- acend gmbh

Setup

This training can be done in two ways:

- on a local machine (proceed with Local Machine Setup)
- on a provided virtual machine using the webshell (proceed with Webshell access)

Local machine setup

Technical prerequisites

To run this training on your local machine please make sure the following requirements are met:

- Operating System: Linux with Kernel >= 4.9.17 or MacOS
- Docker installed
- kubectl >= 1.24 installed
- minikube >= 1.26 installed
- helm installed
- Minimum 8GB RAM

A note on Windows with WSL2: As of August 2022 the default kernel in WSL is missing some Netfilter modules. You can compile it yourself , but the training staff cannot give you any support with cluster related issues.

Install minikube

This training uses minikube to provide a Kubernetes Cluster.

Check the minikube start Guide for instructions on how to install minikube on your system. If you are using the provided virtual machine minikube is already installed.

Install helm

For a complete overview refer to the helm installation website . If you have helm 3 already installed you can skip this step.

Use your package manager (`apt` , `yum` , `brew` etc), download the latest Release or use the following command to install helm helm:

```
curl -s https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
```

Webshell access

Your trainer will give you the necessary details.

1. Cleanup

Remove Kubernetes Cluster

You can list and then remove the minikube Kubernetes clusters with the following command

```
minikube profile list
minikube delete -p cluster1
minkube delete -p cluster2
```

- acend gmbh

Labs

The purpose of these labs is to convey Cilium basics by providing hands-on tasks for people.

Goals of these labs:

- Help you get started with this modern technology
- Explain the basic concepts to you

Additional Docs

- Cilium documentation

1. Introduction

(from https://docs.cilium.io/en/v1.9/intro/)

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

What is Hubble?

Hubble is a fully distributed networking and security observability platform. It is built on top of Cilium and eBPF to enable deep visibility into the communication and behavior of services as well as the networking infrastructure in a completely transparent manner.

By building on top of Cilium, Hubble can leverage eBPF for visibility. By relying on eBPF, all visibility is programmable and allows for a dynamic approach that minimizes overhead while providing deep and detailed visibility as required by users. Hubble has been created and specifically designed to make best use of these new eBPF powers.

Hubble can answer questions such as:

Service dependencies & communication map

- What services are communicating with each other? How frequently? What does the service dependency graph look like?
- What HTTP calls are being made? What Kafka topics does a service consume from or produce to?

Network monitoring & alerting

- Is any network communication failing? Why is communication failing? Is it DNS? Is it an application or network problem? Is the communication broken on layer 4 (TCP) or layer 7 (HTTP)?
- Which services have experienced a DNS resolution problem in the last 5 minutes? Which services have experienced an interrupted TCP connection recently or have seen connections timing out? What is the rate of unanswered TCP SYN requests?

Application monitoring

- What is the rate of 5xx or 4xx HTTP response codes for a particular service or across all clusters?
- What is the 95th and 99th percentile latency between HTTP requests and responses in my cluster? Which services are performing the worst? What is the latency between two services?

Security observability

- Which services had connections blocked due to network policy? What services have been accessed from outside the cluster? Which services have resolved a particular DNS name?

Why Cilium & Hubble?

eBPF is enabling visibility into and control over systems and applications at a granularity and efficiency that was not possible before. It does so in a completely transparent way, without requiring the application to change in any way. eBPF is equally well-equipped to handle modern containerized workloads as well as more traditional workloads such as virtual machines and standard Linux processes.

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 vehicles which may have a drastically reduced lifetime of just a few seconds in microservices architectures.

By leveraging Linux eBPF, 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 eBPF enables Cilium to achieve all of this in a way that is highly scalable even for large-scale environments.

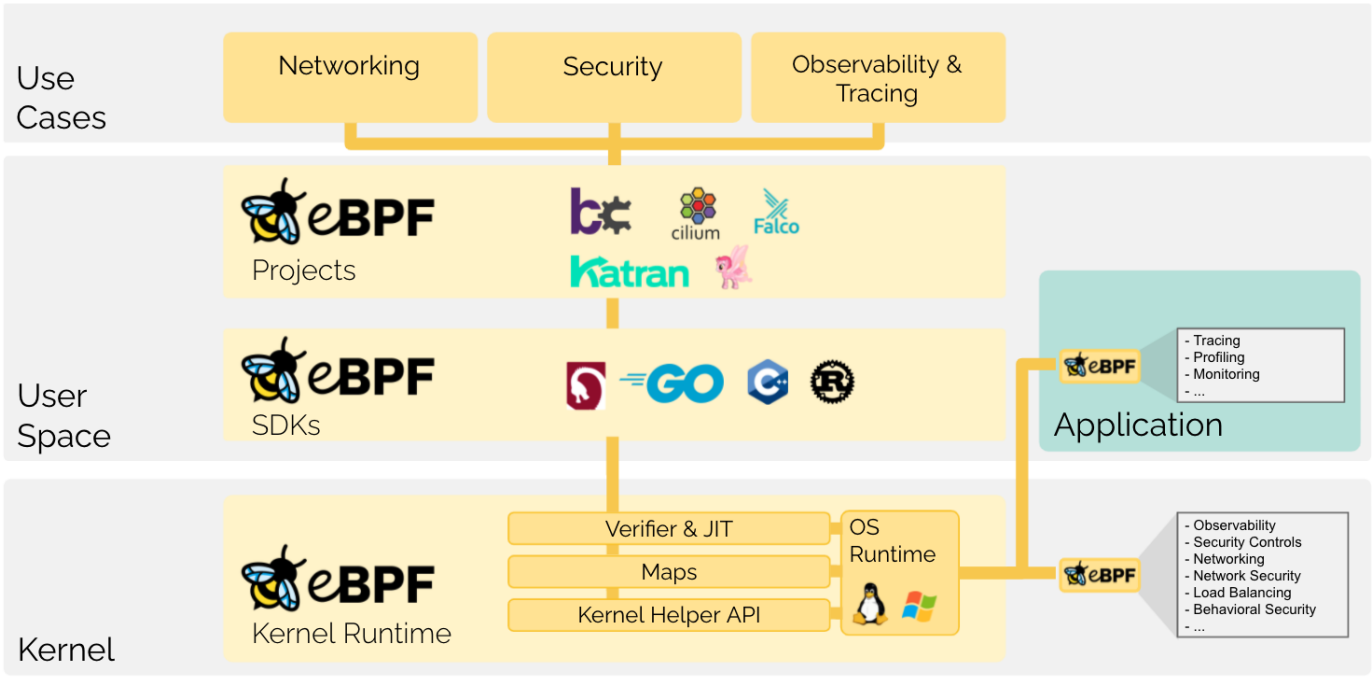
1.1. eBPF

What is eBPF

(from https://ebpf.io/)

eBPF is a revolutionary technology with origins in the Linux kernel that can run sandboxed programs in an operating system kernel. It is used to safely and efficiently extend the capabilities of the kernel without requiring to change kernel source code or load kernel modules.

Historically, the operating system has always been an ideal place to implement observability, security, and networking functionality due to the kernel’s privileged ability to oversee and control the entire system. At the same time, an operating system kernel is hard to evolve due to its central role and high requirement towards stability and security. The rate of innovation at the operating system level has thus traditionally been lower compared to functionality implemented outside of the operating system.

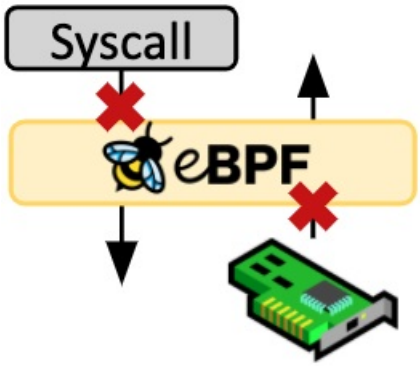


eBPF changes this formula fundamentally. By allowing to run sandboxed programs within the operating system, application developers can run eBPF programs to add additional capabilities to the operating system at runtime. The operating system then guarantees safety and execution efficiency as if natively compiled with the aid of a Just-In-Time (JIT) compiler and verification engine. This has led to a wave of eBPF-based projects covering a wide array of use cases, including next-generation networking, observability, and security functionality.

Today, eBPF is used extensively to drive a wide variety of use cases: Providing high-performance networking and load-balancing in modern data centers and cloud native environments, extracting fine-grained security observability data at low overhead, helping application developers trace applications, providing insights for performance troubleshooting, preventive application and container runtime security enforcement, and much more. The possibilities are endless, and the innovation that eBPF is unlocked has only just begun.

Security

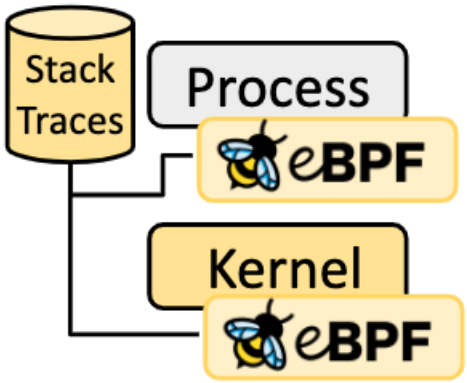
Building on the foundation of seeing and understanding all system calls and combining that with a packet and socket-level view of all networking operations allows for revolutionary new approaches to securing systems. While aspects of system call filtering, network-level filtering, and process context tracing have typically been handled by completely independent systems, eBPF allows for combining the visibility and control of all aspects to create security systems operating on more context with a better level of control.



- acend gmbh

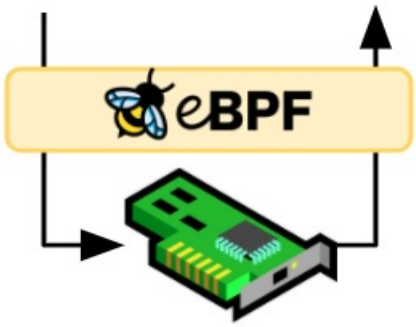
Tracing & Profiling

The ability to attach eBPF programs to tracepoints as well as kernel and user application probe points allows unprecedented visibility into the runtime behavior of applications and the system itself. By giving introspection abilities to both the application and system side, both views can be combined, allowing powerful and unique insights to troubleshoot system performance problems. Advanced statistical data structures allow extracting meaningful visibility data efficiently, without requiring the export of vast amounts of sampling data as typically done by similar systems.



Networking

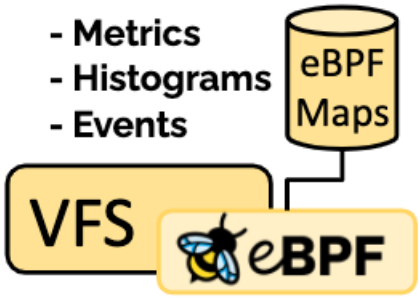
The combination of programmability and efficiency makes eBPF a natural fit for all packet processing requirements of networking solutions. The programmability of eBPF enables adding additional protocol parsers and easily programming any forwarding logic to meet changing requirements without ever leaving the packet processing context of the Linux kernel. The efficiency provided by the JIT compiler provides execution performance close to that of natively compiled in-kernel code.



Observability & Monitoring

Instead of relying on static counters and gauges exposed by the operating system, eBPF enables the collection & in-kernel aggregation of custom metrics and generation of visibility events based on a wide range of possible sources. This extends the depth of visibility that can be achieved as well as reduces the overall system overhead significantly by only collecting the visibility data required and by generating histograms and similar data structures at the source of the event instead of relying on the export of samples.

- acend gmbh



Featured eBPF Talks

2. Install Cilium

2.1. Install Cilium

Cilium can be installed using multiple ways:

- Cilium CLI
- Using Helm

In this lab, we are going to use Helm which is recommended for production use. The Cilium command-line tool is used (Cilium CLI) for verification and troubleshooting.

Task 2.1.1: Install a Kubernetes Cluster

We are going to spin up a new Kubernetes cluster with the following command:

> **Note**
> To start from a clean Kubernetes cluster, make sure `cluster1` is not yet available. You can verify this with `minikube profile list`. If you already have a `cluster1` you can delete the cluster with `minikube delete -p cluster1`.

```
minikube start --network-plugin=cni --cni=false --kubernetes-version=1.24.3 -p cluster1
```

> **Note**
> During this training, you will create multiple clusters. For this, we use a feature in Minikube called profile which you see with the `-p cluster1` option. You can list all your profiles with `minikube profile list` and you can change to another cluster with `minikube profile <profilename>`, this will also set your current context for `kubectl` to the specified profile/cluster.

Minikube installed a new Kubernetes cluster without any Container Network Interface (CNI). CNI installation happens in the next task.

Minikube added a new context to your Kubernetes config file and set this as default. Verify this with the following command:

```
kubectl config current-context
```

This should show `cluster1`. Now check that everything is up and running:

```
kubectl get node
```

This should produce a similar output:

```
NAME       STATUS   ROLES                  AGE   VERSION
cluster1   Ready    control-plane,master   86s   v1.24.3
```

Depending on your Minikube version and environment your node might stay NotReady because no CNI is installed. After we installed Cilium it will become ready.

Check if all pods are running with:

```
kubectl get pod -A
```

which produces the following output

```
NAMESPACE     NAME                                READY   STATUS              RESTARTS   AGE
kube-system   coredns-6d4b75cb6d-nf8wz            0/1     ContainerCreating   0          3m1s
kube-system   etcd-cluster1                       1/1     Running             0          3m7s
kube-system   kube-apiserver-cluster1             1/1     Running             0          3m16s
kube-system   kube-controller-manager-cluster1    1/1     Running             0          3m7s
kube-system   kube-proxy-7l6qk                    1/1     Running             0          3m1s
kube-system   kube-scheduler-cluster1             1/1     Running             0          3m7s
kube-system   storage-provisioner                 1/1     Running             0          3m11s
```

> **Note**
> Depending on your Minikube version, coredns might start or not which is ok. But you should not see any CNI related pods!

Task 2.1.2: Install Cilium CLI

The `cilium` CLI tool is a single binary file that can be downloaded from the project's release page. Follow the instructions depending on your operating system or environment.

Linux/Webshell Setup

> **Note**
> If you are working in our webshell based lab setup, please always follow the Linux setup.

Execute the following command to download the `cilium` CLI:

```
curl -L --remote-name-all https://github.com/cilium/cilium-cli/releases/download/v0.12.12/cilium-linux-amd64.tar.gz{,.s
ha256sum}
sha256sum --check cilium-linux-amd64.tar.gz.sha256sum
sudo tar xzvfC cilium-linux-amd64.tar.gz /usr/local/bin
rm cilium-linux-amd64.tar.gz{,.sha256sum}
```

macOS Setup

Execute the following command to download the `cilium` CLI:

```
curl -L --remote-name-all https://github.com/cilium/cilium-cli/releases/download/v0.12.12/cilium-darwin-amd64.tar.gz{,.
sha256sum}
shasum -a 256 -c cilium-darwin-amd64.tar.gz.sha256sum
sudo tar xzvfC cilium-darwin-amd64.tar.gz /usr/local/bin
rm cilium-darwin-amd64.tar.gz{,.sha256sum}
```

Cilium CLI

Now that we have the `cilium` CLI let's have a look at some commands:

```
cilium version
```

which should give you an output similar to this:

```
cilium-cli: v0.12.12 compiled with go1.19.4 on linux/amd64
cilium image (default): v1.12.5
cilium image (stable): v1.12.5
cilium image (running): unknown. Unable to obtain cilium version, no cilium pods found in namespace "kube-system"
```

> **Note**
> It's ok if your installation does not show the same version.

Then let us look at

```
cilium status
```

```
cilium status
    /¯¯\
 /¯¯\__/¯¯\    Cilium:         1 errors
 \__/¯¯\__/    Operator:       disabled
 /¯¯\__/¯¯\    Hubble:         disabled
 \__/¯¯\__/    ClusterMesh:    disabled
    \__/

Containers:      cilium
                 cilium-operator
Cluster Pods:    0/0 managed by Cilium
Errors:          cilium    cilium    daemonsets.apps "cilium" not found
```

We don't have yet installed Cilium, therefore the error is perfectly fine.

Task 2.1.3: Install Cilium

Let's install Cilium with Helm. First we need to add the Cilium Helm repository:

```
helm repo add cilium https://helm.cilium.io/ --force-update
```

and then we can install Cilium:

```
helm upgrade -i cilium cilium/cilium --version 1.11.7 \
  --namespace kube-system \
  --set ipam.operator.clusterPoolIPv4PodCIDR=10.1.0.0/16 \
  --set cluster.name=cluster1 \
  --set cluster.id=1 \
  --set operator.replicas=1 \
  --set kubeProxyReplacement=disabled \
  --wait
```

Note

You will see a deprecation warning for beta.kubernetes.io/os, this can be ignored for now.

For all values possible in the Cilium Helm chart, have a look at the Repository or the Helm Reference in Cilium's documentation. We disable the kubeProxyReplacement because it would cause problems with multiple clusters running on the same kernel in the later chapters.

and now run again the

```
cilium status --wait
```

command:

```
cilium status
    /¯¯\
 /¯¯\__/¯¯\    Cilium:         OK
 \__/¯¯\__/    Operator:       OK
 /¯¯\__/¯¯\    Hubble:         disabled
 \__/¯¯\__/    ClusterMesh:    disabled
    \__/

DaemonSet         cilium             Desired: 1, Ready: 1/1, Available: 1/1
Deployment        cilium-operator    Desired: 1, Ready: 1/1, Available: 1/1
Containers:       cilium             Running: 1
                  cilium-operator    Running: 1
Cluster Pods:     1/1 managed by Cilium
Image versions    cilium             quay.io/cilium/cilium:v1.11.7: 1
                  cilium-operator    quay.io/cilium/operator-generic:v1.11.7: 1
```

Note

If the output is not the same, make sure all Cilium container are up and in a ready state.

Take a look at the pods again to see what happened under the hood:

```
kubectl get pods -A
```

and you should see now the Pods related to Cilium:

```
NAMESPACE     NAME                                READY   STATUS    RESTARTS      AGE
kube-system   cilium-operator-77577756b6-ksnbw    1/1     Running   0             58s
kube-system   cilium-q4p6q                        1/1     Running   0             58s
kube-system   coredns-6d4b75cb6d-nf8wz            1/1     Running   0             2m42s
kube-system   etcd-cluster1                       1/1     Running   0             2m54s
kube-system   kube-apiserver-cluster1             1/1     Running   0             2m54s
kube-system   kube-controller-manager-cluster1    1/1     Running   0             2m54s
kube-system   kube-proxy-7l6qk                    1/1     Running   0             2m42s
kube-system   kube-scheduler-cluster1             1/1     Running   0             2m54s
kube-system   storage-provisioner                 1/1     Running   1 (2m11s ago)   2m53s
```

Note

It might take some time until all Pods ar in state `Runnning` and `READY`. Wait before continue.

Alright, Cilium is up and running, let us make some tests. The `cilium` CLI allows you to run a connectivity test:

```
cilium connectivity test
```

This will run for some minutes, let's wait.

Note

As we installed an older version of cilium but are using the latest `cilium` CLI, it's ok if some tests are failing.

```
ℹ  Single-node environment detected, enabling single-node connectivity test
ℹ  Monitor aggregation detected, will skip some flow validation steps
  [cluster1] Creating namespace cilium-test for connectivity check...
  [cluster1] Deploying echo-same-node service...
  [cluster1] Deploying DNS test server configmap...
  [cluster1] Deploying same-node deployment...
  [cluster1] Deploying client deployment...
  [cluster1] Deploying client2 deployment...
  [cluster1] Waiting for deployments [client client2 echo-same-node] to become ready...
  [cluster1] Waiting for CiliumEndpoint for pod cilium-test/client-755fb678bd-hfd8w to appear...
  [cluster1] Waiting for CiliumEndpoint for pod cilium-test/client2-5b97d7bc66-5cfsm to appear...
  [cluster1] Waiting for pod cilium-test/client-755fb678bd-hfd8w to reach DNS server on cilium-test/echo-same-node-6477
4c64d5-rmj25 pod...
  [cluster1] Waiting for pod cilium-test/client2-5b97d7bc66-5cfsm to reach DNS server on cilium-test/echo-same-node-647
74c64d5-rmj25 pod...
  [cluster1] Waiting for pod cilium-test/client-755fb678bd-hfd8w to reach default/kubernetes service...
  [cluster1] Waiting for pod cilium-test/client2-5b97d7bc66-5cfsm to reach default/kubernetes service...
  [cluster1] Waiting for CiliumEndpoint for pod cilium-test/echo-same-node-64774c64d5-rmj25 to appear...
  [cluster1] Waiting for Service cilium-test/echo-same-node to become ready...
  [cluster1] Waiting for NodePort 192.168.49.2:30598 (cilium-test/echo-same-node) to become ready...
ℹ  Skipping IPCache check
  Enabling Hubble telescope...
⚠  Unable to contact Hubble Relay, disabling Hubble telescope and flow validation: rpc error: code = Unavailable desc =
  connection error: desc = "transport: Error while dialing dial tcp 127.0.0.1:4245: connect: connection refused"
```

```
 connection error: desc = "transport: Error while dialing dial tcp 127.0.0.1:4245: connect: connection refused
ℹ  Expose Relay locally with:
   cilium hubble enable
   cilium hubble port-forward&
ℹ  Cilium version: 1.11.7
  Running tests...
[=] Test [no-policies]
....................
[=] Test [allow-all-except-world]
........
[=] Test [client-ingress]
..
[=] Test [all-ingress-deny]
......
[=] Test [all-egress-deny]
........
[=] Test [all-entities-deny]
......
[=] Test [cluster-entity]
..
[=] Test [host-entity]
..
[=] Test [echo-ingress]
..

[=] Skipping Test [client-ingress-icmp]
[=] Test [client-egress]
..
[=] Test [client-egress-expression]
..
[=] Test [client-egress-to-echo-service-account]
..
[=] Test [to-entities-world]
......
[=] Test [to-cidr-1111]
....
[=] Test [echo-ingress-from-other-client-deny]
....

[=] Skipping Test [client-ingress-from-other-client-icmp-deny]
[=] Test [client-egress-to-echo-deny]
....
[=] Test [client-ingress-to-echo-named-port-deny]
..
[=] Test [client-egress-to-echo-expression-deny]
..
[=] Test [client-egress-to-echo-service-account-deny]
..
[=] Test [client-egress-to-cidr-deny]
....
[=] Test [client-egress-to-cidr-deny-default]
....
[=] Test [health]
.
[=] Test [echo-ingress-l7]
......
[=] Test [echo-ingress-l7-named-port]
......
[=] Test [client-egress-l7-method]
......
[=] Test [client-egress-l7]
........
[=] Test [client-egress-l7-named-port]
........
[=] Test [dns-only]
........
[=] Test [to-fqdns]
........

  All 29 tests (145 actions) successful, 2 tests skipped, 1 scenarios skipped.
```

Once done, clean up the connectivity test Namespace:

```
kubectl delete ns cilium-test --wait=false
```

Task 2.1.4: Explore your installation

We have learned about the Cilium components. Let us check out the installed CRDs now:

```
kubectl api-resources | grep cilium
```

Which shows CRDs installed by Cilium:

```
ciliumclusterwidenetworkpolicies   ccnp           cilium.io/v2                         false        CiliumClusterwide
NetworkPolicy
ciliumendpoints                    cep,ciliumep   cilium.io/v2                         true         CiliumEndpoint
ciliumexternalworkloads            cew            cilium.io/v2                         false        CiliumExternalWor
kload
ciliumidentities                   ciliumid       cilium.io/v2                         false        CiliumIdentity
ciliumnetworkpolicies              cnp,ciliumnp   cilium.io/v2                         true         CiliumNetworkPoli
cy
ciliumnodes                        cn,ciliumn     cilium.io/v2                         false        CiliumNode
```

And now we check all installed Cilium CRDs

```
kubectl get ccnp,cep,cew,ciliumid,cnp,cn -A
```

We see 1 node, 1 identity and 1 endpoint:

```
NAMESPACE     NAME                                                ENDPOINT ID   IDENTITY ID   INGRESS ENFORCEMENT   EGRE
SS ENFORCEMENT   VISIBILITY POLICY   ENDPOINT STATE   IPV4          IPV6
kube-system   ciliumendpoint.cilium.io/coredns-64897985d-7485t   465           67688
                                    ready            10.1.0.215

NAMESPACE   NAME                              NAMESPACE     AGE
            ciliumidentity.cilium.io/67688   kube-system   18m

NAMESPACE   NAME                             AGE
            ciliumnode.cilium.io/cluster1   18m
```

Note

It might be possible that you still see identites created by `cilium connectivity test`. They will be deleted by `cilium-operator` after max. 15 minutes.

