES commit access will be revoked.

The following reasons for revocation have been given in the original
proposal or during discussion:

*[list of reasons to remove access]*

The following reasons for retaining access were discussed:

*[list of reasons to retain access]*

The counter-argument for each reason for retaining access is:

*[list of counter-arguments for retaining access]*
```

Vote Results for Revocation of Commit Access

```
The voting period for revoking the commit access of *[candidate]* initiated
at *[date and time]* is now closed with the following results:

-  YES: *[count of yes votes]* (*[% of voters]*)

-  NO: *[count of no votes]* (*[% of voters]*)

-  ABSTAIN: *[count of abstentions]* (*[% of voters]*)
```

```
Based on these results commit access *[is/is NOT]* revoked. The following
reasons for retaining commit access were proposed in NO votes:

*[list of reasons]*

The counter-arguments for each of these reasons are:

*[list of counter-arguments]*
```

Notification of Commit Revocation for Detrimental Behavior

```
After private discussion with you and careful consideration of the
situation, the other committers to the Cilium project have
concluded that it is in the best interest of the project that your commit
access to the project repositories be revoked and this has now occurred.

The reasons for this decision are:

*[list of reasons for removing access]*

While your goals and those of the project no longer appear to be aligned we
greatly appreciate all the work you have done for the project and wish you
continued success in your future work.
```

CHAPTER 10

Glossary

Cilium has some terms with special meanings. These should all be covered throughout the documentation but for convenience we have also listed some of them below with short descriptions. If you need more information, please ask us on Slack. Feel free to extend this document with words you expected to see here.

Endpoint A Cilium endpoint is one or more application containers which can be addressed by an individual IP address.

Identity The identity of an endpoint is derived based on the labels associated with the pod or container.

Label Cilium labels are similar to regular container names / labels, the exception being that they can be key / value pairs.

Policy A Cilium policy consists of a list of rules. The security policy can be specified in The Kubernetes NetworkPolicy format or The Cilium policy language.

HTTP Routing Table