▶ Can you guess why only the coredns Pod is listed as an Endpoint and Identity?

▶ Is it possible to have more CiliumNodes than nodes in a Kubernetes cluster?

We have discussed CNI plugin installations, let us check out the Cilium installation on the node.

We can either start a debug container on the node and chroot its /

```
kubectl debug node/cluster1 -it --image=busybox
```

```
chroot /host
```

or we use docker to access the node:

```
docker exec -it cluster1 bash
```

Now we have a shell with access to the node. We will check out the Cilium installation:

```
ls -l /etc/cni/net.d/
cat /etc/cni/net.d/05-cilium.conf
/opt/cni/bin/cilium-cni --help
ip a
exit #exit twice if you used kubectl debug
```

We make a few oberservations:

- Kubernetes uses the configuration file with the lowest number so it takes Cilium with the prefix 05.
- The configuration file is written as a CNI spec .
- The `cilium` binary was installed to /opt/cni/bin.
- Cilium created a virtual network interfaces pair `cilium_net` , `cilium_host` and the vxlan overlay interface `cilium_vxlan` .
- We see the virtual network interface (`lxc` device) of the coredns pod (the Endpoint in Cilium terms).

Install Cilium with the `cilium` cli

This is what the installation with the `cilium` cli would have looked like:

```
# cilium install --config cluster-pool-ipv4-cidr=10.1.0.0/16 --cluster-name cluster1 --cluster-id 1 --version 1.11.7
```

Be careful to never use CLI and Helm together to install, this can break an already running Cilium installation.

After this initial installation, we can proceed by upgrading to a newer version in the next lab.

2.2. Upgrade Cilium

In the previous lab, we intentionally installed version `v1.11.7` of Cilium. In this lab, we show you how to upgrade this installation.

Task 2.2.1: Running pre-flight check

When rolling out an upgrade with Kubernetes, Kubernetes will first terminate the Pod followed by pulling the new image version and then finally spin up the new image. In order to reduce the downtime of the agent and to prevent `ErrImagePull` errors during the upgrade, the pre-flight check pre-pulls the new image version. If you are running in "Kubernetes Without kube-proxy" mode you must also pass on the Kubernetes API Server IP and/or the Kubernetes API Server Port when generating the `cilium-preflight.yaml` file.

```
helm install cilium-preflight cilium/cilium --version 1.12.10 \
  --namespace=kube-system \
  --set preflight.enabled=true \
  --set agent=false \
  --set operator.enabled=false \
  --wait
```

Task 2.2.2: Clean up pre-flight check

To check the preflight Pods we check if the pods are `READY` using:

```
kubectl get pod -A | grep cilium-pre-flight
```

and you should get an output like this:

```
kube-system   cilium-pre-flight-check-84f67b54f6-hz57g   1/1     Running   0              63s
kube-system   cilium-pre-flight-check-skglp              1/1     Running   0              63s
```

The pods are `READY` with a value of `1/1` and therefore we can delete the `cilium-preflight` release again with:

```
helm delete cilium-preflight --namespace=kube-system
```

Task 2.2.3: Upgrading Cilium

During normal cluster operations, all Cilium components should run the same version. Upgrading just one of them (e.g., upgrading the agent without upgrading the operator) could result in unexpected cluster behavior. The following steps will describe how to upgrade all of the components from one stable release to a later stable release.

When upgrading from one minor release to another minor release, for example 1.x to 1.y, it is recommended to upgrade to the latest patch release for a Cilium release series first. The latest patch releases for each supported version of Cilium are here . Upgrading to the latest patch release ensures the most seamless experience if a rollback is required following the minor release upgrade. The upgrade guides for previous versions can be found for each minor version at the bottom left corner.

Helm can be used to either upgrade Cilium directly or to generate a new set of YAML files that can be used to upgrade an existing deployment via kubectl. By default, Helm will generate the new templates using the default values files packaged with each new release. You still need to ensure that you are specifying the equivalent options as used for the initial deployment, either by specifying them at the command line or by committing the values to a YAML file.

To minimize datapath disruption during the upgrade, the `upgradeCompatibility` option should be set to the initial Cilium version which was installed in this cluster.

```
helm upgrade -i cilium cilium/cilium --version 1.12.10 \
  --namespace kube-system \
  --set ipam.operator.clusterPoolIPv4PodCIDRList={10.1.0.0/16} \
  --set cluster.name=cluster1 \
  --set cluster.id=1 \
  --set operator.replicas=1 \
  --set kubeProxyReplacement=disabled \
  --set upgradeCompatibility=1.11 \
  --wait
```

Note

When upgrading from one minor release to another minor release using `helm upgrade` , do not use Helm's `--reuse-values` flag. The `--reuse-values` flag ignores any newly introduced values present in the new release and thus may cause the Helm template to render incorrectly. Instead, if you want to reuse the values from your existing installation, save the old values in a values file, check the file for any renamed or deprecated values, and then pass it to the `helm upgrade` command as described above. You can retrieve and save the values from an existing installation with the following command:

```
helm get values cilium --namespace=kube-system -o yaml > old-values.yaml
```

The `--reuse-values` flag may only be safely used if the Cilium chart version remains unchanged, for example when `helm upgrade` is used to apply configuration changes without upgrading Cilium.

Task 2.2.4: Explore your installation after the upgrade

We can run:

```
cilium status --wait
```

again to verify the upgrade to the new version succeded

- acend gmbh

```
    /¯¯\
 /¯¯\__/¯¯\    Cilium:         OK
 \__/¯¯\__/    Operator:       OK
 /¯¯\__/¯¯\    Hubble:         disabled
 \__/¯¯\__/    ClusterMesh:    disabled
    \__/

DaemonSet         cilium             Desired: 1, Ready: 1/1, Available: 1/1
Deployment        cilium-operator    Desired: 1, Ready: 1/1, Available: 1/1
Containers:       cilium-operator    Running: 1
                  cilium             Running: 1
Cluster Pods:     1/1 managed by Cilium
Image versions    cilium             quay.io/cilium/cilium:v1.12.10:: 1
                  cilium-operator    quay.io/cilium/operator-generic:v1.12.10@: 1
```

And we see the right version in the `cilium` and `cilium-operator` images.

Nice to know

In Cilium release 1.11.0 automatic mount of eBPF maps in the host filesystem were enabled. These eBPF maps are basically very efficient key-value stores used by Cilium. Having them mounted in the filesystem, allows the datapath to continue operating even if the `cilium-agent` is restarting. We can verify that Cilium created global traffic control eBPF maps on the node in /sys/fs/bpf/tc/globals/:

```
docker exec cluster1 ls /sys/fs/bpf/tc/globals/
```

Rolling Back

Occasionally, it may be necessary to undo the rollout because a step was missed or something went wrong during the upgrade. To undo the rollout run:

```
helm history cilium --namespace=kube-system
# helm rollback cilium [REVISION] --namespace=kube-system
```

This will revert the latest changes to the Cilium DaemonSet and return Cilium to the state it was in prior to the upgrade.

- acend gmbh

3. Hubble

3.1. Hubble

Before we start with the CNI functionality of Cilium and its security components we want to enable the optional Hubble component (which is disabled by default). So we can take full advantage of its eBFP observability capabilities.

Task 3.1.1: Install the Hubble CLI

Similar to the `cilium` CLI, the `hubble` CLI interfaces with Hubble and allows observing network traffic within Kubernetes.

So let us install the `hubble` CLI.

Linux/Webshell Setup

Execute the following command to download the `hubble` CLI:

```
curl -L --remote-name-all https://github.com/cilium/hubble/releases/download/v0.11.1/hubble-linux-amd64.tar.gz{,.sha256
sum}
sha256sum --check hubble-linux-amd64.tar.gz.sha256sum
sudo tar xzvfC hubble-linux-amd64.tar.gz /usr/local/bin
rm hubble-linux-amd64.tar.gz{,.sha256sum}
```

macOS Setup

Execute the following command to download the `hubble` CLI:

```
curl -L --remote-name-all https://github.com/cilium/hubble/releases/download/v0.11.1/hubble-darwin-amd64.tar.gz{,.sha25
6sum}
shasum -a 256 -c hubble-darwin-amd64.tar.gz.sha256sum
sudo tar xzvfC hubble-darwin-amd64.tar.gz /usr/local/bin
rm hubble-darwin-amd64.tar.gz{,.sha256sum}
```

Hubble CLI

Now that we have the `hubble` CLI let's have a look at some commands:

```
hubble version
```

should show

```
hubble 0.11.1 compiled with go1.19.5 on linux/amd64
```

or

```
hubble help
```

should show

```
Hubble is a utility to observe and inspect recent Cilium routed traffic in a cluster.

Usage:
  hubble [command]

Available Commands:
  completion  Output shell completion code
  config      Modify or view hubble config
  help        Help about any command
  list        List Hubble objects
  observe     Observe flows of a Hubble server
  status      Display status of Hubble server
  version     Display detailed version information

Global Flags:
      --config string   Optional config file (default "/home/user/.config/hubble/config.yaml")
  -D, --debug           Enable debug messages

Get help:
  -h, --help    Help for any command or subcommand

Use "hubble [command] --help" for more information about a command.
```

Task 3.1.2: Deploy a simple application

Before we enable Hubble in Cilium we want to make sure we have at least one application to observe.

Let’s have a look at the following resource definitions:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  labels:
    app: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend-container
        image: docker.io/byrnedo/alpine-curl:0.1.8
        imagePullPolicy: IfNotPresent
        command: [ "/bin/ash", "-c", "sleep 1000000000" ]
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: not-frontend
  labels:
```

```
    app: not-frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: not-frontend
  template:
    metadata:
      labels:
        app: not-frontend
    spec:
      containers:
      - name: not-frontend-container
        image: docker.io/byrnedo/alpine-curl:0.1.8
        imagePullPolicy: IfNotPresent
        command: [ "/bin/ash", "-c", "sleep 1000000000" ]
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
  labels:
    app: backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: backend-container
        env:
        - name: PORT
          value: "8080"
        ports:
        - containerPort: 8080
        image: docker.io/cilium/json-mock:1.2
        imagePullPolicy: IfNotPresent
---
apiVersion: v1
kind: Service
metadata:
  name: backend
  labels:
    app: backend
spec:
  type: ClusterIP
  selector:
    app: backend
  ports:
  - name: http
    port: 8080
```

The application consists of two client deployments (`frontend` and `not-frontend`) and one backend deployment (`backend`). We are going to send requests from the frontend and not-frontend pods to the backend pod.

Create a file `simple-app.yaml` with the above content.

Deploy the app:

```
kubectl apply -f simple-app.yaml
```

this gives you the following output:

- acend gmbh

```
deployment.apps/frontend created
deployment.apps/not-frontend created
deployment.apps/backend created
service/backend created
```

Verify with the following command that everything is up and running:

```
kubectl get all,cep,ciliumid
```

```
NAME                                READY   STATUS    RESTARTS   AGE
pod/backend-65f7c794cc-b9j66        1/1     Running   0          3m17s
pod/frontend-76fbb99468-mbzcm       1/1     Running   0          3m17s
pod/not-frontend-8f467ccbd-cbks8    1/1     Running   0          3m17s

NAME                 TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
service/backend      ClusterIP   10.97.228.29   <none>        8080/TCP   3m17s
service/kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP    45m

NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/backend         1/1     1            1           3m17s
deployment.apps/frontend        1/1     1            1           3m17s
deployment.apps/not-frontend    1/1     1            1           3m17s

NAME                                       DESIRED   CURRENT   READY   AGE
replicaset.apps/backend-65f7c794cc         1         1         1       3m17s
replicaset.apps/frontend-76fbb99468        1         1         1       3m17s
replicaset.apps/not-frontend-8f467ccbd     1         1         1       3m17s

NAME                                                 ENDPOINT ID   IDENTITY ID   INGRESS ENFORCEMENT   EGRESS ENFORC
EMENT   VISIBILITY POLICY   ENDPOINT STATE   IPV4          IPV6
ciliumendpoint.cilium.io/backend-65f7c794cc-b9j66    144           67823
                            ready            10.1.0.44
ciliumendpoint.cilium.io/frontend-76fbb99468-mbzcm   1898          76556
                            ready            10.1.0.161
ciliumendpoint.cilium.io/not-frontend-8f467ccbd-cbks8   208           127021
                            ready            10.1.0.128

NAME                              NAMESPACE     AGE
ciliumidentity.cilium.io/127021   default       3m15s
ciliumidentity.cilium.io/67688    kube-system   41m
ciliumidentity.cilium.io/67823    default       3m15s
ciliumidentity.cilium.io/76556    default       3m15s
```

Let us make life a bit easier by storing the pods name into an environment variable so we can reuse it later again:

```
export FRONTEND=$(kubectl get pods -l app=frontend -o jsonpath='{.items[0].metadata.name}')
echo ${FRONTEND}
export NOT_FRONTEND=$(kubectl get pods -l app=not-frontend -o jsonpath='{.items[0].metadata.name}')
echo ${NOT_FRONTEND}
```

Task 3.1.3: Enable Hubble in Cilium

When you install Cilium using Helm, then Hubble is already enabled. The value for this is `hubble.enabled` which is set to `true` in the `values.yaml` of the Cilium Helm Chart. But we also want to enable Hubble Relay. With the following Helm command you can enable Hubble with Hubble Relay:

```
helm upgrade -i cilium cilium/cilium --version 1.12.10 \
  --namespace kube-system \
  --set ipam.operator.clusterPoolIPv4PodCIDRList={10.1.0.0/16} \
  --set cluster.name=cluster1 \
  --set cluster.id=1 \
  --set operator.replicas=1 \
  --set upgradeCompatibility=1.11 \
  --set kubeProxyReplacement=disabled \
  `# hubble and hubble relay variables:` \
  --set hubble.enabled=true \
  --set hubble.relay.enabled=true \
  --wait
```

If you have installed Cilium with the `cilium` CLI then Hubble component is not enabled by default (nor is Hubble Relay). You can enable Hubble using the following `cilium` CLI command:

```
# cilium hubble enable
```

and then wait until Hubble is enabled:

```
  Found existing CA in secret cilium-ca
  Patching ConfigMap cilium-config to enable Hubble...
♻  Restarted Cilium pods
  Waiting for Cilium to become ready before deploying other Hubble component(s)...
  Generating certificates for Relay...
  Deploying Relay from quay.io/cilium/hubble-relay:v1.12.10...
  Waiting for Hubble to be installed...
  Hubble was successfully enabled!
```

When you have a look at your running pods with `kubectl get pod -A` you should see a Pod with a name starting with `hubble-relay` :

```
kubectl get pod -A
```

```
NAMESPACE     NAME                                READY   STATUS    RESTARTS      AGE
default       backend-6f884b6495-v7bvt            1/1     Running   0             52s
default       frontend-77d99ffc5d-lcsph           1/1     Running   0             52s
default       not-frontend-7db9747986-snjwp       1/1     Running   0             52s
kube-system   cilium-ksr7h                        1/1     Running   0             9m16s
kube-system   cilium-operator-6f5c6f768d-r2qgn    1/1     Running   0             9m17s
kube-system   coredns-6d4b75cb6d-nf8wz            1/1     Running   0             22m
kube-system   etcd-cluster1                       1/1     Running   0             22m
kube-system   hubble-relay-84b4ddb556-nr7c8       1/1     Running   0             10s
kube-system   kube-apiserver-cluster1             1/1     Running   0             22m
kube-system   kube-controller-manager-cluster1    1/1     Running   0             22m
kube-system   kube-proxy-7l6qk                    1/1     Running   0             22m
kube-system   kube-scheduler-cluster1             1/1     Running   0             22m
kube-system   storage-provisioner                 1/1     Running   1 (21m ago)   22m
```

Cilium agents are restarting, and a new Hubble Relay pod is now present. We can wait for Cilium and Hubble to be ready by running:

```
cilium status --wait
```

which should give you an output similar to this:

```
    /¯¯\
 /¯¯\__/¯¯\    Cilium:         OK
 \__/¯¯\__/    Operator:       OK
 /¯¯\__/¯¯\    Hubble:         OK
 \__/¯¯\__/    ClusterMesh:    disabled
    \__/

DaemonSet         cilium             Desired: 1, Ready: 1/1, Available: 1/1
Deployment        cilium-operator    Desired: 1, Ready: 1/1, Available: 1/1
Deployment        hubble-relay       Desired: 1, Ready: 1/1, Available: 1/1
Containers:       cilium             Running: 1
                  cilium-operator    Running: 1
                  hubble-relay       Running: 1
Cluster Pods:     9/9 managed by Cilium
Image versions    cilium             quay.io/cilium/cilium:v1.11.2@sha256:ea677508010800214b0b5497055f38ed3bff57963fa23
99bcb1c69cf9476453a: 1
                  cilium-operator    quay.io/cilium/operator-generic:v1.11.2@sha256:b522279577d0d5f1ad7cadaacb7321d1b17
2d8ae8c8bc816e503c897b420cfe3: 1
                  hubble-relay       quay.io/cilium/hubble-relay:v1.11.2@sha256:306ce38354a0a892b0c175ae7013cf178a46b79
f51c52adb5465d87f14df0838: 1
```

Hubble is now enabled. We can now locally port-forward to the Hubble pod:

```
cilium hubble port-forward&
```

Note

The port-forwarding is needed as the hubble Kubernetes service is only a `ClusterIP` service and not exposed outside of the cluster network. With the port-forwarding you can access the hubble service from your localhost.

Note

Note the `&` after the command which puts the process in the background so we can continue working in the shell.

And then check Hubble status via the Hubble CLI (which uses the port-forwarding just opened):

```
hubble status
```

```
Healthcheck (via localhost:4245): Ok
Current/Max Flows: 947/4095 (23.13%)
Flows/s: 3.84
Connected Nodes: 1/1
```

Note

If the nodes are not yet connected, give it some time and try again. There is a Certificate Authority thats first needs to be fully loaded by the components.

The Hubble CLI is now primed for observing network traffic within the cluster.

Task 3.1.4: Observing flows with Hubble

We now want to use the `hubble` CLI to observe some network flows in our Kubernetes cluster. Let us have a look at the following command:

```
hubble observe
```

which gives you a list of network flows:

```
Nov 23 14:49:03.030: 10.0.0.113:46274 <- kube-system/hubble-relay-f6d85866c-csthd:4245 to-stack FORWARDED (TCP Flags: A
CK, PSH)
Nov 23 14:49:03.030: 10.0.0.113:46274 -> kube-system/hubble-relay-f6d85866c-csthd:4245 to-endpoint FORWARDED (TCP Flags
: RST)
Nov 23 14:49:04.011: 10.0.0.113:44840 <- 10.0.0.114:4240 to-stack FORWARDED (TCP Flags: ACK)
Nov 23 14:49:04.011: 10.0.0.113:44840 -> 10.0.0.114:4240 to-endpoint FORWARDED (TCP Flags: ACK)
Nov 23 14:49:04.226: 10.0.0.113:32898 -> kube-system/coredns-558bd4d5db-xzvc9:8080 to-endpoint FORWARDED (TCP Flags: SY
N)
Nov 23 14:49:04.226: 10.0.0.113:32898 <- kube-system/coredns-558bd4d5db-xzvc9:8080 to-stack FORWARDED (TCP Flags: SYN,
ACK)
Nov 23 14:49:04.227: 10.0.0.113:32898 -> kube-system/coredns-558bd4d5db-xzvc9:8080 to-endpoint FORWARDED (TCP Flags: AC
K)
Nov 23 14:49:04.227: 10.0.0.113:32898 -> kube-system/coredns-558bd4d5db-xzvc9:8080 to-endpoint FORWARDED (TCP Flags: AC
K, PSH)
Nov 23 14:49:04.227: 10.0.0.113:32898 <- kube-system/coredns-558bd4d5db-xzvc9:8080 to-stack FORWARDED (TCP Flags: ACK,
PSH)
Nov 23 14:49:04.227: 10.0.0.113:32898 -> kube-system/coredns-558bd4d5db-xzvc9:8080 to-endpoint FORWARDED (TCP Flags: AC
K, FIN)
Nov 23 14:49:04.227: 10.0.0.113:32898 <- kube-system/coredns-558bd4d5db-xzvc9:8080 to-stack FORWARDED (TCP Flags: ACK,
FIN)
Nov 23 14:49:04.227: 10.0.0.113:32898 -> kube-system/coredns-558bd4d5db-xzvc9:8080 to-endpoint FORWARDED (TCP Flags: AC
K)
Nov 23 14:49:04.842: 10.0.0.113:34716 -> kube-system/coredns-558bd4d5db-xzvc9:8181 to-endpoint FORWARDED (TCP Flags: SY
N)
Nov 23 14:49:04.842: 10.0.0.113:34716 <- kube-system/coredns-558bd4d5db-xzvc9:8181 to-stack FORWARDED (TCP Flags: SYN,
ACK)
Nov 23 14:49:04.842: 10.0.0.113:34716 -> kube-system/coredns-558bd4d5db-xzvc9:8181 to-endpoint FORWARDED (TCP Flags: AC
K)
Nov 23 14:49:04.842: 10.0.0.113:34716 -> kube-system/coredns-558bd4d5db-xzvc9:8181 to-endpoint FORWARDED (TCP Flags: AC
K, PSH)
Nov 23 14:49:04.842: 10.0.0.113:34716 <- kube-system/coredns-558bd4d5db-xzvc9:8181 to-stack FORWARDED (TCP Flags: ACK,
PSH)
Nov 23 14:49:04.843: 10.0.0.113:34716 <- kube-system/coredns-558bd4d5db-xzvc9:8181 to-stack FORWARDED (TCP Flags: ACK,
FIN)
Nov 23 14:49:04.843: 10.0.0.113:34716 -> kube-system/coredns-558bd4d5db-xzvc9:8181 to-endpoint FORWARDED (TCP Flags: AC
K, FIN)
Nov 23 14:49:05.971: kube-system/hubble-relay-f6d85866c-csthd:40844 -> 192.168.49.2:4244 to-stack FORWARDED (TCP Flags:
 ACK, PSH)
```

with

```
hubble observe -f
```

you can observe and follow the currently active flows in your Kubernetes cluster. Stop the command with `CTRL+C` .

Let us produce some traffic:

```
for i in {1..10}; do
  kubectl exec -ti ${FRONTEND} -- curl -I --connect-timeout 5 backend:8080
  kubectl exec -ti ${NOT_FRONTEND} -- curl -I --connect-timeout 5 backend:8080
done
```

We can now use the `hubble` CLI to filter traffic we are interested in. Here are some examples to specifically retrieve the network activity between our frontends and backend:

```
hubble observe --to-pod backend
hubble observe --namespace default --protocol tcp --port 8080
```

Note that Hubble tells us the action, here `FORWARDED`, but it could also be `DROPPED`. If you only want to see `DROPPED` traffic. You can execute

```
hubble observe --verdict DROPPED
```

For now this should only show some packets that have been sent to an already deleted pod. After we configured NetworkPolicies we will see other dropped packets.

3.2. Hubble UI

Not only does Hubble allow us to inspect flows from the command line, but it also allows us to see them in real-time on a graphical service map via Hubble UI. Again, this also is an optional component that is disabled by default.

Task 3.2.1: Enable the Hubble UI component

Enabling the optional Hubble UI component with Helm looks like this:

```
helm upgrade -i cilium cilium/cilium --version 1.12.10 \
  --namespace kube-system \
  --set ipam.operator.clusterPoolIPv4PodCIDRList={10.1.0.0/16} \
  --set cluster.name=cluster1 \
  --set cluster.id=1 \
  --set operator.replicas=1 \
  --set upgradeCompatibility=1.11 \
  --set kubeProxyReplacement=disabled \
  --set hubble.enabled=true \
  --set hubble.relay.enabled=true \
  `# enable hubble ui variable:` \
  --set hubble.ui.enabled=true \
  --wait
```

> **Note**
> When using the `cilium` CLI, you can execute the following command to enable the Hubble UI:
>
> ```
> # cilium hubble enable --ui
> ```

Take a look at the pods again to see what happened under the hood:

```
kubectl get pods -A
```

We see, there is again a new Pod running for the `hubble-ui` component.

```
NAMESPACE     NAME                                READY   STATUS    RESTARTS       AGE
default       backend-6f884b6495-v7bvt            1/1     Running   0              94m
default       frontend-77d99ffc5d-lcsph           1/1     Running   0              94m
default       not-frontend-7db9747986-snjwp       1/1     Running   0              94m
kube-system   cilium-ksr7h                        1/1     Running   0              102m
kube-system   cilium-operator-6f5c6f768d-r2qgn    1/1     Running   0              102m
kube-system   coredns-6d4b75cb6d-nf8wz            1/1     Running   0              115m
kube-system   etcd-cluster1                       1/1     Running   0              115m
kube-system   hubble-relay-84b4ddb556-nr7c8       1/1     Running   0              93m
kube-system   hubble-ui-579fdfbc58-578g9          2/2     Running   0              19s
kube-system   kube-apiserver-cluster1             1/1     Running   0              115m
kube-system   kube-controller-manager-cluster1    1/1     Running   0              115m
kube-system   kube-proxy-7l6qk                    1/1     Running   0              115m
kube-system   kube-scheduler-cluster1             1/1     Running   0              115m
kube-system   storage-provisioner                 1/1     Running   1 (115m ago)   115m
```

Cilium agents are restarting, and a new Hubble UI Pod is now present on top of the Hubble Relay pod. As above, we can wait for Cilium and Hubble to be ready by running:

```
cilium status --wait
```

```
cilium status --wait
    /¯¯\
 /¯¯\__/¯¯\    Cilium:         OK
 \__/¯¯\__/    Operator:       OK
 /¯¯\__/¯¯\    Hubble:         OK
 \__/¯¯\__/    ClusterMesh:    disabled
    \__/

Deployment        hubble-relay       Desired: 1, Ready: 1/1, Available: 1/1
Deployment        cilium-operator    Desired: 1, Ready: 1/1, Available: 1/1
Deployment        hubble-ui          Desired: 1, Ready: 1/1, Available: 1/1
DaemonSet         cilium             Desired: 1, Ready: 1/1, Available: 1/1
Containers:       cilium             Running: 1
                  hubble-ui          Running: 1
                  hubble-relay       Running: 1
                  cilium-operator    Running: 1
Cluster Pods:     6/6 managed by Cilium
Image versions    cilium             quay.io/cilium/cilium:v1.12.1:: 1
                  hubble-ui          quay.io/cilium/hubble-ui:v0.9.1: 1
                  hubble-ui          quay.io/cilium/hubble-ui-backend:v0.9.1: 1
                  hubble-relay       quay.io/cilium/hubble-relay:v1.12.1: 1
                  cilium-operator    quay.io/cilium/operator-generic:v1.12.1: 1
```

And then check Hubble status:

```
hubble status
```

Note

Our earlier command kubectl port-forward should still be running (can be checked by running jobs or `ps aux | grep "port-forward"`). If it does not, Hubble status will fail and we have to run it again:

```
cilium hubble port-forward&
hubble status
```

To start Hubble UI execute

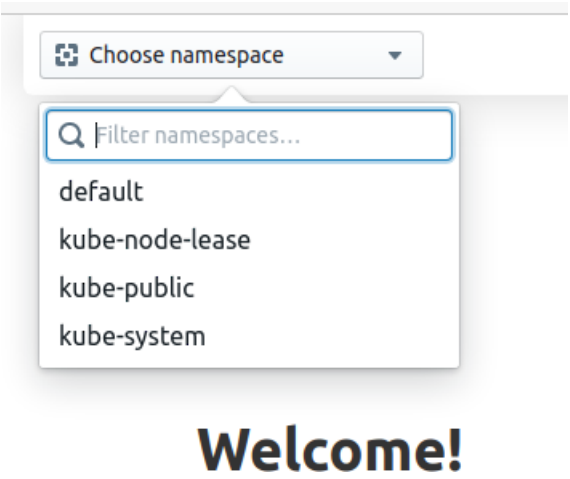
```
kubectl port-forward -n kube-system --address ::,0.0.0.0 svc/hubble-ui 12000:80 &
```

In our Webshell environment you can use the public IP of the VM to access Hubble. A simple way is to execute

```
echo "http://$(curl -s ifconfig.me):12000"
```

and copy the output in a new browser tab. If you are working locally, the browser should open http://localhost:12000/ (open it manually if not).

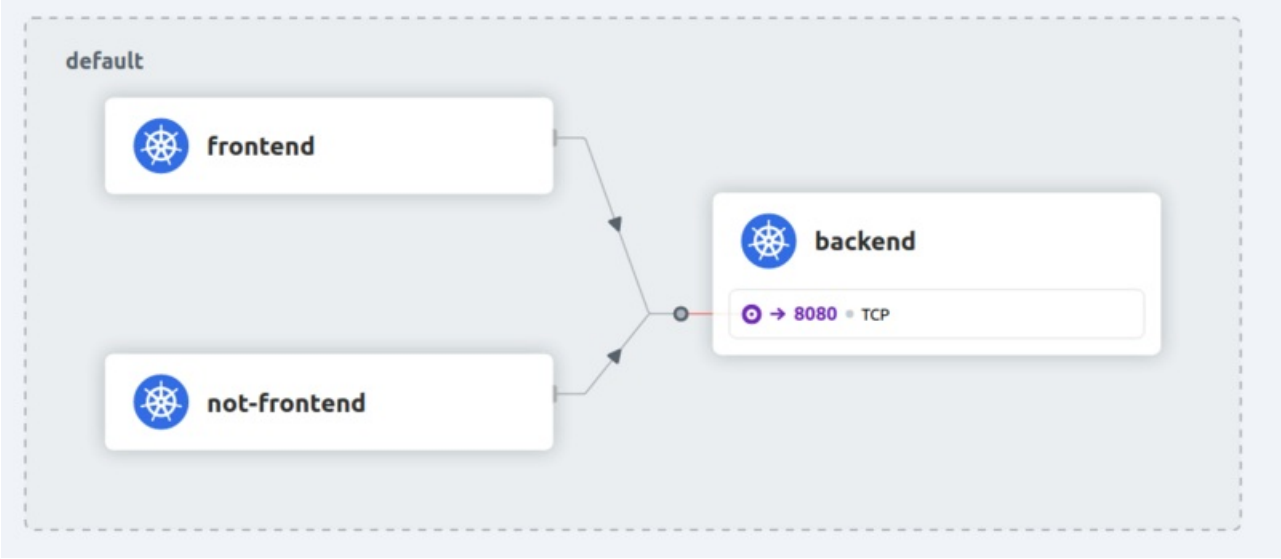
We can then access the graphical service map by selecting our `default` Namespace:



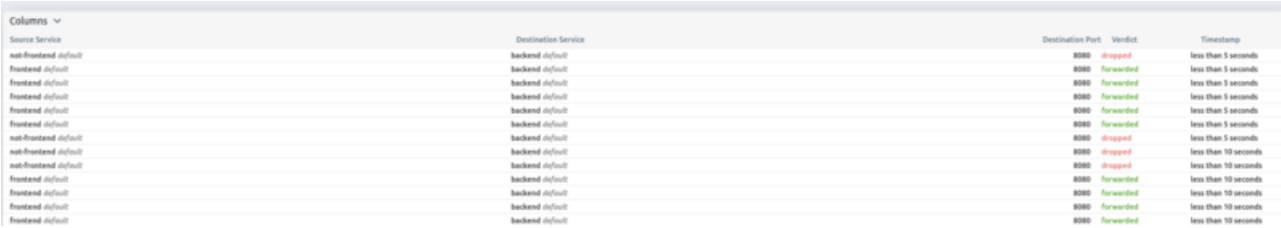
If you see a spinning circle and the message “Waiting for service map data…” you can generate some network activity again:

```
for i in {1..10}; do
  kubectl exec -ti ${FRONTEND} -- curl -I --connect-timeout 5 backend:8080
  kubectl exec -ti ${NOT_FRONTEND} -- curl -I --connect-timeout 5 backend:8080
done
```

and then you should see a service map in the Hubble UI



and also a table with the already familiar flow output previously seen in the `hubble observe` command:



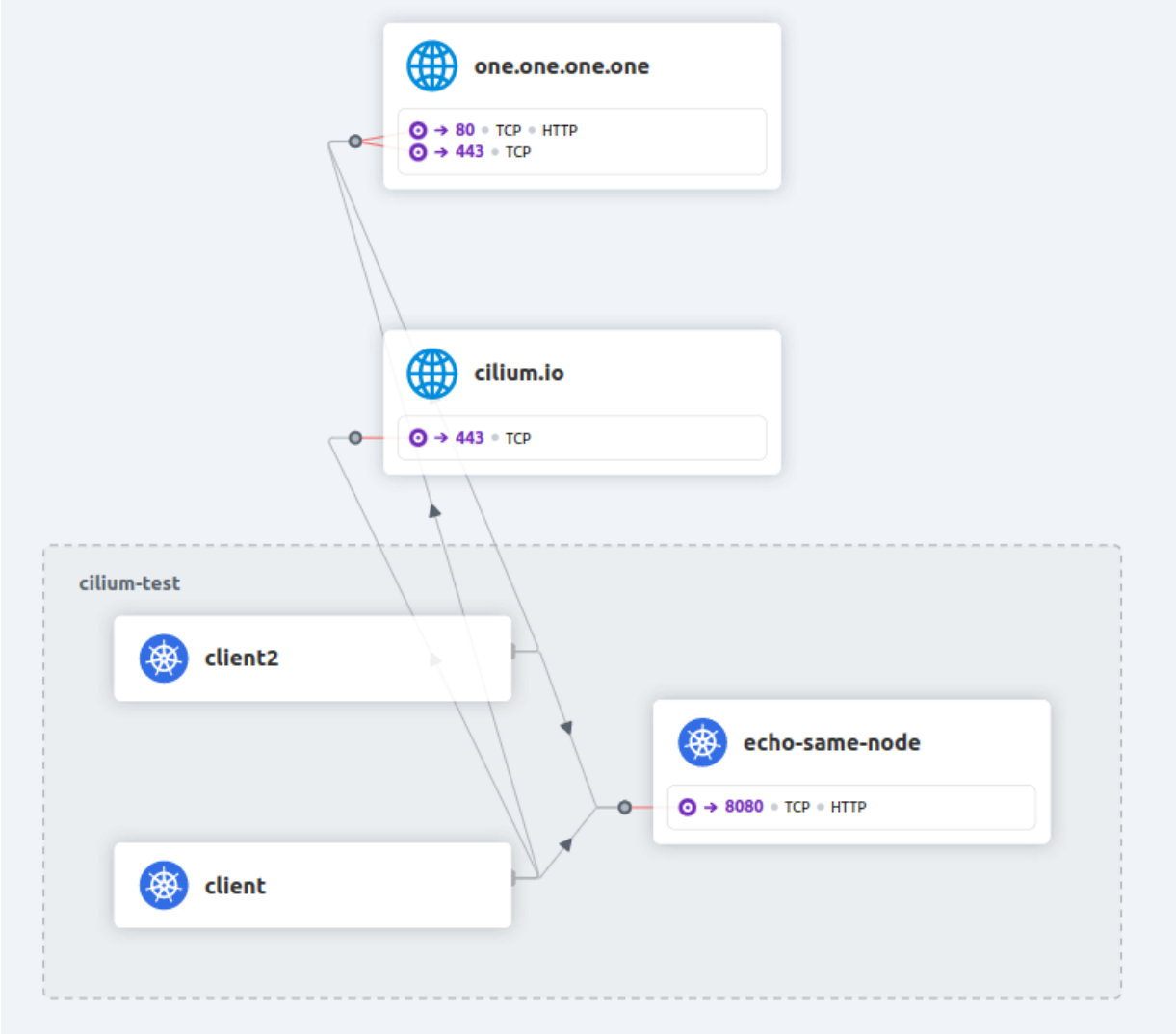
Hubble flows are displayed in real-time at the bottom, with a visualization of the namespace objects in the center. Click on any flow, and click on any property from the right-side panel: notice that the filters at the

top of the UI have been updated accordingly.

Let’s run a connectivity test again and see what happens in Hubble UI in the `cilium-test` namespace. In the Hubble UI dropdown change to `cilium-test`. Since this test runs for a few minutes this could be a good time to grab a ☕.

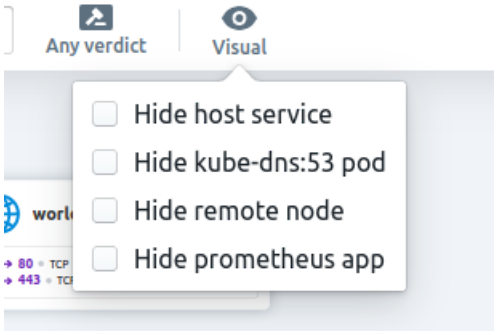
```
cilium connectivity test --test 'client-egress-to-echo-service-account' --test to-entities-world --test to-fqdns
```

We can see that Hubble UI is not only capable of displaying flows within a Namespace, it also helps visualize flows going in or out of it.



And there are also several visual options in the Hubble UI:

- acend gmbh



Once done, clean up the connectivity test Namespace again:

```
kubectl delete ns cilium-test --wait=false
```

- acend gmbh

4. Metrics

With metrics displayed in Grafana or another UI, we can get a quick overview of our cluster state and its traffic.

Both Cilium and Hubble can be configured to serve Prometheus metrics independently of each other. Cilium metrics show us the state of Cilium itself, namely of the `cilium-agent`, `cilium-envoy`, and `cilium-operator` processes. Hubble metrics on the other hand give us information about the traffic of our applications.

Task 4.1: Enable metrics

We start by enabling different metrics, for dropped and HTTP traffic we also want to have metrics specified by pod.

```
helm upgrade -i cilium cilium/cilium --version 1.12.10 \
  --namespace kube-system \
  --set ipam.operator.clusterPoolIPv4PodCIDRList={10.1.0.0/16} \
  --set cluster.name=cluster1 \
  --set cluster.id=1 \
  --set operator.replicas=1 \
  --set upgradeCompatibility=1.11 \
  --set kubeProxyReplacement=disabled \
  --set hubble.enabled=true \
  --set hubble.relay.enabled=true \
  --set hubble.ui.enabled=true \
  `# enable metrics:` \
  --set prometheus.enabled=true \
  --set operator.prometheus.enabled=true \
  --set hubble.metrics.enabled="{dns,drop:destinationContext=pod;sourceContext=pod,tcp,flow,port-distribution,icmp,http
:destinationContext=pod}"
```

Verify Cilium metrics

We now verify that the Cilium agent has different metric endpoints exposed and list some of them:

- hubble port 9965
- cilium agent port 9962
- cilium envoy port 9095

```
CILIUM_AGENT_IP=$(kubectl get pod -n kube-system -l k8s-app=cilium -o jsonpath="{.items[0].status.hostIP}")
kubectl run -n kube-system -it --env="CILIUM_AGENT_IP=${CILIUM_AGENT_IP}" --rm curl --image=curlimages/curl -- sh
```

```
echo ${CILIUM_AGENT_IP}
curl -s ${CILIUM_AGENT_IP}:9962/metrics | grep cilium_nodes_all_num #show total number of cilium nodes
curl -s ${CILIUM_AGENT_IP}:9965/metrics | grep hubble_tcp_flags_total # show total number of TCP flags
exit
```

You should see now an output like this.

```
If you don't see a command prompt, try pressing enter.
echo ${CILIUM_AGENT_IP}
192.168.49.2
/ $ curl -s ${CILIUM_AGENT_IP}:9962/metrics | grep cilium_nodes_all_num #show total number of cilium nodes
# HELP cilium_nodes_all_num Number of nodes managed
# TYPE cilium_nodes_all_num gauge
cilium_nodes_all_num 1
/ $ curl -s ${CILIUM_AGENT_IP}:9965/metrics | grep hubble_tcp_flags_total # show total number of TCP flags
# HELP hubble_tcp_flags_total TCP flag occurrences
# TYPE hubble_tcp_flags_total counter
hubble_tcp_flags_total{family="IPv4",flag="FIN"} 2704
hubble_tcp_flags_total{family="IPv4",flag="RST"} 388
hubble_tcp_flags_total{family="IPv4",flag="SYN"} 1609
hubble_tcp_flags_total{family="IPv4",flag="SYN-ACK"} 1549
```

Note

The Cilium agent pods run as DaemonSet on the HostNetwork. This means you could also directly call a node.

```
NODE=$(kubectl get nodes --selector=kubernetes.io/role!=master -o jsonpath={.items[*].status.addresses[?\(@.type==\"In
ternalIP\"\)].address})
curl -s $NODE:9962/metrics | grep cilium_nodes_all_num
```

Note

It is not yet possible to get metrics from Cilium Envoy (port 9095). Envoy only starts on a node if there is at least one Pod with a layer 7 networkpolicy.

Task 4.2: Store and visualize metrics

To make sense of metrics, we store them in Prometheus and visualize them with Grafana dashboards. Install both into `cilium-monitoring` Namespace to store and visualize Cilium and Hubble metrics.

```
kubectl apply -f https://raw.githubusercontent.com/cilium/cilium/v1.12/examples/kubernetes/addons/prometheus/monitoring
-example.yaml
```

Make sure Prometheus and Grafana pods are up and running before continuing with the next step.

```
kubectl -n cilium-monitoring get pod
```

you should see both Pods in state `Running` :

```
NAME                          READY   STATUS    RESTARTS   AGE
grafana-6c7d4c9fd8-2xdp2      1/1     Running   0          41s
prometheus-55777f54d9-hkpkq   1/1     Running   0          41s
```

Generate some traffic for some minutes in the background

```
FRONTEND=$(kubectl get pods -l app=frontend -o jsonpath='{.items[0].metadata.name}')
i=0; while [ $i -le 300 ]; do kubectl exec -ti ${FRONTEND} -- curl -Is backend:8080; sleep 1; ((i++)); done &
```

In a second terminal access Grafana with kubectl proxy-forward (for those in the webshell: don't forget to connect to the VM first)

```
kubectl -n cilium-monitoring port-forward service/grafana --address ::,0.0.0.0 3000:3000 &
echo "http://$(curl -s ifconfig.me):3000/dashboards"
```

Now open a new tab in your browser and go to URL from the output (for those working on their localmachine use http://localhost:3000/dashboards). In Grafana use the left side menu: `Dashboard` , click on `Manage` , then click on `Hubble` . For a better view, you can change the timespan to the last 5 minutes.

Verify that you see the generated traffic under Network, Forwarded vs Dropped Traffic. Not all graphs will have data available. This is because we have not yet used network policies or any layer 7 components. This will be done in the later chapters.

Change to the `Cilium Metrics` Dashboard. Here we see information about Cilium itself. Again not all graphs contain data as we have not used all features of Cilium yet.

Try to find the number of IPs allocated and the number of Cilium endpoints.

Leave the Grafana Tab open, we will use it in the later chapters.

5. Troubleshooting

For more details on Troubleshooting, have a look into Cilium's Troubleshooting Documentation .

Component & Cluster Health

An initial overview of Cilium can be retrieved by listing all pods to verify whether all pods have the status `Running`:

```
kubectl -n kube-system get pods -l k8s-app=cilium
```

In our single node cluster there is only one cilium pod running:

```
NAME           READY     STATUS    RESTARTS   AGE
cilium-ksr7h   1/1       Running   0          12m16
```

If Cilium encounters a problem that it cannot recover from, it will automatically report the failure state via `cilium status` which is regularly queried by the Kubernetes liveness probe to automatically restart Cilium pods. If a Cilium Pod is in state `CrashLoopBackoff` then this indicates a permanent failure scenario.

If a particular Cilium Pod is not in a running state, the status and health of the agent on that node can be retrieved by running `cilium status` in the context of that pod:

```
kubectl -n kube-system exec ds/cilium -- cilium status
```

The output looks similar to this:

```
Defaulted container "cilium-agent" out of: cilium-agent, mount-cgroup (init), apply-sysctl-overwrites (init), mount-bpf
-fs (init), clean-cilium-state (init)
KVStore:                 Ok   Disabled
Kubernetes:              Ok   1.24 (v1.24.3) [linux/amd64]
Kubernetes APIs:         ["cilium/v2::CiliumClusterwideNetworkPolicy", "cilium/v2::CiliumEndpoint", "cilium/v2::CiliumN
etworkPolicy", "cilium/v2::CiliumNode", "core/v1::Namespace", "core/v1::Node", "core/v1::Pods", "core/v1::Service", "di
scovery/v1::EndpointSlice", "networking.k8s.io/v1::NetworkPolicy"]
KubeProxyReplacement:    Disabled
Host firewall:           Disabled
CNI Chaining:            none
Cilium:                  Ok   1.12.5 (v1.12.5-701acde)
NodeMonitor:             Listening for events on 8 CPUs with 64x4096 of shared memory
Cilium health daemon:    Ok
IPAM:                    IPv4: 10/254 allocated from 10.1.0.0/24,
ClusterMesh:             0/0 clusters ready, 0 global-services
BandwidthManager:        Disabled
Host Routing:            Legacy
Masquerading:            IPTables [IPv4: Enabled, IPv6: Disabled]
Controller Status:       50/50 healthy
Proxy Status:            OK, ip 10.1.0.182, 0 redirects active on ports 10000-20000
Global Identity Range:   min 256, max 65535
Hubble:                  Ok   Current/Max Flows: 4095/4095 (100.00%), Flows/s: 8.71   Metrics: Ok
Encryption:              Disabled
Cluster health:          1/1 reachable   (2023-01-26T08:23:50Z)
```

More detailed information about the status of Cilium can be inspected with:

```
kubectl -n kube-system exec ds/cilium -- cilium status --verbose
```

Verbose output includes detailed IPAM state (allocated addresses), Cilium controller status, and details of the Proxy status.

Logs

To retrieve log files of a cilium pod, run:

```
kubectl -n kube-system logs --timestamps <pod-name>
```

The `<pod-name>` can be determined with the following command and by selecting the name of one of the pods:

```
kubectl -n kube-system get pods -l k8s-app=cilium
```

If the Cilium Pod was already restarted due to the liveness problem after encountering an issue, it can be useful to retrieve the logs of the Pod previous to the last restart:

```
kubectl -n kube-system logs --timestamps -p <pod-name>
```

Policy Troubleshooting - Ensure Pod is managed by Cilium

A potential cause for policy enforcement not functioning as expected is that the networking of the Pod selected by the policy is not being managed by Cilium. The following situations result in unmanaged pods:

- The Pod is running in host networking and will use the host's IP address directly. Such pods have full network connectivity but Cilium will not provide security policy enforcement for such pods.
- The Pod was started before Cilium was deployed. Cilium only manages pods that have been deployed after Cilium itself was started. Cilium will not provide security policy enforcement for such pods.

If Pod networking is not managed by Cilium, ingress and egress policy rules selecting the respective pods will not be applied. See the section Network Policy for more details.

For a quick assessment of whether any pods are not managed by Cilium, the Cilium CLI will print the number of managed pods. If this prints that all of the pods are managed by Cilium, then there is no problem:

```
cilium status
```

- acend gmbh

```
    /¯¯\
 /¯¯\__/¯¯\    Cilium:         OK
 \__/¯¯\__/    Operator:       OK
 /¯¯\__/¯¯\    Hubble:         OK
 \__/¯¯\__/    ClusterMesh:    disabled
    \__/

Deployment        cilium-operator    Desired: 2, Ready: 2/2, Available: 2/2
Deployment        hubble-relay       Desired: 1, Ready: 1/1, Available: 1/1
Deployment        hubble-ui          Desired: 1, Ready: 1/1, Available: 1/1
DaemonSet         cilium             Desired: 2, Ready: 2/2, Available: 2/2
Containers:       cilium-operator    Running: 2
                  hubble-relay       Running: 1
                  hubble-ui          Running: 1
                  cilium             Running: 2
Cluster Pods:     5/5 managed by Cilium
```

You can run the following script to list the pods which are not managed by Cilium:

```
curl -sLO https://raw.githubusercontent.com/cilium/cilium/master/contrib/k8s/k8s-unmanaged.sh
chmod +x k8s-unmanaged.sh
./k8s-unmanaged.sh
```

> **Note**
> It’s ok if you don’t see any Pods listed with the above command. We don’t have any unmanaged Pods in our setup.

Reporting a problem - Automatic log & state collection

Before you report a problem, make sure to retrieve the necessary information from your cluster before the failure state is lost.

Execute the `cilium sysdump` command to collect troubleshooting information from your Kubernetes cluster:

```
cilium sysdump
```

Note that by default `cilium sysdump` will attempt to collect as many logs as possible for all the nodes in the cluster. If your cluster size is above 20 nodes, consider setting the following options to limit the size of the sysdump. This is not required, but is useful for those who have a constraint on bandwidth or upload size.

- set the `--node-list` option to pick only a few nodes in case the cluster has many of them.
- set the `--logs-since-time` option to go back in time to when the issues started.
- set the `--logs-limit-bytes` option to limit the size of the log files (note: passed onto kubectl logs; does not apply to entire collection archive). Ideally, a sysdump that has a full history of select nodes, rather than a brief history of all the nodes, would be preferred (by using `--node-list`). The second recommended way would be to use `--logs-since-time` if you are able to narrow down when the issues started. Lastly, if the Cilium agent and Operator logs are too large, consider `--logs-limit-bytes` .

Use `--help` to see more options:

```
cilium sysdump --help
```

6. Network Policies

Network Policies

One CNI function is the ability to enforce network policies and implement an in-cluster zero-trust container strategy. Network policies are a default Kubernetes object for controlling network traffic, but a CNI such as Cilium is required to enforce them. We will demonstrate traffic blocking with our simple app.

> **Note**
> If you are not yet familiar with Kubernetes Network Policies we suggest going to the Kubernetes Documentation

Task 6.1: Cilium Endpoints and Identities

Each Pod from our simple application is represented in Cilium as an Endpoint . We can use the `cilium` tool inside a Cilium Pod to list them.

First get all Cilium pods with:

```
kubectl -n kube-system get pods -l k8s-app=cilium
```

```
NAME           READY     STATUS    RESTARTS   AGE
cilium-ksr7h   1/1       Running   0          13m16
```

and then run:

```
kubectl -n kube-system exec <podname> -- cilium endpoint list
```

> **Note**
> Or we just execute the first Pod of the DaemonSet:
>
> ```
> kubectl -n kube-system exec ds/cilium -- cilium endpoint list
> ```

Cilium will match these endpoints with labels and generate identities as a result. The identity is what is used to enforce basic connectivity between endpoints. We can see this change of identity:

```
kubectl run test-identity --image=nginx
sleep 5 # just wait for the pod to get ready
kubectl -n kube-system exec daemonset/cilium -- cilium endpoint list | grep -E -B4 -A1  'IDENTITY|run'
kubectl label pod test-identity this=that
sleep 5 # give some time to process
kubectl -n kube-system exec daemonset/cilium -- cilium endpoint list | grep -E -B4 -A1  'IDENTITY|run'
kubectl delete pod test-identity
```

We see that the number for this Pod in the column IDENTITY has changed after we added another label. If you run `endpoint list` right after pod-labeling you might also see `waiting-for-identity` as the status of the endpoint.

Task 6.2: Verify connectivity

Make sure your `FRONTEND` and `NOT_FRONTEND` environment variable are still set. Otherwise set them again:

```
FRONTEND=$(kubectl get pods -l app=frontend -o jsonpath='{.items[0].metadata.name}')
echo ${FRONTEND}
NOT_FRONTEND=$(kubectl get pods -l app=not-frontend -o jsonpath='{.items[0].metadata.name}')
echo ${NOT_FRONTEND}
```

Now we generate some traffic as a baseline test.

```
kubectl exec -ti ${FRONTEND} -- curl -I --connect-timeout 5 backend:8080
```

and

```
kubectl exec -ti ${NOT_FRONTEND} -- curl -I --connect-timeout 5 backend:8080
```

This will execute a simple `curl` call from the `frontend` and `not-frondend` application to the `backend` application:

```
# Frontend
HTTP/1.1 200 OK
X-Powered-By: Express
Vary: Origin, Accept-Encoding
Access-Control-Allow-Credentials: true
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Last-Modified: Sat, 26 Oct 1985 08:15:00 GMT
ETag: W/"83d-7438674ba0"
Content-Type: text/html; charset=UTF-8
Content-Length: 2109
Date: Tue, 23 Nov 2021 12:50:44 GMT
Connection: keep-alive

# Not Frontend
HTTP/1.1 200 OK
X-Powered-By: Express
Vary: Origin, Accept-Encoding
Access-Control-Allow-Credentials: true
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Last-Modified: Sat, 26 Oct 1985 08:15:00 GMT
ETag: W/"83d-7438674ba0"
Content-Type: text/html; charset=UTF-8
Content-Length: 2109
Date: Tue, 23 Nov 2021 12:50:44 GMT
Connection: keep-alive
```

and we see, both applications can connect to the `backend` application.

Until now ingress and egress policy enforcement are still disabled on all of our pods because no network

policy has been imported yet selecting any of the pods. Let us change this.

Task 6.3: Deny traffic with a Network Policy

We block traffic by applying a network policy. Create a file `backend-ingress-deny.yaml` with the following content:

```
---
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: backend-ingress-deny
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
  - Ingress
```

The policy will deny all ingress traffic as it is of type Ingress but specifies no allow rule, and will be applied to all pods with the `app=backend` label thanks to the podSelector.

Ok, then let's create the policy with:

```
kubectl apply -f backend-ingress-deny.yaml
```

and you can verify the created `NetworkPolicy` with:

```
kubectl get netpol
```

which gives you an output similar to this:

```
NAME                   POD-SELECTOR   AGE
backend-ingress-deny   app=backend    2s
```

Task 6.4: Verify connectivity again

We can now execute the connectivity check again:

```
kubectl exec -ti ${FRONTEND} -- curl -I --connect-timeout 5 backend:8080
```

and

```
kubectl exec -ti ${NOT_FRONTEND} -- curl -I --connect-timeout 5 backend:8080
```

but this time you see that the `frontend` and `not-frontend` application cannot connect anymore to the `backend`:

```
# Frontend
curl: (28) Connection timed out after 5001 milliseconds
command terminated with exit code 28
# Not Frontend
curl: (28) Connection timed out after 5001 milliseconds
command terminated with exit code 28
```

The network policy correctly switched the default ingress behavior from default allow to default deny. We can also check this in Grafana.

> **Note**
>
> Note: our earlier Grafana port-forward should still be running (can be checked by running jobs or `ps aux | grep "grafana"`). If it does not open the URL from the command output below (or http://localhost:3000/dashboards with a local setup).

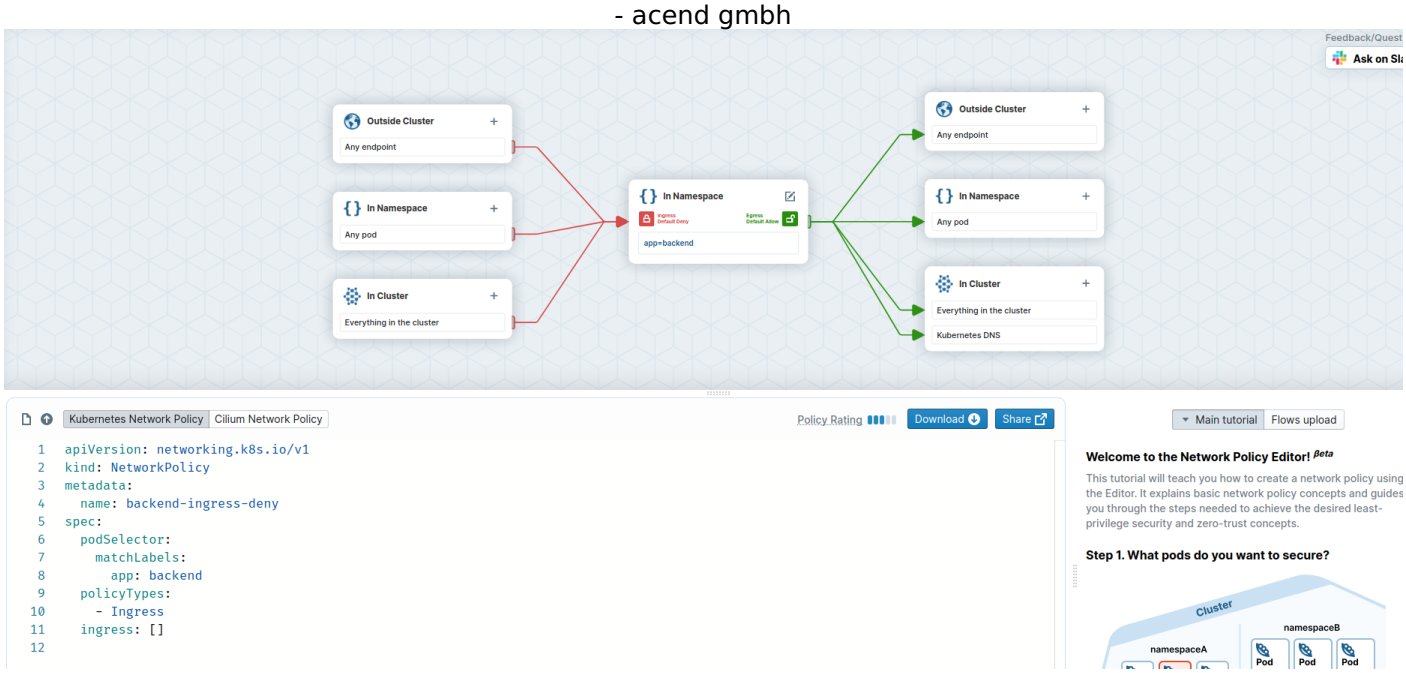
```
kubectl -n cilium-monitoring port-forward service/grafana --address 0.0.0.0 --address :: 3000:3000 &
echo "http://$(curl -s ifconfig.me):3000/dashboards"
```

In Grafana browse to the dashboard `Hubble`. You should see now data in more graphs. Check the graphs `Drop Reason`, `Forwarded vs Dropped`. In `Top 10 Source Pods with Denied Packets` you should find the name of the pods from our simple application.

Let's now selectively re-allow traffic again, but only from frontend to backend.

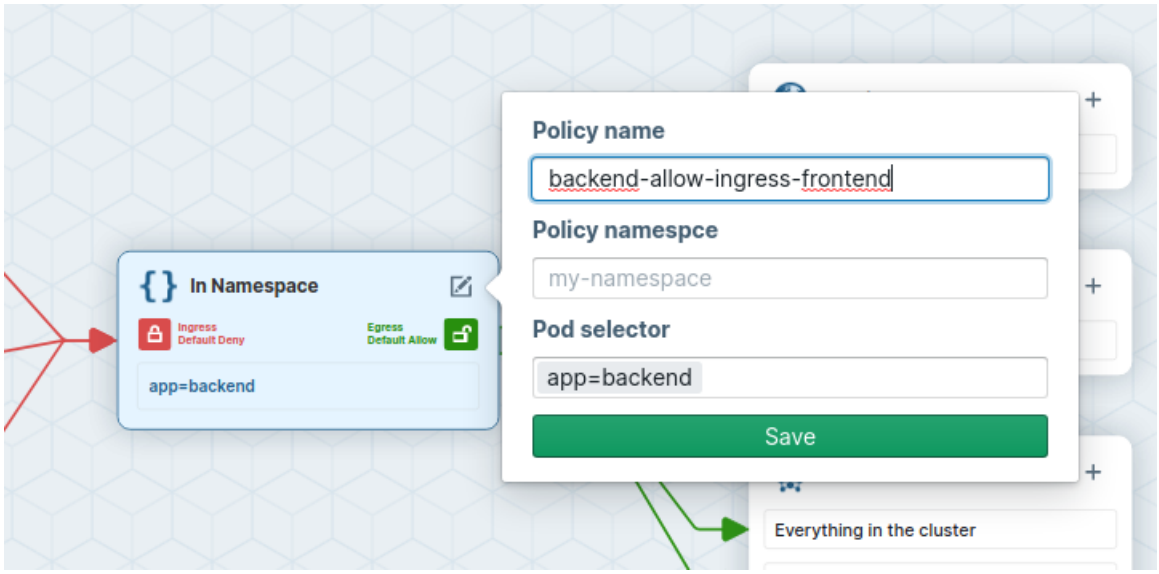
Task 6.5: Allow traffic from frontend to backend

We can do it by crafting a new network policy manually, but we can also use the Network Policy Editor to help us out:

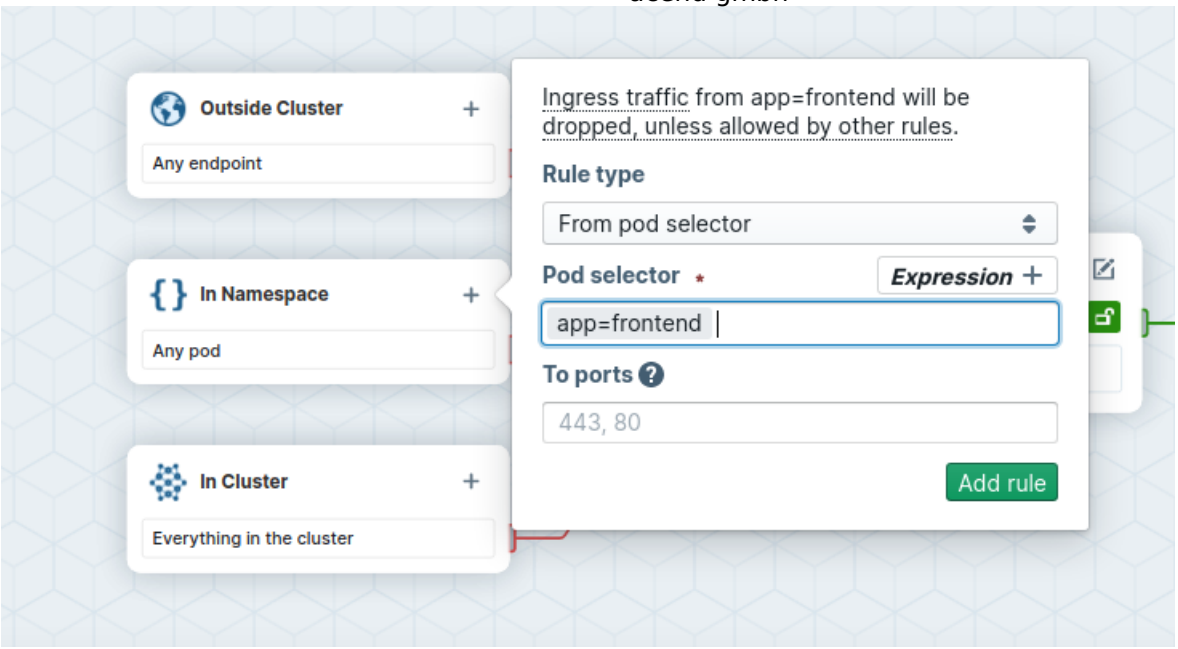


Above you see our original policy, we create an new one with the editor now.

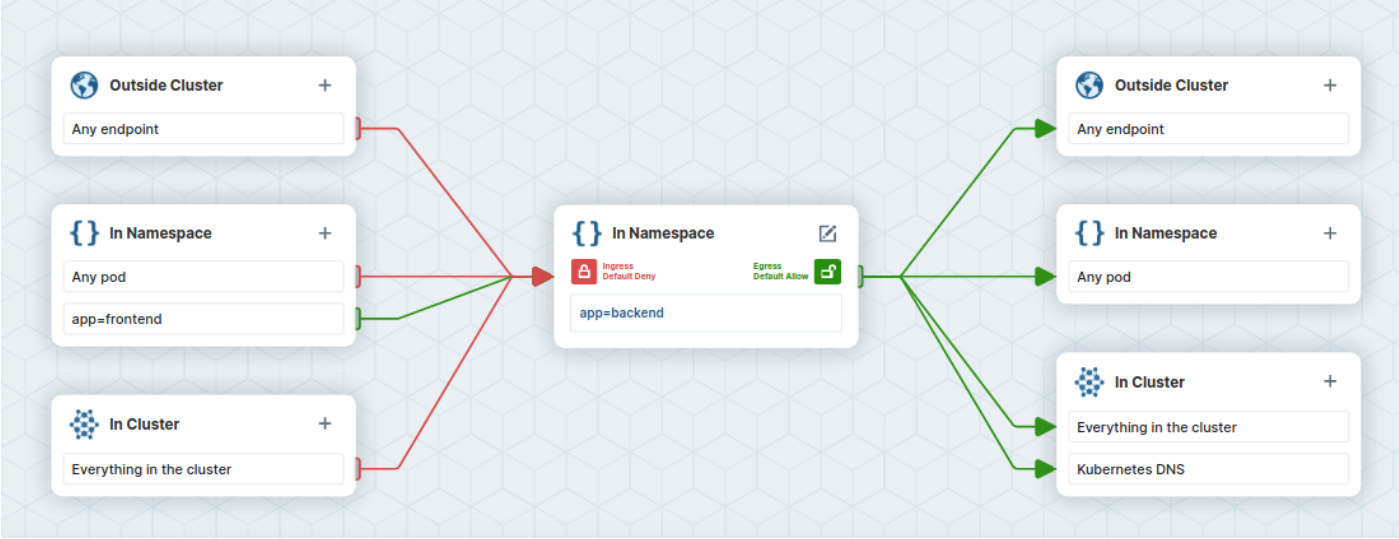
- Go to https://networkpolicy.io/editor .
- Name the network policy to backend-allow-ingress-frontend (using the Edit button in the center).
- add app=backend as Pod Selector
- Set Ingress to default deny



- On the ingress side, add `app=frontend` as podSelector for pods in the same Namespace.



- Inspect the ingress flow colors: the policy will deny all ingress traffic to pods labeled `app=backend`, except for traffic coming from pods labeled `app=frontend`.



- Copy the policy YAML into a file named `backend-allow-ingress-frontend.yaml`. Make sure to use the `Networkpolicy` and not the `CiliumNetworkPolicy`.

The file should look like this:

```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: "backend-allow-ingress-frontend"
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
    - Ingress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: frontend
```

Apply the new policy:

```
kubectl apply -f backend-allow-ingress-frontend.yaml
```

and then execute the connectivity test again:

```
kubectl exec -ti ${FRONTEND} -- curl -I --connect-timeout 5 backend:8080
```

and

```
kubectl exec -ti ${NOT_FRONTEND} -- curl -I --connect-timeout 5 backend:8080
```

This time, the `frontend` application is able to connect to the `backend` but the `not-frontend` application still cannot connect to the `backend` :

```
# Frontend
HTTP/1.1 200 OK
X-Powered-By: Express
Vary: Origin, Accept-Encoding
Access-Control-Allow-Credentials: true
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Last-Modified: Sat, 26 Oct 1985 08:15:00 GMT
ETag: W/"83d-7438674ba0"
Content-Type: text/html; charset=UTF-8
Content-Length: 2109
Date: Tue, 23 Nov 2021 13:08:27 GMT
Connection: keep-alive

# Not Frontend
curl: (28) Connection timed out after 5001 milliseconds
command terminated with exit code 28
```

Note that this is working despite the fact we did not delete the previous `backend-ingress-deny` policy:

```
kubectl get netpol
```

```
NAME                              POD-SELECTOR   AGE
backend-allow-ingress-frontend    app=backend    2m7s
backend-ingress-deny              app=backend    12m
```

Network policies are additive. Just like with firewalls, it is thus a good idea to have default DENY policies and then add more specific ALLOW policies as needed.

We can verify our connection being blocked with Hubble.

Generate some traffic.

```
kubectl exec -ti ${NOT_FRONTEND} -- curl -I --connect-timeout 5 backend:8080
```

With `hubble observe` you can now check the packet being dropped as well as the reason why (Policy denied).

Note

Our earlier port-forward should still be running (can be checked by running jobs or `ps aux | grep "port-forward svc/hubble-relay"`). If it does not, Hubble status will fail and we have to run it again:

```
kubectl -n kube-system port-forward svc/hubble-relay 4245:80 &
hubble status
```

```
hubble observe --from-label app=not-frontend --to-label app=backend
```

And the output should look like this:

```
Jan 26 09:07:03.396: default/not-frontend-7db9747986-gktg6:45002 (ID:84671) <> default/backend-6f884b6495-69bbh:8080 (I
D:68421) policy-verdict:none INGRESS DENIED (TCP Flags: SYN)
Jan 26 09:07:03.396: default/not-frontend-7db9747986-gktg6:45002 (ID:84671) <> default/backend-6f884b6495-69bbh:8080 (I
D:68421) Policy denied DROPPED (TCP Flags: SYN)
Jan 26 09:07:04.401: default/not-frontend-7db9747986-gktg6:45002 (ID:84671) <> default/backend-6f884b6495-69bbh:8080 (I
D:68421) policy-verdict:none INGRESS DENIED (TCP Flags: SYN)
Jan 26 09:07:04.401: default/not-frontend-7db9747986-gktg6:45002 (ID:84671) <> default/backend-6f884b6495-69bbh:8080 (I
D:68421) Policy denied DROPPED (TCP Flags: SYN)
Jan 26 09:07:06.418: default/not-frontend-7db9747986-gktg6:45002 (ID:84671) <> default/backend-6f884b6495-69bbh:8080 (I
D:68421) policy-verdict:none INGRESS DENIED (TCP Flags: SYN)
Jan 26 09:07:06.418: default/not-frontend-7db9747986-gktg6:45002 (ID:84671) <> default/backend-6f884b6495-69bbh:8080 (I
D:68421) Policy denied DROPPED (TCP Flags: SYN)
```

Task 6.6: Inspecting the Cilium endpoints again

We can now check the Cilium endpoints again.

```
kubectl -n kube-system exec -it ds/cilium -- cilium endpoint list
```

And now we see that the pods with the label `app=backend` now have ingress policy enforcement enabled.

```
ENDPOINT   POLICY (ingress)   POLICY (egress)   IDENTITY   LABELS (source:key[=value])
                      IPv6   IPv4         STATUS
           ENFORCEMENT        ENFORCEMENT
248        Enabled            Disabled          68421      k8s:app=backend
                              10.1.0.1     ready
                                                          k8s:io.cilium.k8s.namespace.labels.kubernetes.io/metadata.na
me=default
                                                          k8s:io.cilium.k8s.policy.cluster=cluster1

                                                          k8s:io.cilium.k8s.policy.serviceaccount=default

                                                          k8s:io.kubernetes.pod.namespace=default
```

7. Cilium Network Policies

Cilium Network Policies

On top of the default Kubernetes network policies, Cilium provides extended policy enforcement capabilities (such as Identity-aware, HTTP-aware and DNS-aware) via Cilium Network Policies.

7.1. DNS-aware Network Policy

Task 7.1.1: Create and use a DNS-aware Network Policy

In this task, we want to keep our backend pods from reaching anything except FQDN kubernetes.io.

First we store the `backend` Pod name into an environment variable:

```
BACKEND=$(kubectl get pods -l app=backend -o jsonpath='{.items[0].metadata.name}')
echo ${BACKEND}
```

and then let us check if we can reach `https://kubernetes.io` and `https://cilium.io` :

```
kubectl exec -ti ${BACKEND} -- curl -Ik --connect-timeout 5 https://kubernetes.io | head -1
```

```
kubectl exec -ti ${BACKEND} -- curl -Ik --connect-timeout 5 https://cilium.io | head -1
```

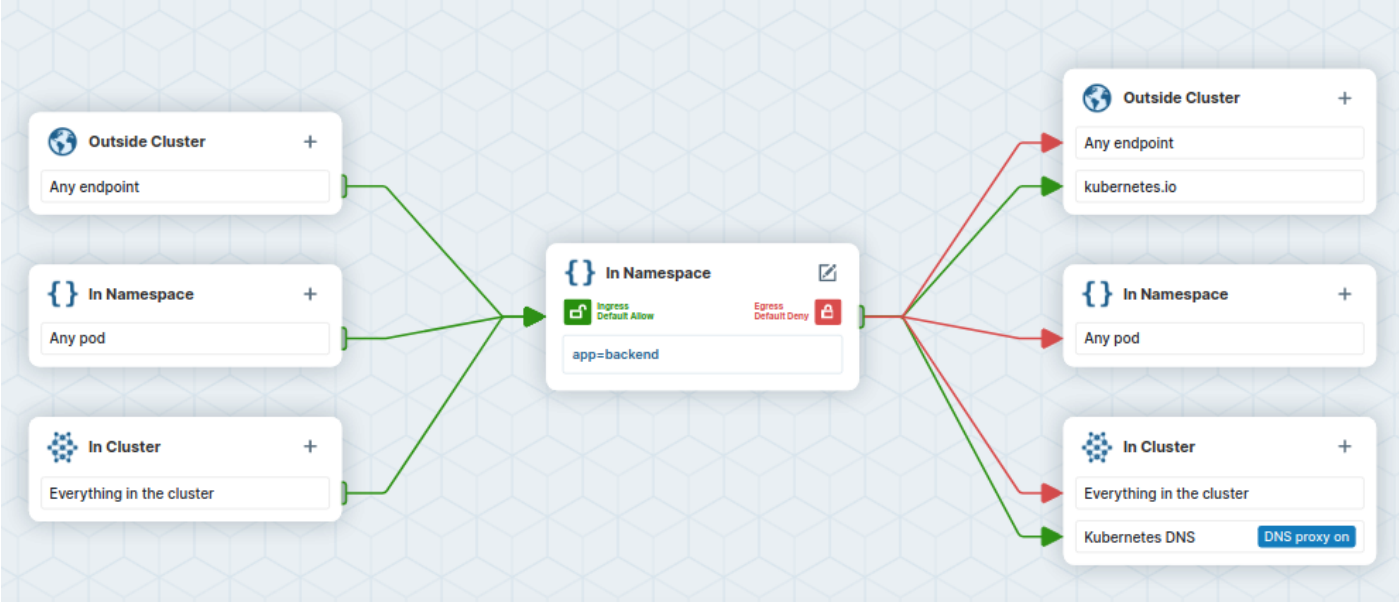
```
# Call to https://kubernetes.io
HTTP/2 200
# Call to https://cilium.io
HTTP/2 200
```

Again, in Kubernetes, all traffic is allowed by default, and since we did not apply any Egress network policy for now, connections from the backend pods are not blocked.

Let us have a look at the following `CiliumNetworkPolicy` :

```
---
kind: CiliumNetworkPolicy
apiVersion: cilium.io/v2
metadata:
  name: backend-egress-allow-fqdn
spec:
  endpointSelector:
    matchLabels:
      app: backend
  egress:
    - toEndpoints:
        - matchLabels:
            "k8s:io.kubernetes.pod.namespace": kube-system
            "k8s:k8s-app": kube-dns
      toPorts:
        - ports:
            - port: "53"
              protocol: ANY
          rules:
            dns:
              - matchPattern: "*"
    - toFQDNs:
        - matchName: kubernetes.io
```

The policy will deny all egress traffic from pods labeled `app=backend` except when traffic is destined for `kubernetes.io` or is a DNS request (necessary for resolving `kubernetes.io` from coredns). In the policy editor this looks like this:



Create the file `backend-egress-allow-fqdn.yaml` with the above content and apply the network policy:

```
kubectl apply -f backend-egress-allow-fqdn.yaml
```

and check if the `CiliumNetworkPolicy` was created:

```
kubectl get cnp
```

- acend gmbh

```
NAME                        AGE
backend-egress-allow-fqdn   2s
```

Note the usage of `cnp` (standing for `CiliumNetworkPolicy`) instead of the default netpol since we are using custom Cilium resources.

And check that the traffic is now only authorized when destined for `kubernetes.io`:

```
kubectl exec -ti ${BACKEND} -- curl -Ik --connect-timeout 5 https://kubernetes.io | head -1
```

```
kubectl exec -ti ${BACKEND} -- curl -Ik --connect-timeout 5 https://cilium.io | head -1
```

```
# Call to https://kubernetes.io
HTTP/2 200
# Call to https://cilium.io
curl: (28) Connection timed out after 5001 milliseconds
command terminated with exit code 28
```

> **Note**
> You can now check the `Hubble Metrics` dashboard in Grafana again. The graphs under DNS should soon show some data as well. This is because with a Layer 7 Policy we have enabled the Envoy in Cilium Agent.

With the ingress and egress policies in place on `app=backend` pods, we have implemented a simple zero-trust model to all traffic to and from our backend. In a real-world scenario, cluster administrators may leverage network policies and overlay them at all levels and for all kinds of traffic.

Task 7.1.2: Cleanup

To not mess up the proceeding labs we are going to delete the `CiliumNetworkPolicy` again and therefore allow all egress traffic again:

```
kubectl delete cnp backend-egress-allow-fqdn
```

7.2. HTTP-aware L7 Policy

Task 7.2.1: Deploy a new Demo Application

In this Star Wars inspired example, there are three microservices applications: deathstar, tiefighter, and xwing. The deathstar runs an HTTP webservice on port 80, which is exposed as a Kubernetes Service to load balance requests to deathstar across two Pod replicas. The deathstar service provides landing services to the empire's spaceships so that they can request a landing port. The tiefighter Pod represents a landing-request client service on a typical empire ship and xwing represents a similar service on an alliance ship. They exist so that we can test different security policies for access control to deathstar landing services.

The file `sw-app.yaml` contains a Kubernetes Deployment for each of the three services. Each deployment is identified using the Kubernetes labels (`org=empire` , `class=deathstar`), (`org=empire` , `class=tiefighter`), and (`org=alliance` , `class=xwing`). It also includes a deathstar-service, which load balances traffic to all pods with labels `org=empire` and `class=deathstar` .

```
---
apiVersion: v1
kind: Service
metadata:
  name: deathstar
  labels:
    app.kubernetes.io/name: deathstar
spec:
  type: ClusterIP
  ports:
  - port: 80
  selector:
    org: empire
    class: deathstar
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deathstar
  labels:
    app.kubernetes.io/name: deathstar
spec:
  replicas: 2
  selector:
    matchLabels:
      org: empire
      class: deathstar
  template:
    metadata:
      labels:
        org: empire
        class: deathstar
        app.kubernetes.io/name: deathstar
    spec:
      containers:
      - name: deathstar
        image: docker.io/cilium/starwars
---
apiVersion: v1
kind: Pod
metadata:
  name: tiefighter
  labels:
    org: empire
    class: tiefighter
    app.kubernetes.io/name: tiefighter
spec:
  containers:
  - name: spaceship
    image: docker.io/tgraf/netperf
---
apiVersion: v1
kind: Pod
metadata:
  name: xwing
  labels:
    app.kubernetes.io/name: xwing
    org: alliance
    class: xwing
spec:
  containers:
  - name: spaceship
    image: docker.io/tgraf/netperf
```

Create and apply the file with:

```
kubectl apply -f sw-app.yaml
```

And as we have already some Network Policies in our Namespace the default ingress behavior is default deny. Therefore we need a new Network Policy to access services on `deathstar`:

Create a file `cnp.yaml` with the following content:

```
---
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "rule1"
spec:
  description: "L3-L4 policy to restrict deathstar access to empire ships only"
  endpointSelector:
    matchLabels:
      org: empire
      class: deathstar
  ingress:
  - fromEndpoints:
    - matchLabels:
        org: empire
    toPorts:
    - ports:
      - port: "80"
        protocol: TCP
```

Apply the `CiliumNetworkPolicy` with:

```
kubectl apply -f cnp.yaml
```

With this policy, our `tiefighter` has access to the `deathstar` application. You can verify this with:

```
kubectl exec tiefighter -- curl -m 2 -s -XPOST deathstar.default.svc.cluster.local/v1/request-landing
```

```
Ship landed
```

but the `xwing` does not have access:

```
kubectl exec xwing -- curl -m 2 -s -XPOST deathstar.default.svc.cluster.local/v1/request-landing
```

```
command terminated with exit code 28
```

Task 7.2.2: Apply and Test HTTP-aware L7 Policy

In the simple scenario above, it was sufficient to either give tiefighter / xwing full access to deathstar'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 deathstar's API should be limited to making only the set of HTTP requests it requires for legitimate operation.

For example, consider that the deathstar service exposes some maintenance APIs that should not be called by random empire ships. To see this run:

```
kubectl exec tiefighter -- curl -s -XPUT deathstar.default.svc.cluster.local/v1/exhaust-port
```

```
Panic: deathstar exploded

goroutine 1 [running]:
main.HandleGarbage(0x2080c3f50, 0x2, 0x4, 0x425c0, 0x5, 0xa)
        /code/src/github.com/empire/deathstar/
        temp/main.go:9 +0x64
main.main()
        /code/src/github.com/empire/deathstar/
        temp/main.go:5 +0x85
```

Cilium is capable of enforcing HTTP-layer (i.e., L7) policies to limit what URLs the tiefighter is allowed to reach. Here is an example policy file that extends our original policy by limiting tiefighter to making only a POST /v1/request-landing API call, but disallowing all other calls (including PUT /v1/exhaust-port).

Create a file `cnp-l7.yaml` with the following content:

```
---
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "rule1"
spec:
  description: "L7 policy to restrict access to specific HTTP call"
  endpointSelector:
    matchLabels:
      org: empire
      class: deathstar
  ingress:
  - fromEndpoints:
    - matchLabels:
        org: empire
    toPorts:
    - ports:
      - port: "80"
        protocol: TCP
      rules:
        http:
        - method: "POST"
          path: "/v1/request-landing"
```

Update the existing rule to apply the L7-aware policy to protect deathstar using with:

```
kubectl apply -f cnp-l7.yaml
```

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

```
kubectl exec tiefighter -- curl -s -XPOST deathstar.default.svc.cluster.local/v1/request-landing
```

- acend gmbh

```
Ship landed
```

and

```
kubectl exec tiefighter -- curl -s -XPUT deathstar.default.svc.cluster.local/v1/exhaust-port
```

```
Access denied
```

> **Note**
> You can now check the `Hubble` dashboard in Grafana again. The graphs under HTTP should soon show some data as well. To generate more data just request-landing on `deathstar` a few times with `tiefighter`

8. Transparent Encryption

Host traffic/endpoint traffic encryption

To secure communication inside a Kubernetes cluster Cilium supports transparent encryption of traffic between Cilium-managed endpoints either using IPsec or WireGuard® .

Task 8.1: Increase cluster size

By default Minikube creates single-node clusters. Add a second node to the cluster:

```
minikube -p cluster1 node add
```

Task 8.2: Move frontend app to a different node

To see traffic between nodes, we move the frontend pod from Chapter 3 to the newly created node:

Create a file `patch.yaml` with the follwing content_

```
spec:
  template:
    spec:
      nodeSelector:
        kubernetes.io/hostname: cluster1-m02
```

You can patch the frontend deployment now:

```
kubectl patch deployments.apps frontend --type merge --patch-file patch.yaml
```

We should see the frontend now running on the new node `cluster1-m02` :

```
kubectl get pods -o wide
```

```
NAME                              READY   STATUS        RESTARTS      AGE   IP           NODE           NOMINATED NODE   R
EADINESS GATES
backend-65f7c794cc-hh6pw          1/1     Running       0             22m   10.1.0.39    cluster1       <none>           <
none>
deathstar-6c94dcc57b-6chpk        1/1     Running       1 (10m ago)   11m   10.1.0.207   cluster1       <none>           <
none>
deathstar-6c94dcc57b-vtt8b        1/1     Running       0             11m   10.1.0.220   cluster1       <none>           <
none>
frontend-6db4b77ff6-kznfl         1/1     Running       0             35s   10.1.1.7     cluster1-m02   <none>           <
none>
not-frontend-8f467ccbd-4jl6z      1/1     Running       0             22m   10.1.0.115   cluster1       <none>           <
none>
tiefighter                        1/1     Running       0             11m   10.1.0.185   cluster1       <none>           <
none>
xwing                             1/1     Running       0             11m   10.1.0.205   cluster1       <none>           <
none>
```

Task 8.3: Sniff traffic between nodes

To check if we see unencrypted traffic between nodes we will use tcpdump. Let us filter on the host interfce for all packets containing the string `password` :

```
CILIUM_AGENT=$(kubectl get pod -n kube-system -l k8s-app=cilium -o jsonpath="{.items[0].metadata.name}")
kubectl debug -n kube-system -i ${CILIUM_AGENT} --image=nicolaka/netshoot -- tcpdump -ni eth0 -vv | grep password
```

In a second terminal we will call our backend service with a password. For those using the Webshell a second Terminal can be opened using the menu `Terminal` then `Split Terminal` , also don't forget to ssh into the VM again. Now in this second terminal run:

```
FRONTEND=$(kubectl get pods -l app=frontend -o jsonpath='{.items[0].metadata.name}')
for i in {1..10}; do
 kubectl exec -ti ${FRONTEND} -- curl -Is backend:8080?password=secret
done
```

You should now see our string `password` sniffed in the network traffic. Hit `Ctrl+C` to stop sniffing but keep the second terminal open.

Task 8.4: Enable node traffic encryption with WireGuard

Enabling WireGuard based encryption with Helm is simple:

```
helm upgrade -i cilium cilium/cilium --version 1.12.10 \
  --namespace kube-system \
  --set ipam.operator.clusterPoolIPv4PodCIDRList={10.1.0.0/16} \
  --set cluster.name=cluster1 \
  --set cluster.id=1 \
  --set operator.replicas=1 \
  --set upgradeCompatibility=1.11 \
  --set kubeProxyReplacement=disabled \
  --set hubble.enabled=true \
  --set hubble.relay.enabled=true \
  --set hubble.ui.enabled=true \
  --set prometheus.enabled=true \
  --set operator.prometheus.enabled=true \
  --set hubble.enabled=true \
  --set hubble.metrics.enabled="{dns,drop:destinationContext=pod;sourceContext=pod,tcp,flow,port-distribution,icmp,http
:destinationContext=pod}" \
  `# enable wireguard:` \
  --set l7Proxy=false \
  --set encryption.enabled=true \
  --set encryption.type=wireguard \
  --set encryption.wireguard.userspaceFallback=true \
  --wait
```

Afterwards restart the Cilium DaemonSet:

```
kubectl -n kube-system rollout restart ds cilium
```

Currently, L7 policy enforcement and visibility is not supported with WireGuard, this is why we have to disable it.

Task 8.5: Verify encryption is working

Verify the number of peers in encryption is 1 (this can take a while, the number is sum of nodes - 1)

```
kubectl -n kube-system exec -ti ds/cilium -- cilium status | grep Encryption
```

You should see something similar to this (in this example we have a two-node cluster):

```
Encryption:             Wireguard       [cilium_wg0 (Pubkey: XbTJd5Gnp7F8cG2Ymj6q11dBx8OtP1J5ZOAhswPiYAc=, Port: 51871,
 Peers: 1)]
```

We now check if the traffic is really encrypted, we start sniffing again:

```
CILIUM_AGENT=$(kubectl get pod -n kube-system -l k8s-app=cilium -o jsonpath="{.items[0].metadata.name}")
kubectl debug -n kube-system -i ${CILIUM_AGENT} --image=nicolaka/netshoot -- tcpdump -ni eth0 -vv | grep password
```

Now in the other terminal generate traffic:

```
FRONTEND=$(kubectl get pods -l app=frontend -o jsonpath='{.items[0].metadata.name}')
for i in {1..10}; do
 kubectl exec -ti ${FRONTEND} -- curl -Is backend:8080?password=secret
done
```

As you should see the traffic is encrypted now and we can't find our string anymore in plaintext on eth0. To sniff the traffic before it is encrypted replace the interface `eth0` with the WireGuard interface `cilium_wg0` .

Hit `Ctrl+C` to stop sniffing. You can close the second terminal with `exit` .

Task 8.6: CleanUp

To not mess up the next ClusterMesh Lab we are going to disable WireGuard encryption again:

```
helm upgrade -i cilium cilium/cilium --version 1.12.10\
  --namespace kube-system \
  --reuse-values \
  --set l7Proxy=true \
  --set encryption.enabled=false \
  --wait
```

and then restart the Cilium Daemonset:

```
kubectl -n kube-system rollout restart ds cilium
```

Verify that it is disabled again:

```
kubectl -n kube-system exec -ti ds/cilium -- cilium status | grep Encryption
```

```
Encryption:                      Disabled
```

remove the second node and move backend back to first node

```
kubectl delete -f simple-app.yaml
minikube node delete cluster1-m02 --profile cluster1
kubectl apply -f simple-app.yaml
```

9. Cluster Mesh

9.1. Enable Cluster Mesh

Task 9.1.1: Create a second Kubernetes Cluster

To create a Cluster Mesh, we need a second Kubernetes cluster. For the Cluster Mesh to work, the PodCIDR ranges in all clusters and nodes must be non-conflicting and have unique IP addresses. The nodes in all clusters must have IP connectivity between each other and the network between the clusters must allow inter-cluster communication.

> **Note**
> The exact ports are documented in the Firewall Rules section.

To start a second cluster run the following command:

```
minikube start --network-plugin=cni --cni=false --kubernetes-version=1.24.3 -p cluster2
```

As Minikube with the Docker driver uses separated Docker networks, we need to make sure that your system forwards traffic between the two networks. To enable forwarding by default execute:

```
sudo iptables -I DOCKER-USER -j ACCEPT
```

Then install Cilium using Helm. Remember, we need a different PodCIDR for the second cluster, therefore while installing Cilium, we have to change this config:

```
helm upgrade -i cilium cilium/cilium --version 1.12.10 \
  --namespace kube-system \
  --set ipam.operator.clusterPoolIPv4PodCIDRList={10.2.0.0/16} \
  --set cluster.name=cluster2 \
  --set cluster.id=2 \
  --set operator.replicas=1 \
  --set kubeProxyReplacement=disabled \
  --wait
```

Then wait until the cluster and Cilium is ready.

```
cilium status --wait
```

- acend gmbh

```
    /¯¯\
 /¯¯\__/¯¯\    Cilium:         OK
 \__/¯¯\__/    Operator:       OK
 /¯¯\__/¯¯\    Hubble:         disabled
 \__/¯¯\__/    ClusterMesh:    disabled
    \__/

DaemonSet         cilium             Desired: 1, Ready: 1/1, Available: 1/1
Deployment        cilium-operator    Desired: 1, Ready: 1/1, Available: 1/1
Containers:       cilium-operator    Running: 1
                  cilium             Running: 1
Cluster Pods:     1/1 managed by Cilium
Image versions    cilium             quay.io/cilium/cilium:v1.12.10: 1
                  cilium-operator    quay.io/cilium/operator-generic:v1.12.10: 1
```

You can verify the correct PodCIDR using:

```
kubectl get pod -A -o wide
```

Have a look at the `coredns-` Pod and verify that it's IP is from your defined `10.2.0.0/16` range.

```
NAMESPACE     NAME                                READY   STATUS    RESTARTS   AGE   IP             NODE       NOMINATED
 NODE   READINESS GATES
kube-system   cilium-operator-776958f5bb-m5hww    1/1     Running   0          29s   192.168.58.2   cluster2   <none>
        <none>
kube-system   cilium-qg9xj                        1/1     Running   0          29s   192.168.58.2   cluster2   <none>
        <none>
kube-system   coredns-558bd4d5db-z6cxh            1/1     Running   0          38s   10.2.0.240     cluster2   <none>
        <none>
kube-system   etcd-cluster2                       1/1     Running   0          44s   192.168.58.2   cluster2   <none>
        <none>
kube-system   kube-apiserver-cluster2             1/1     Running   0          44s   192.168.58.2   cluster2   <none>
        <none>
kube-system   kube-controller-manager-cluster2    1/1     Running   0          44s   192.168.58.2   cluster2   <none>
        <none>
kube-system   kube-proxy-bqk4r                    1/1     Running   0          38s   192.168.58.2   cluster2   <none>
        <none>
kube-system   kube-scheduler-cluster2             1/1     Running   0          44s   192.168.58.2   cluster2   <none>
        <none>
kube-system   storage-provisioner                 1/1     Running   1          49s   192.168.58.2   cluster2   <none>
        <none>
```

The second cluster and Cilium is ready to use.

Task 9.1.2: Enable Cluster Mesh on both Cluster

Now let us enable the Cluster Mesh using the `cilium` CLI on both clusters:

> **Note**
> Although so far we used Helm to install and update Cilium, enabling Cilium Service Mesh using Helm is currently undocumented . We make an exception from the rule to never mix Helm and CLI installations and do it with the CLI.

```
cilium clustermesh enable --context cluster1 --service-type NodePort
cilium clustermesh enable --context cluster2 --service-type NodePort
```

You can now verify the Cluster Mesh status using:

```
cilium clustermesh status --context cluster1 --wait
```

```
⚠  Service type NodePort detected! Service may fail when nodes are removed from the cluster!
  Cluster access information is available:
  - 192.168.49.2:31839
  Service "clustermesh-apiserver" of type "NodePort" found
  [cluster1] Waiting for deployment clustermesh-apiserver to become ready...
  Cluster Connections:
  Global services: [ min:0 / avg:0.0 / max:0 ]
```

To connect the two clusters, the following step needs to be done in one direction only. The connection will automatically be established in both directions:

```
cilium clustermesh connect --context cluster1 --destination-context cluster2
```

The output should look something like this:

```
  Extracting access information of cluster cluster2...
  Extracting secrets from cluster cluster2...
⚠  Service type NodePort detected! Service may fail when nodes are removed from the cluster!
ℹ  Found ClusterMesh service IPs: [192.168.58.2]
  Extracting access information of cluster cluster1...
  Extracting secrets from cluster cluster1...
⚠  Service type NodePort detected! Service may fail when nodes are removed from the cluster!
ℹ  Found ClusterMesh service IPs: [192.168.49.2]
  Connecting cluster cluster1 -> cluster2...
  Secret cilium-clustermesh does not exist yet, creating it...
  Patching existing secret cilium-clustermesh...
  Patching DaemonSet with IP aliases cilium-clustermesh...
  Connecting cluster cluster2 -> cluster1...
  Secret cilium-clustermesh does not exist yet, creating it...
  Patching existing secret ciliugm-clustermesh...
  Patching DaemonSet with IP aliases cilium-clustermesh...
  Connected cluster cluster1 and cluster2!
```

It may take a bit for the clusters to be connected. You can execute the following command

```
cilium clustermesh status --context cluster1 --wait
```

to wait for the connection to be successful. The output should be:

```
⚠  Service type NodePort detected! Service may fail when nodes are removed from the cluster!
  Cluster access information is available:
  - 192.168.58.2:32117
  Service "clustermesh-apiserver" of type "NodePort" found
  [cluster2] Waiting for deployment clustermesh-apiserver to become ready...
  All 1 nodes are connected to all clusters [min:1 / avg:1.0 / max:1]
  Cluster Connections:
- cluster1: 1/1 configured, 1/1 connected
  Global services: [ min:3 / avg:3.0 / max:3 ]
```

The two clusters are now connected.

Task 9.1.3: Cluster Mesh Troubleshooting

Use the following list of steps to troubleshoot issues with Cluster Mesh:

```
cilium status --context cluster1
```

or

```
cilium status --context cluster2
```

which gives you an output similar to this:

```
    /¯¯\
 /¯¯\__/¯¯\    Cilium:         OK
 \__/¯¯\__/    Operator:       OK
 /¯¯\__/¯¯\    Hubble:         OK
 \__/¯¯\__/    ClusterMesh:    OK
    \__/

DaemonSet         cilium                   Desired: 1, Ready: 1/1, Available: 1/1
Deployment        cilium-operator          Desired: 1, Ready: 1/1, Available: 1/1
Deployment        hubble-relay             Desired: 1, Ready: 1/1, Available: 1/1
Deployment        clustermesh-apiserver    Desired: 1, Ready: 1/1, Available: 1/1
Containers:       cilium                   Running: 1
                  cilium-operator          Running: 1
                  hubble-relay             Running: 1
                  clustermesh-apiserver    Running: 1
Cluster Pods:     6/6 managed by Cilium
Image versions    cilium                   quay.io/cilium/cilium:v1.12.10: 1
                  cilium-operator          quay.io/cilium/operator-generic:v1.12.10: 1
                  hubble-relay             quay.io/cilium/hubble-relay:v1.12.10: 1
                  clustermesh-apiserver    quay.io/coreos/etcd:v3.4.13: 1
                  clustermesh-apiserver    quay.io/cilium/clustermesh-apiserver:v1.12.10: 1
```

If you cannot resolve the issue with the above commands, follow the steps in [Cilium's Cluster Mesh Troubleshooting Guide] .

9.2. Load-balancing with Global Services

This lab will guide you to perform load-balancing and service discovery across multiple Kubernetes clusters.

Task 9.2.1: Load-balancing with Global Services

Establishing load-balancing between clusters is achieved by defining a Kubernetes service with an identical name and Namespace in each cluster and adding the `annotation io.cilium/global-service: "true"` to declare it global. Cilium will automatically perform load-balancing to pods in both clusters.

We are going to deploy a global service and a sample application on both of our connected clusters.

First the Kubernetes service. Create a file `svc.yaml` with the following content:

```
---
apiVersion: v1
kind: Service
metadata:
  name: rebel-base
  annotations:
    io.cilium/global-service: "true"
spec:
  type: ClusterIP
  ports:
  - port: 80
  selector:
    name: rebel-base
```

Apply this with:

```
kubectl --context cluster1 apply -f svc.yaml
kubectl --context cluster2 apply -f svc.yaml
```

Then deploy our sample application on both clusters.

`cluster1.yaml` :

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rebel-base
spec:
  selector:
    matchLabels:
      name: rebel-base
  replicas: 2
  template:
    metadata:
      labels:
        name: rebel-base
    spec:
      containers:
      - name: rebel-base
        image: docker.io/nginx:1.15.8
        volumeMounts:
          - name: html
            mountPath: /usr/share/nginx/html/
```

```
        livenessProbe:
          httpGet:
            path: /
            port: 80
          periodSeconds: 1
        readinessProbe:
          httpGet:
            path: /
            port: 80
      volumes:
        - name: html
          configMap:
            name: rebel-base-response
            items:
              - key: message
                path: index.html
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: rebel-base-response
data:
  message: "{\"Galaxy\": \"Alderaan\", \"Cluster\": \"Cluster-1\"}\n"
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: x-wing
spec:
  selector:
    matchLabels:
      name: x-wing
  replicas: 2
  template:
    metadata:
      labels:
        name: x-wing
    spec:
      containers:
      - name: x-wing-container
        image: docker.io/cilium/json-mock:1.2
        livenessProbe:
          exec:
            command:
            - curl
            - -sS
            - -o
            - /dev/null
            - localhost
        readinessProbe:
          exec:
            command:
            - curl
            - -sS
            - -o
            - /dev/null
            - localhost
```

```
kubectl --context cluster1 apply -f cluster1.yaml
```

cluster2.yaml：

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rebel-base
spec:
```

```
  selector:
    matchLabels:
      name: rebel-base
  replicas: 2
  template:
    metadata:
      labels:
        name: rebel-base
    spec:
      containers:
      - name: rebel-base
        image: docker.io/nginx:1.15.8
        volumeMounts:
          - name: html
            mountPath: /usr/share/nginx/html/
        livenessProbe:
          httpGet:
            path: /
            port: 80
          periodSeconds: 1
        readinessProbe:
          httpGet:
            path: /
            port: 80
      volumes:
        - name: html
          configMap:
            name: rebel-base-response
            items:
              - key: message
                path: index.html
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: rebel-base-response
data:
  message: "{\"Galaxy\": \"Alderaan\", \"Cluster\": \"Cluster-2\"}\n"
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: x-wing
spec:
  selector:
    matchLabels:
      name: x-wing
  replicas: 2
  template:
    metadata:
      labels:
        name: x-wing
    spec:
      containers:
      - name: x-wing-container
        image: docker.io/cilium/json-mock:1.2
        livenessProbe:
          exec:
            command:
            - curl
            - -sS
            - -o
            - /dev/null
            - localhost
        readinessProbe:
          exec:
            command:
            - curl
            - -sS
            - -o
            - /dev/null
            - localhost
```

```
kubectl --context cluster2 apply -f cluster2.yaml
```

Now you can execute from either cluster the following command (there are two x-wing pods, simply select one):

```
XWINGPOD=$(kubectl --context cluster1 get pod -l name=x-wing -o jsonpath="{.items[0].metadata.name}")
for i in {1..10}; do
  kubectl --context cluster1  exec -it $XWINGPOD -- curl -m 1 rebel-base
done
```

as a result you get the following output:

```
{"Galaxy": "Alderaan", "Cluster": "Cluster-2"}
{"Galaxy": "Alderaan", "Cluster": "Cluster-1"}
{"Galaxy": "Alderaan", "Cluster": "Cluster-2"}
{"Galaxy": "Alderaan", "Cluster": "Cluster-1"}
{"Galaxy": "Alderaan", "Cluster": "Cluster-1"}
{"Galaxy": "Alderaan", "Cluster": "Cluster-1"}
{"Galaxy": "Alderaan", "Cluster": "Cluster-1"}
{"Galaxy": "Alderaan", "Cluster": "Cluster-1"}
{"Galaxy": "Alderaan", "Cluster": "Cluster-1"}
{"Galaxy": "Alderaan", "Cluster": "Cluster-2"}
```

and as you see, you get results from both clusters. Even if you scale down your `rebel-base` Deployment on `cluster1` with

```
kubectl --context cluster1 scale deployment rebel-base --replicas=0
```

and then execute the `curl` `for` loop again, you still get answers, this time only from `cluster2` :

```
{"Galaxy": "Alderaan", "Cluster": "Cluster-2"}
{"Galaxy": "Alderaan", "Cluster": "Cluster-2"}
{"Galaxy": "Alderaan", "Cluster": "Cluster-2"}
{"Galaxy": "Alderaan", "Cluster": "Cluster-2"}
{"Galaxy": "Alderaan", "Cluster": "Cluster-2"}
{"Galaxy": "Alderaan", "Cluster": "Cluster-2"}
{"Galaxy": "Alderaan", "Cluster": "Cluster-2"}
{"Galaxy": "Alderaan", "Cluster": "Cluster-2"}
{"Galaxy": "Alderaan", "Cluster": "Cluster-2"}
{"Galaxy": "Alderaan", "Cluster": "Cluster-2"}
```

Scale your `rebel-base` Deployment back to one replica:

```
kubectl --context cluster1 scale deployment rebel-base --replicas=1
```

9.3. Network Policies

Task 9.3.1: Allowing Specific Communication Between Clusters

The following policy illustrates how to allow particular pods to communicate between two clusters.

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "allow-cross-cluster"
spec:
  description: "Allow x-wing in cluster1 to only contact rebel-base in cluster1"
  endpointSelector:
    matchLabels:
      name: x-wing
      io.cilium.k8s.policy.cluster: cluster1
  egress:
  - toEndpoints:
    - matchLabels:
        "k8s:io.kubernetes.pod.namespace": kube-system
        "k8s:k8s-app": kube-dns
    toPorts:
    - ports:
        - port: "53"
          protocol: ANY
      rules:
        dns:
          - matchPattern: "*"
  - toEndpoints:
    - matchLabels:
        name: rebel-base
        io.cilium.k8s.policy.cluster: cluster1
```

> **Note**
> For the Pods to resolve the `rebel-base` service name they still need connectivity to Kubernetes DNS Service. Therefore access to that is also allowed.

Kubernetes security policies are not automatically distributed across clusters, it is your responsibility to apply `CiliumNetworkPolicy` or `NetworkPolicy` in all clusters.

Create a file `cnp-cm.yaml` with the above content and apply the `CiliumNetworkPolicy` to both clusters:

```
kubectl --context cluster1 apply -f cnp-cm.yaml
kubectl --context cluster2 apply -f cnp-cm.yaml
```

Let us run our `curl` `for` loop again

```
XWINGPOD=$(kubectl --context cluster1 get pod -l name=x-wing -o jsonpath="{.items[0].metadata.name}")
for i in {1..10}; do
  kubectl --context cluster1 exec -it $XWINGPOD -- curl -m 1 rebel-base
done
```

and as an result you see:

```
curl: (28) Connection timed out after 1001 milliseconds
command terminated with exit code 28
curl: (28) Connection timed out after 1000 milliseconds
command terminated with exit code 28
{"Galaxy": "Alderaan", "Cluster": "Cluster-1"}
curl: (28) Connection timed out after 1000 milliseconds
command terminated with exit code 28
{"Galaxy": "Alderaan", "Cluster": "Cluster-1"}
{"Galaxy": "Alderaan", "Cluster": "Cluster-1"}
{"Galaxy": "Alderaan", "Cluster": "Cluster-1"}
curl: (28) Connection timed out after 1000 milliseconds
command terminated with exit code 28
{"Galaxy": "Alderaan", "Cluster": "Cluster-1"}
curl: (28) Connection timed out after 1000 milliseconds
command terminated with exit code 28
```

All connections to `cluster2` are dropped while the ones to `cluster1` are still working.

Task 9.3.2: Cleanup

We will disconnect our cluster mesh again and delete the second cluster:

```
cilium clustermesh disconnect --context cluster1 --destination-context cluster2
minikube delete --profile cluster2
minikube profile cluster1
```

10. Advanced Networking

10.1. Host Firewall

Cilium is capable to act as a host firewall to enforce security policies for Kubernetes nodes. In this lab, we are going to show you briefly how this works.

Task 10.1.1: Enable the Host Firewall in Cilium

We need to enable the host firewall in the Cilium config. This can be done using Helm:

```
helm upgrade -i cilium cilium/cilium --version 1.12.10 \
  --namespace kube-system \
  --set ipam.operator.clusterPoolIPv4PodCIDRList={10.1.0.0/16} \
  --set cluster.name=cluster1 \
  --set cluster.id=1 \
  --set operator.replicas=1 \
  --set upgradeCompatibility=1.11 \
  --set kubeProxyReplacement=disabled \
  --set hubble.enabled=true \
  --set hubble.relay.enabled=true \
  --set hubble.ui.enabled=true \
  --set prometheus.enabled=true \
  --set operator.prometheus.enabled=true \
  --set hubble.enabled=true \
  --set hubble.metrics.enabled="{dns,drop:destinationContext=pod;sourceContext=pod,tcp,flow,port-distribution,icmp,http
:destinationContext=pod}" \
  `# enable host firewall:` \
  --set hostFirewall.enabled=true \
  --set devices='{eth0}' \
  --wait
```

The devices flag refers to the network devices Cilium is configured on such as `eth0`. Omitting this option leads Cilium to auto-detect what interfaces the host firewall applies to.

Make sure to restart the `cilium` Pods with:

```
kubectl -n kube-system rollout restart ds/cilium
```

At this point, the Cilium-managed nodes are ready to enforce Network Policies.

Task 10.1.2: Attach a Label to the Node

In this lab, we will apply host policies only to nodes with the label `node-access=ssh`. We thus first need to attach that label to a node in the cluster.

```
kubectl label node cluster1 node-access=ssh
```

Task 10.1.3: Enable Policy Audit Mode for the Host Endpoint

Host Policies enforce access control over connectivity to and from nodes. Particular care must be taken to ensure that when host policies are imported, Cilium does not block access to the nodes or break the cluster's normal behavior (for example by blocking communication with kube-apiserver).

To avoid such issues, we can switch the host firewall in audit mode, to validate the impact of host policies before enforcing them. When Policy Audit Mode is enabled, no network policy is enforced so this setting is not recommended for production deployment.

```
CILIUM_POD_NAME=$(kubectl -n kube-system get pods -l "k8s-app=cilium" -o jsonpath="{.items[?(@.spec.nodeName=='cluster1
')].metadata.name}")
HOST_EP_ID=$(kubectl -n kube-system exec $CILIUM_POD_NAME -- cilium endpoint list -o jsonpath='{[?(@.status.identity.id
==1)].id}')
kubectl -n kube-system exec $CILIUM_POD_NAME -- cilium endpoint config $HOST_EP_ID PolicyAuditMode=Enabled
```

Verification:

```
kubectl -n kube-system exec $CILIUM_POD_NAME -- cilium endpoint config $HOST_EP_ID | grep PolicyAuditMode
```

The output should show you:

```
PolicyAuditMode          Enabled
```

Task 10.1.4: Apply a Host Network Policy

Host Policies match on node labels using a Node Selector to identify the nodes to which the policy applies. The following policy applies to all nodes. It allows communications from outside the cluster only on port TCP/22. All communications from the cluster to the hosts are allowed.

Host policies don't apply to communications between pods or between pods and the outside of the cluster, except if those pods are host-networking pods.

Create a file `ccwnp.yaml` with the following content:

```
---
apiVersion: "cilium.io/v2"
kind: CiliumClusterwideNetworkPolicy
metadata:
  name: "demo-host-policy"
spec:
  description: ""
  nodeSelector:
    matchLabels:
      node-access: ssh
  ingress:
  - toPorts:
    - ports:
      - port: "22"
        protocol: TCP
```

- acend gmbh

And then apply this `CiliumClusterwideNetworkPolicy` with:

```
kubectl apply -f ccwnp.yaml
```

The host is represented as a special endpoint, with label `reserved:host`, in the output of the command `cilium endpoint list`. You can therefore inspect the status of the policy using that command:

```
kubectl -n kube-system exec $(kubectl -n kube-system get pods -l k8s-app=cilium -o jsonpath='{.items[0].metadata.name}'
) -- cilium endpoint list
```

You will see that the ingress policy enforcement for the `reserved:host` endpoint is `Disabled` but with `Audit` enabled:

```
Defaulted container "cilium-agent" out of: cilium-agent, mount-cgroup (init), clean-cilium-state (init)
ENDPOINT   POLICY (ingress)   POLICY (egress)   IDENTITY   LABELS (source:key[=value])
                IPv6   IPv4         STATUS
           ENFORCEMENT        ENFORCEMENT
671        Disabled (Audit)   Disabled          1          k8s:minikube.k8s.io/commit=3e64b11ed75e56e4898ea85f96b2e4af0
301f43d                               ready
                                                            k8s:minikube.k8s.io/name=cluster1
                                                            k8s:minikube.k8s.io/updated_at=2022_02_14T13_45_35_0700
                                                            k8s:minikube.k8s.io/version=v1.25.1
                                                            k8s:node-access=ssh
                                                            k8s:node-role.kubernetes.io/control-plane
                                                            k8s:node-role.kubernetes.io/master
                                                            k8s:node.kubernetes.io/exclude-from-external-load-balancers
                                                            reserved:host
810        Disabled           Disabled          129160     k8s:io.cilium.k8s.namespace.labels.kubernetes.io/metadata.na
me=kube-system          10.1.0.249   ready
                                                            k8s:io.cilium.k8s.policy.cluster=cluster1
                                                            k8s:io.cilium.k8s.policy.serviceaccount=coredns
                                                            k8s:io.kubernetes.pod.namespace=kube-system
                                                            k8s:k8s-app=kube-dns
4081       Disabled           Disabled          4          reserved:health
```

As long as the host endpoint is running in audit mode, communications disallowed by the policy won't be dropped. They will however be reported by `cilium monitor` as `action audit`. The audit mode thus allows you to adjust the host policy to your environment, to avoid unexpected connection breakages.

You can montitor the policy verdicts with:

```
kubectl -n kube-system exec $(kubectl -n kube-system get pods -l k8s-app=cilium -o jsonpath='{.items[0].metadata.name}'
) -- cilium monitor -t policy-verdict --related-to $HOST_EP_ID
```

Open a second terminal to produce some traffic:

Note

If you are working in our Webshell environment, make sure to first login again to your VM after opening the second terminal.

```
curl -k https://192.168.49.2:8443
```

Also try to start an SSH session (you can cancel the command when the password promt is shown):

```
ssh 192.168.49.2
```

In the verdict log you should see an output similar to the following one. For the `curl` request you see that the action is set to `audit`:

```
Policy verdict log: flow 0xfd71ed86 local EP ID 671, remote ID world, proto 6, ingress, action audit, match none, 192.1
68.49.1:50760 -> 192.168.49.2:8443 tcp SYN
Policy verdict log: flow 0xfd71ed86 local EP ID 671, remote ID world, proto 6, ingress, action audit, match none, 192.1
68.49.1:50760 -> 192.168.49.2:8443 tcp SYN
```

The request to the SSH port has action `allow`:

```
Policy verdict log: flow 0x6b5b1b60 local EP ID 671, remote ID world, proto 6, ingress, action allow, match L4-Only, 19
2.168.49.1:48254 -> 192.168.49.2:22 tcp SYN
Policy verdict log: flow 0x6b5b1b60 local EP ID 671, remote ID world, proto 6, ingress, action allow, match L4-Only, 19
2.168.49.1:48254 -> 192.168.49.2:22 tcp SYN
```

Task 10.1.5: Clean Up

Once you are confident all required communication to the host from outside the cluster is allowed, you can disable policy audit mode to enforce the host policy.

> **Note**
> When enforcing the host policy, make sure that none of the communications required to access the cluster or for the cluster to work properly are denied. They should appear as `action allow`.

We are not going to do this extended task (as it would require some more rules for the cluster to continue working). But the command to disable the audit mode looks like this:

```
# kubectl -n kube-system  exec $CILIUM_POD_NAME -- cilium endpoint config $HOST_EP_ID PolicyAuditMode=Disabled
```

Simply cleanup and continue:

```
kubectl delete ccnp demo-host-policy
kubectl label node cluster1 node-access-
```

10.2. Kubernetes without kube-proxy

In this lab, we are going to provision a new Kubernetes cluster without `kube-proxy` to use Cilium as a full replacement for it.

Task 10.2.1: Deploy a new Kubernetes Cluster without kube-proxy

Create a new Kubernetes cluster using `minikube`. As `minikube` uses `kubeadm` we can skip the phase where `kubeadm` installs the `kube-proxy` addon. Execute the following command to create a third cluster:

```
minikube start --network-plugin=cni --cni=false --kubernetes-version=1.24.3 --extra-config=kubeadm.skip-phases=addon/ku
be-proxy -p kubeless
```

```
😄  [cluster3] minikube v1.24.3 on Ubuntu 20.04
    Automatically selected the docker driver. Other choices: virtualbox, ssh
    With --network-plugin=cni, you will need to provide your own CNI. See --cni flag as a user-friendly alternative
    Starting control plane node cluster3 in cluster cluster3
    Pulling base image ...
    Creating docker container (CPUs=2, Memory=8000MB) ...
    Preparing Kubernetes v1.24.3 on Docker 20.10.8 ...
      ▪ kubeadm.skip-phases=addon/kube-proxy
      ▪ Generating certificates and keys ...
      ▪ Booting up control plane ...
      ▪ Configuring RBAC rules ...
    Verifying Kubernetes components...
      ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
    Enabled addons: storage-provisioner, default-storageclass
    Done! kubectl is now configured to use "cluster3" cluster and "default" namespace by default
```

Task 10.2.2: Deploy Cilium and enable the Kube Proxy replacement

As the `cilium` and `cilium-operator` Pods by default try to communicate with the Kubernetes API using the default `kubernetes` service IP, they cannot do this with disabled `kube-proxy`. We, therefore, need to set the `KUBERNETES_SERVICE_HOST` and `KUBERNETES_SERVICE_PORT` environment variables to tell the two Pods how to connect to the Kubernetes API.

To find the correct IP address execute the following command:

```
API_SERVER_IP=$(kubectl config view -o jsonpath='{.clusters[?(@.name == "kubeless")].cluster.server}' | cut -f 3 -d / |
 cut -f1 -d:)
API_SERVER_PORT=$(kubectl config view -o jsonpath='{.clusters[?(@.name == "kubeless")].cluster.server}' | cut -f 3 -d /
 | cut -f2 -d:)
echo "$API_SERVER_IP:$API_SERVER_PORT"
```

Use the shown IP address and port in the next Helm command to install Cilium:

```
helm upgrade -i cilium cilium/cilium --version 1.12.10 \
  --namespace kube-system \
  --set ipam.operator.clusterPoolIPv4PodCIDRList={10.3.0.0/16} \
  --set cluster.name=kubeless \
  --set cluster.id=3 \
  --set operator.replicas=1 \
  --set kubeProxyReplacement=strict \
  --set k8sServiceHost=$API_SERVER_IP \
  --set k8sServicePort=$API_SERVER_PORT \
  --wait
```

Note

Having a cluster running with kubeProxyReplacement set to partial breaks other minikube clusters running on the same host. If you still want to play around with `cluster1` after this chapter, you need to reboot your maching and start only cluster1 with `minikube start --profile cluster1`

We can now compare the running Pods on `cluster1` and `kubeless` in the `kube-system` Namespace.

```
kubectl --context cluster1 -n kube-system get pod
```

Here we see the running `kube-proxy` pod:

```
NAME                                     READY   STATUS    RESTARTS       AGE
cilium-operator-cb65bcb9b-cnxnq          1/1     Running   0              19m
cilium-tq9kk                             1/1     Running   0              8m42s
clustermesh-apiserver-67fd99fd9b-x2svr   2/2     Running   0              61m
coredns-6d4b75cb6d-fd6vk                 1/1     Running   1 (82m ago)    97m
etcd-cluster1                            1/1     Running   1 (82m ago)    98m
hubble-relay-84b4ddb556-nvftg            1/1     Running   0              19m
hubble-ui-579fdfbc58-t6xst               2/2     Running   0              19m
kube-apiserver-cluster1                  1/1     Running   1 (81m ago)    98m
kube-controller-manager-cluster1         1/1     Running   1 (82m ago)    98m
kube-proxy-5j84l                         1/1     Running   1 (82m ago)    97m
kube-scheduler-cluster1                  1/1     Running   1 (81m ago)    98m
storage-provisioner                      1/1     Running   2 (82m ago)    98m
```

On `kubeless` there is no `kube-proxy` Pod anymore:

```
kubectl --context kubeless -n kube-system get pod
```

```
NAME                                 READY   STATUS    RESTARTS       AGE
cilium-operator-68bfb94678-785dk     1/1     Running   0              17m
cilium-vrqms                         1/1     Running   0              17m
coredns-64897985d-fk5lj              1/1     Running   0              59m
etcd-cluster3                        1/1     Running   0              59m
kube-apiserver-cluster3              1/1     Running   0              59m
kube-controller-manager-cluster3     1/1     Running   0              59m
kube-scheduler-cluster3              1/1     Running   0              59m
storage-provisioner                  1/1     Running   13 (17m ago)   59m
```

Task 10.2.3: Deploy our simple app again to the new

cluster

As this is a new cluster we want to deploy our `simple-app.yaml` from lab 03 again to run some experiments. Run the following command using the `simple-app.yaml` from lab 03:

```
kubectl apply -f simple-app.yaml
```

Now let us redo the task from lab 03.

Let’s make life again a bit easier by storing the Pod’s name into an environment variable so we can reuse it later again:

```
FRONTEND=$(kubectl get pods -l app=frontend -o jsonpath='{.items[0].metadata.name}')
echo ${FRONTEND}
NOT_FRONTEND=$(kubectl get pods -l app=not-frontend -o jsonpath='{.items[0].metadata.name}')
echo ${NOT_FRONTEND}
```

Then execute:

```
kubectl exec -ti ${FRONTEND} -- curl -I --connect-timeout 5 backend:8080
```

and

```
kubectl exec -ti ${NOT_FRONTEND} -- curl -I --connect-timeout 5 backend:8080
```

You see that altought we have no `kube-proxy` running, the backend service can still be reached.

```
HTTP/1.1 200 OK
X-Powered-By: Express
Vary: Origin, Accept-Encoding
Access-Control-Allow-Credentials: true
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Last-Modified: Sat, 26 Oct 1985 08:15:00 GMT
ETag: W/"83d-7438674ba0"
Content-Type: text/html; charset=UTF-8
Content-Length: 2109
Date: Tue, 14 Dec 2021 10:01:16 GMT
Connection: keep-alive

HTTP/1.1 200 OK
X-Powered-By: Express
Vary: Origin, Accept-Encoding
Access-Control-Allow-Credentials: true
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Last-Modified: Sat, 26 Oct 1985 08:15:00 GMT
ETag: W/"83d-7438674ba0"
Content-Type: text/html; charset=UTF-8
Content-Length: 2109
Date: Tue, 14 Dec 2021 10:01:16 GMT
Connection: keep-alive
```

11. Cilium Service Mesh

With release 1.12 Cilium enabled direct ingress support and service mesh features like layer 7 loadbalancing

Task 11.1: Installation

```
helm upgrade -i cilium cilium/cilium --version 1.12.10 \
  --namespace kube-system \
  --reuse-values \
  --set ingressController.enabled=true \
  --wait
```

For Kubernetes Ingress to work kubeProxyReplacement needs to be set to `strict` or `partial`. This is why we stay on the `kubeless` cluster.

Wait until cilium is ready (check with `cilium status`). For Ingress to work it is necessary to restart the agent and the operator.

```
kubectl -n kube-system rollout restart deployment/cilium-operator
kubectl -n kube-system rollout restart ds/cilium
```

Task 11.2: Create Ingress

Cilium Service Mesh can handle ingress traffic with its Envoy proxy.

We will use this feature to allow traffic to our simple app from outside the cluster. Create a file named `ingress.yaml` with the text below inside:

```
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: backend
spec:
  ingressClassName: cilium
  rules:
  - http:
      paths:
      - backend:
          service:
            name: backend
            port:
              number: 8080
        path: /
        pathType: Prefix
```

Apply it with:

```
kubectl apply -f ingress.yaml
```

Check the ingress and the service:

```
kubectl describe ingress backend
kubectl get svc cilium-ingress-backend
```

We see that Cilium created a Service with type Loadbalancer for our Ingress. Unfortunately, Minikube has no loadbalancer deployed, in our setup the external IP will stay pending.

As a workaround, we can test the service from inside Kubernetes.

```
SERVICE_IP=$(kubectl get svc cilium-ingress-backend -ojsonpath={.spec.clusterIP})
kubectl run --rm=true -it --restart=Never --image=curlimages/curl -- curl --connect-timeout 5 http://${SERVICE_IP}/publ
ic
```

You should get the following output:

```
[
  {
    "id": 1,
    "body": "public information"
  }
]pod "curl" deleted
```

Task 11.3: Layer 7 Loadbalancing

Ingress alone is not really a Service Mesh feature. Let us test a traffic control example by loadbalancing a service inside the proxy.

Start by creating the second service. Create a file named `backend2.yaml` and put in the text below:

```
---
apiVersion: v1
data:
  default.json: |
    {
      "private": [
        { "id": 1, "body": "another secret information from a different backend" }
      ],
      "public": [
        { "id": 1, "body": "another public information from a different backend" }
      ]
    }
kind: ConfigMap
metadata:
  name: default-json
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-2
  labels:
    app: backend-2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: backend-2
  template:
    metadata:
      labels:
        app: backend-2
    spec:
      volumes:
      - name: default-json
        configMap:
          name: default-json
      containers:
      - name: backend-container
        env:
        - name: PORT
          value: "8080"
        ports:
        - containerPort: 8080
        image: docker.io/cilium/json-mock:1.2
        imagePullPolicy: IfNotPresent
        volumeMounts:
        - name: default-json
          mountPath: /default.json
          subPath: default.json
---
apiVersion: v1
kind: Service
metadata:
  name: backend-2
  labels:
    app: backend-2
spec:
  type: ClusterIP
  selector:
    app: backend-2
  ports:
  - name: http
    port: 8080
```

Apply it:

```
kubectl apply -f backend2.yaml
```

Call it:

```
kubectl run --rm=true -it --restart=Never --image=curlimages/curl -- curl --connect-timeout 3 http://backend-2:8080/pub
lic
```

We see output very similiar to our simple application backend, but with a changed text.

As layer 7 loadbalancing requires traffic to be routed through the proxy, we will enable this for our backend Pods using a `CiliumNetworkPolicy` with HTTP rules. We will block access to `/public` and allow requests to `/private`:

Create a file `cnp-l7-sm.yaml` with the following content:

```
---
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "rule1"
spec:
  description: "enable L7 without blocking"
  endpointSelector:
    matchLabels:
      app: backend
  ingress:
  - fromEntities:
    - "all"
    toPorts:
    - ports:
      - port: "8080"
        protocol: TCP
      rules:
        http:
        - method: "GET"
          path: "/private"
---
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "rule2"
spec:
  description: "enable L7 without blocking"
  endpointSelector:
    matchLabels:
      app: backend-2
  ingress:
  - fromEntities:
    - "all"
    toPorts:
    - ports:
      - port: "8080"
        protocol: TCP
      rules:
        http:
        - method: "GET"
          path: "/private"
```

And apply the `CiliumNetworkPolicy` with:

```
kubectl apply -f cnp-l7-sm.yaml
```

Until now only the backend service is replying to Ingress traffic. Now we configure Envoy to loadbalance the traffic 50/50 between backend and backend-2 with retries. We are using a CustomResource called `CiliumEnvoyConfig` for this. Create a file `envoyconfig.yaml` with the following content:

```
apiVersion: cilium.io/v2
kind: CiliumEnvoyConfig
metadata:
  name: envoy-lb-listener
spec:
  services:
    - name: backend
      namespace: default
    - name: backend-2
      namespace: default
  resources:
    - "@type": type.googleapis.com/envoy.config.listener.v3.Listener
      name: envoy-lb-listener
      filter_chains:
        - filters:
            - name: envoy.filters.network.http_connection_manager
              typed_config:
                "@type": type.googleapis.com/envoy.extensions.filters.network.http_connection_manager.v3.HttpConnection
Manager
                stat_prefix: envoy-lb-listener
                rds:
                  route_config_name: lb_route
                http_filters:
                  - name: envoy.filters.http.router
                    typed_config:
                      "@type": type.googleapis.com/envoy.extensions.filters.http.router.v3.Router
    - "@type": type.googleapis.com/envoy.config.route.v3.RouteConfiguration
      name: lb_route
      virtual_hosts:
        - name: "lb_route"
          domains: ["*"]
          routes:
            - match:
                prefix: "/private"
              route:
                weighted_clusters:
                  clusters:
                    - name: "default/backend"
                      weight: 50
                    - name: "default/backend-2"
                      weight: 50
                retry_policy:
                  retry_on: 5xx
                  num_retries: 3
                  per_try_timeout: 1s
    - "@type": type.googleapis.com/envoy.config.cluster.v3.Cluster
      name: "default/backend"
      connect_timeout: 5s
      lb_policy: ROUND_ROBIN
      type: EDS
      outlier_detection:
        split_external_local_origin_errors: true
        consecutive_local_origin_failure: 2
    - "@type": type.googleapis.com/envoy.config.cluster.v3.Cluster
      name: "default/backend-2"
      connect_timeout: 3s
      lb_policy: ROUND_ROBIN
      type: EDS
      outlier_detection:
        split_external_local_origin_errors: true
        consecutive_local_origin_failure: 2
```

Note

If you want to read more about Envoy configuration Envoy Architectural Overview is a good place to start.

Apply the `CiliumEnvoyConfig` with:

```
kubectl apply -f envoyconfig.yaml
```

Test it by running `curl` a few times – different backends should respond:

```
for i in {1..10}; do
  kubectl run --rm=true -it --image=curlimages/curl --restart=Never curl -- curl --connect-timeout 5 http://backend:808
0/private
done
```

We see both backends replying. If you call it many times the distribution would be equal.

```
[
  {
    "id": 1,
    "body": "another secret information from a different backend"
  }
]pod "curl" deleted
[
  {
    "id": 1,
    "body": "secret information"
  }
]pod "curl" deleted
```

This basic traffic control example shows only one function of Cilium Service Mesh, other features include i.e. TLS termination, support for tracing and canary-rollouts.

Task 11.4: Cleanup

We don’t need this cluster anymore and therefore you can delete the cluster with:

```
minikube delete --profile kubeless
```

12. eBPF

To deepen our understanding of eBPF we will write and compile a small eBPF app:

Task 12.1: Hello World

ebpf-go is a pure Go library that provides utilities for loading, compiling, and debugging eBPF programs written by the cilium project.

We will use this library and add our own hello world app as an example to it:

```
git clone https://github.com/cilium/ebpf.git
cd ebpf/
git checkout v0.9.3
cd examples
mkdir helloworld
cd helloworld
```

In the `helloworld` directory create two files named `helloworld.bpf.c` (eBPF code) and `helloworld.go` (loading, user side):

helloworld.bpf.c:

```
#include "common.h"

// SEC is a macro that expands to create an ELF section which bpf loaders parse.
// we want our function to be executed whenever syscall execve (program execution) is called
SEC("tracepoint/syscalls/sys_enter_execve")
int bpf_prog(void *ctx) {
  char msg[] = "Hello world";
  // bpf_printk is a bpf helper function which writes strings to /sys/kernel/debug/tracing/trace_pipe (good for debuggi
ng purposes)
  bpf_printk("%s", msg);
  // bpf programs need to return an int
  return 0;
}

char LICENSE[] SEC("license") = "GPL";
```

helloworld.go:

```
package main

import (
 "log"

 "github.com/cilium/ebpf/link"
 "github.com/cilium/ebpf/rlimit"
)

func main() {

 // Allow the current process to lock memory for eBPF resources.
 if err := rlimit.RemoveMemlock(); err != nil {
  log.Fatal(err)
 }

 // Here we load our bpf code into the kernel, these functions are in the
 // .go file created by bpf2go
 objs := bpfObjects{}
 if err := loadBpfObjects(&objs, nil); err != nil {
  log.Fatalf("loading objects: %s", err)
 }
 defer objs.Close()

 //SEC("tracepoint/syscalls/sys_enter_execve")
 kp, err := link.Tracepoint("syscalls", "sys_enter_execve", objs.BpfProg, nil)
 if err != nil {
  log.Fatalf("opening tracepoint: %s", err)
 }
 defer kp.Close()

 for {
 }

 log.Println("Received signal, exiting program..")
}
```

To compile the C code into ebpf bytecode with the corresponding Go source files we use a tool named bpf2go along with clang. For a stable outcome we use the toolchain inside a docker container:

```
docker pull "ghcr.io/cilium/ebpf-builder:1666886595"
docker run -it --rm -v "$(pwd)/../..":/ebpf \
   -w /ebpf/examples/helloworld \
  --env MAKEFLAGS \
  --env CFLAGS="-fdebug-prefix-map=/ebpf=." \
  --env HOME="/tmp" \
  "ghcr.io/cilium/ebpf-builder:1666886595" /bin/bash
```

Now in the container we generate the ELF and go files:

```
GOPACKAGE=main go run github.com/cilium/ebpf/cmd/bpf2go -cc clang-14 -cflags '-O2 -g -Wall -Werror' bpf helloworld.bpf.
c -- -I../headers
```

Let us examine the newly created files: `bpf_bpfel.go` / `bpf_bpfeb.go` contain the go code for the user state side of our app. The `bpf_bpfel.o` / `bpf_bpfeb.o` files are ELF files and can be examined using readelf:

```
readelf --section-details --headers bpf_bpfel.o
```

We see two things:

- that Machine reads “Linux BPF” and
- our tracepoint sys_enter_execve in the sections part (tracepoint/syscalls/sys_enter_execve).

> **Note**
> There are always two files created: bpf_bpfel.o for little endian systems (like x86) and bpfen.o for big endian systems.

Now we have everything in place to build our app:

```
go mod tidy
go build helloworld.go bpf_bpfel.go
exit #exit container
```

Let us cat tracepipe first in a second terminal (webshell: don’t forget to connect to the vm first):

```
sudo cat /sys/kernel/debug/tracing/trace_pipe
```

and in the first terminal execute our eBPF app:

```
sudo ./helloworld
```

Now we can see, that for each programm called in linux, our code is executed and writes “Hello world” to trace_pipe.

Close now apps by hitting Ctrl+c, you can also close the second terminal.

13. Cilium Enterprise

So far, we used the Cilium CNI in the Open Source Software (OSS) version. Cilium OSS has joined the CNCF as an incubating project and only recently during KubeCon 2022 NA applied to become a CNCF graduated project . Isovalent , the company behind Cilium also offers enterprise support for the Cilium CNI. In this lab, we are going to look at some of the enterprise features.

Task 13.1: Create a Kubernetes Cluster and install Cilium Enterprise

We are going to spin up a new Kubernetes cluster with the following command:

```
minikube start --network-plugin=cni --cni=false --kubernetes-version=1.24.3 -p cilium-enterprise
```

Now check that everything is up and running:

```
kubectl get node
```

This should produce a similar output:

```
NAME                STATUS   ROLES                  AGE   VERSION
cilium-enterprise   Ready    control-plane,master   86s   v1.24.3
```

Alright, everything is up and running and we can continue with the Cilium Enterprise Installation. First we need to add the Helm chart repository:

```
helm repo add isovalent https://....
```

> **Note**
> Your trainer will provide you with the Helm chart url.

Next, create a `cilium-enterprise-values.yaml` file with the following content:

```
cilium:
  hubble:
    enabled: false
    relay:
      enabled: false
  nodeinit:
    enabled: true
  ipam:
    mode: cluster-pool
hubble-enterprise:
  enabled: false
  enterprise:
    enabled: false
hubble-ui:
  enabled: false
```

And then install Cilium enterprise with Helm:

```
helm install cilium-enterprise isovalent/cilium-enterprise --version 1.12.7  \
  --namespace kube-system -f cilium-enterprise-values.yaml
```

To confirm that the cilium daemonset is running Cilium Enterprise, execute the following command and verify that the container registry for `cilium-agent` is set to `quay.io/isovalent/cilium` :

```
kubectl get ds -n kube-system cilium -o jsonpath='{.spec.template.spec.containers[0].image}' | cut -d: -f1
```

Run the following command and validate that cilium daemonset is up and running:

```
kubectl get ds -n kube-system cilium
```

This should give you an output similar to this:

```
NAME              DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
cilium            1         1         1       1            1           <none>          91s
```

13.1. Network Visibility with Hubble

Task 13.1.1: Enable Hubble, Hubble Relay, and Hubble UI

Edit your `cilium-enterprise-values.yaml` file so that it reads:

```
cilium:
  (...)
  extraConfig:
    # Disable Hubble flow export.
    export-file-path: ""
  hubble:
    enabled: true
    tls:
      enabled: true
    relay:
      enabled: true
  (...)
hubble-ui:
  enabled: true
```

Then, run helm upgrade command to apply the new configuration:

> **Note**
> Running the helm upgrade command below will restart Cilium daemonset.

```
helm upgrade cilium-enterprise isovalent/cilium-enterprise --version 1.12.3+1 \
  --namespace kube-system -f cilium-enterprise-values.yaml --wait
```

Task 13.1.2: Deploy a simple application

To accually see something with Hubble, we first deploy our `simple-app.yaml` from lab 03 again to run some experiments. Run the following command using the `simple-app.yaml` from lab 03:

```
kubectl apply -f simple-app.yaml
```

Now let us redo the task from lab 03.

Let’s make life again a bit easier by storing the Pod’s name into an environment variable so we can reuse it later again:

```
FRONTEND=$(kubectl get pods -l app=frontend -o jsonpath='{.items[0].metadata.name}')
echo ${FRONTEND}
NOT_FRONTEND=$(kubectl get pods -l app=not-frontend -o jsonpath='{.items[0].metadata.name}')
echo ${NOT_FRONTEND}
```

Then execute

```
kubectl exec -ti ${FRONTEND} -- curl -I --connect-timeout 5 backend:8080
```

and

```
kubectl exec -ti ${NOT_FRONTEND} -- curl -I --connect-timeout 5 backend:8080
```

You see that altought we have no `kube-proxy` running, the backend service can still be reached.

```
HTTP/1.1 200 OK
X-Powered-By: Express
Vary: Origin, Accept-Encoding
Access-Control-Allow-Credentials: true
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Last-Modified: Sat, 26 Oct 1985 08:15:00 GMT
ETag: W/"83d-7438674ba0"
Content-Type: text/html; charset=UTF-8
Content-Length: 2109
Date: Tue, 14 Dec 2021 10:01:16 GMT
Connection: keep-alive

HTTP/1.1 200 OK
X-Powered-By: Express
Vary: Origin, Accept-Encoding
Access-Control-Allow-Credentials: true
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Last-Modified: Sat, 26 Oct 1985 08:15:00 GMT
ETag: W/"83d-7438674ba0"
Content-Type: text/html; charset=UTF-8
Content-Length: 2109
Date: Tue, 14 Dec 2021 10:01:16 GMT
Connection: keep-alive
```

Task 13.1.3: Access Hubble UI

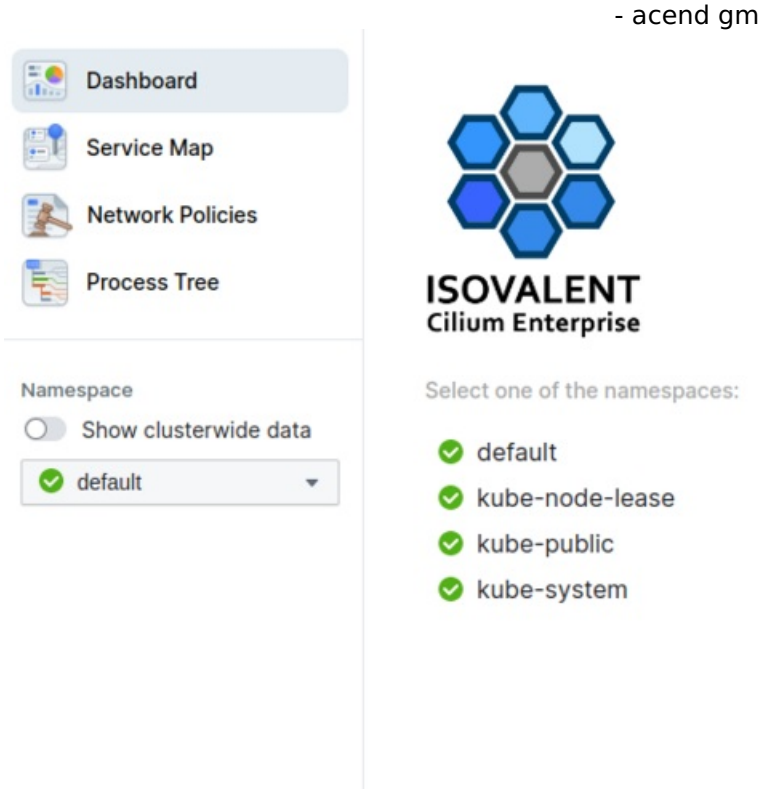
To access Hubble UI, forward a local port to the Hubble UI service:

```
kubectl port-forward -n kube-system svc/hubble-ui 12000:80 &
```

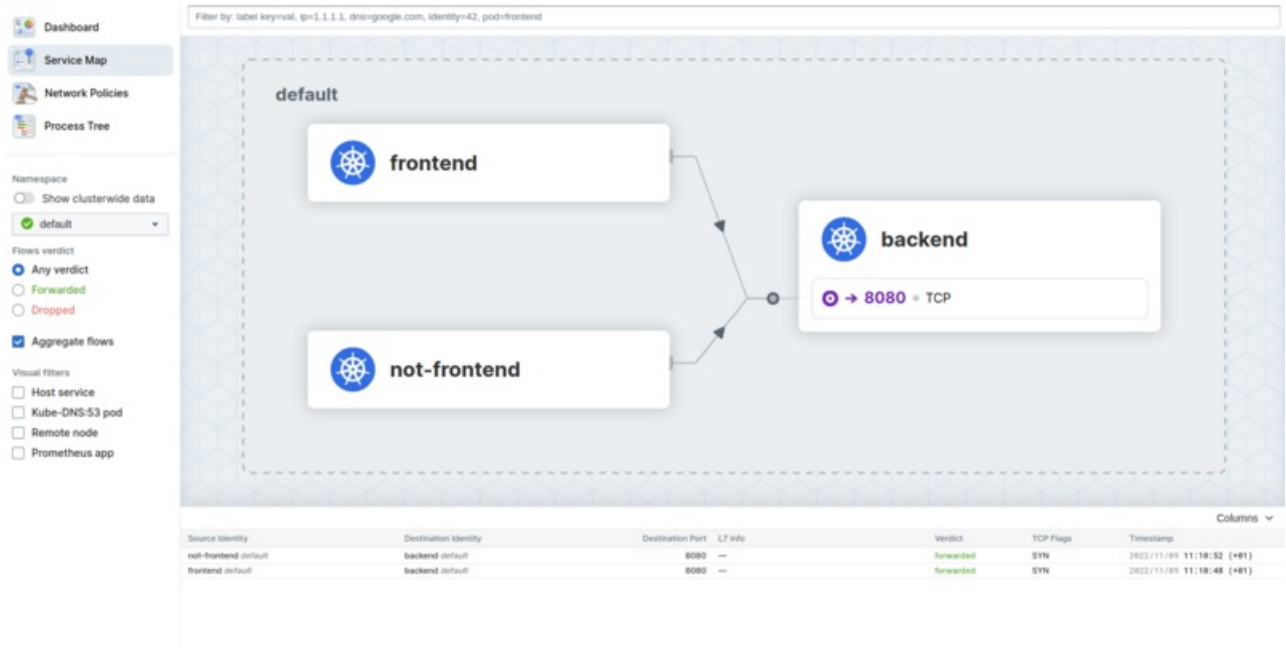
In our Webshell environment you can use the public IP of the VM to access Hubble. A simple way is to execute

```
echo "http://$(curl -s ifconfig.me):12000"
```

and copy the output in a new browser tab. If you are working locally, open your browser and go to http://localhost:12000/.



Select the `default` Namespace and go to Service Map:



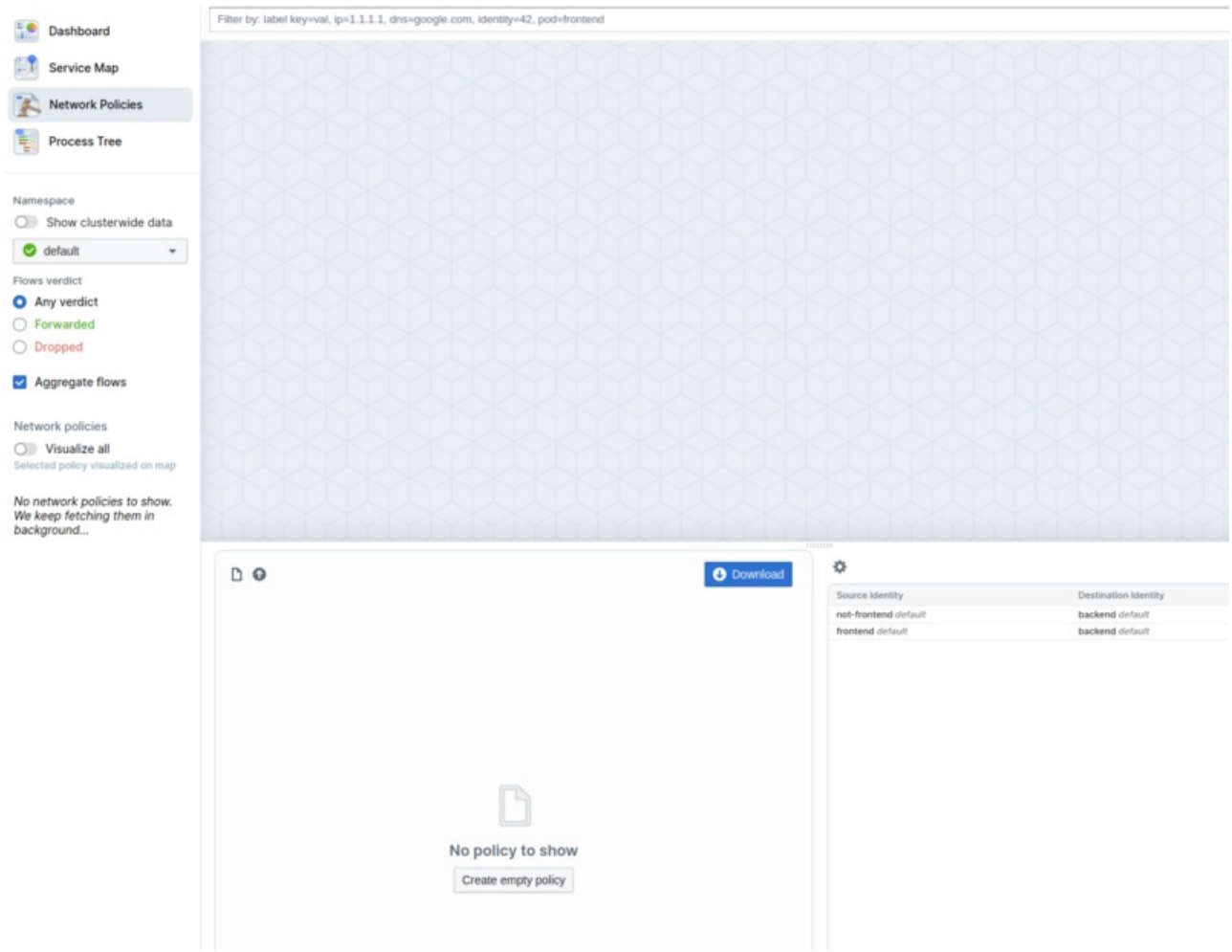
You should see our already deployed simple app with the `frontend`, `notfrondend` and `backend` Pod.

13.2. Network Policies

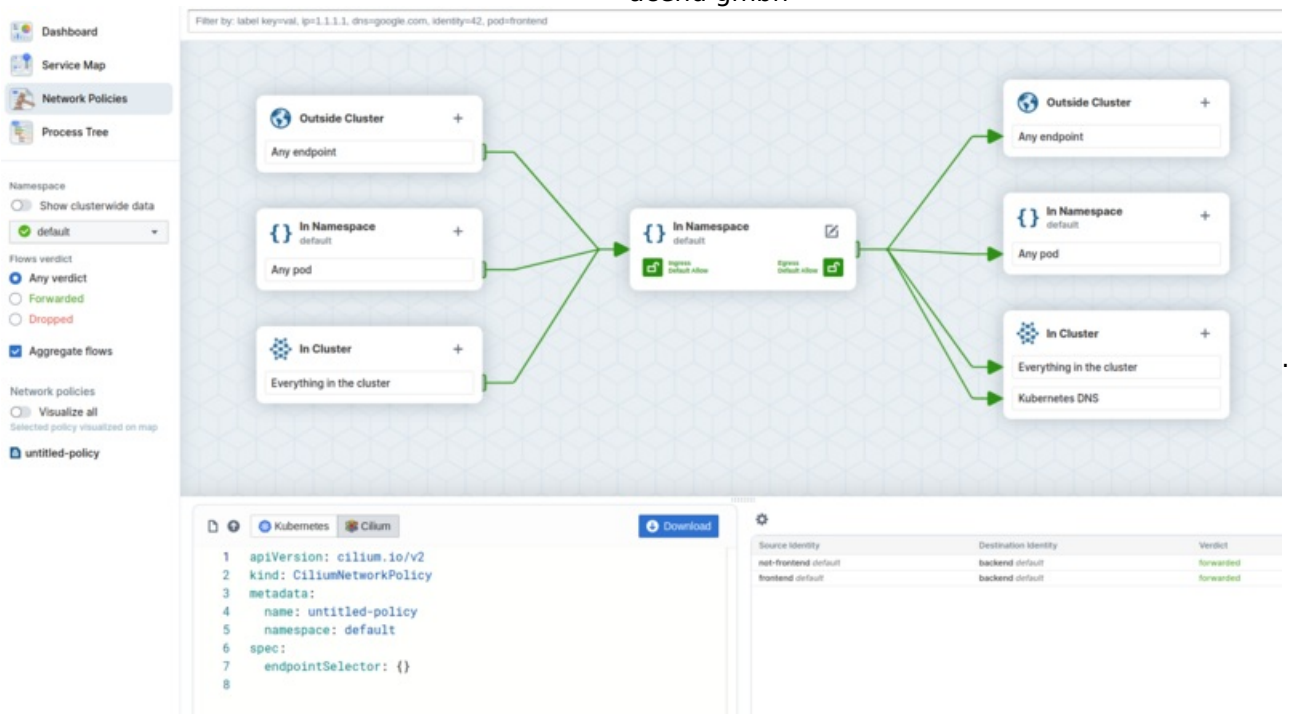
Task 13.2.1: Create a network policy with the Hubble UI

The Enterprise Hubble UI has an intergate Network Policy Editor similar to the one we already know from lab Cilium Network Policy. The Enterprise Network Policies Editor allows you to use knowlege of the current flows to easealy create new policies.

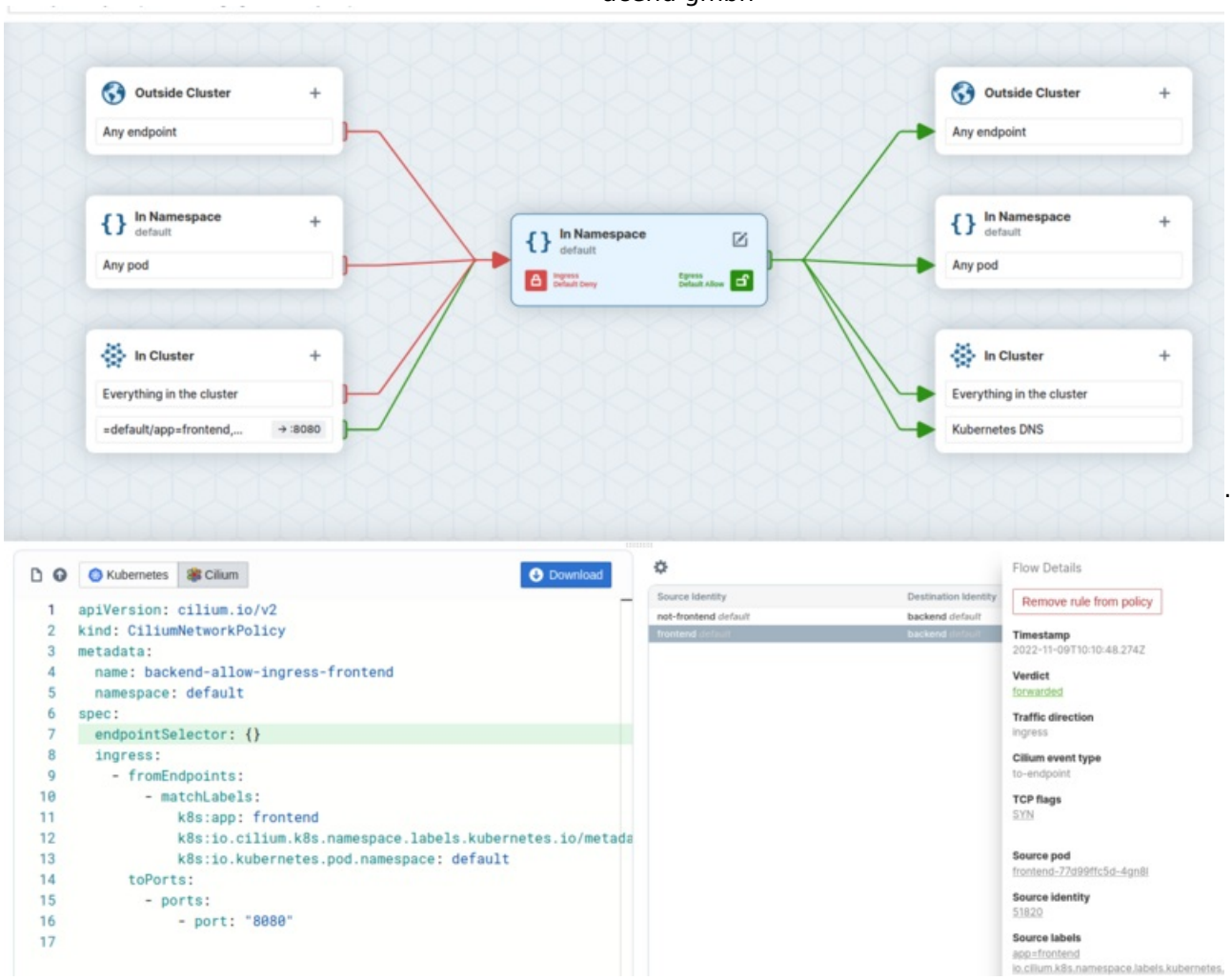
Go to `Network Policies`:



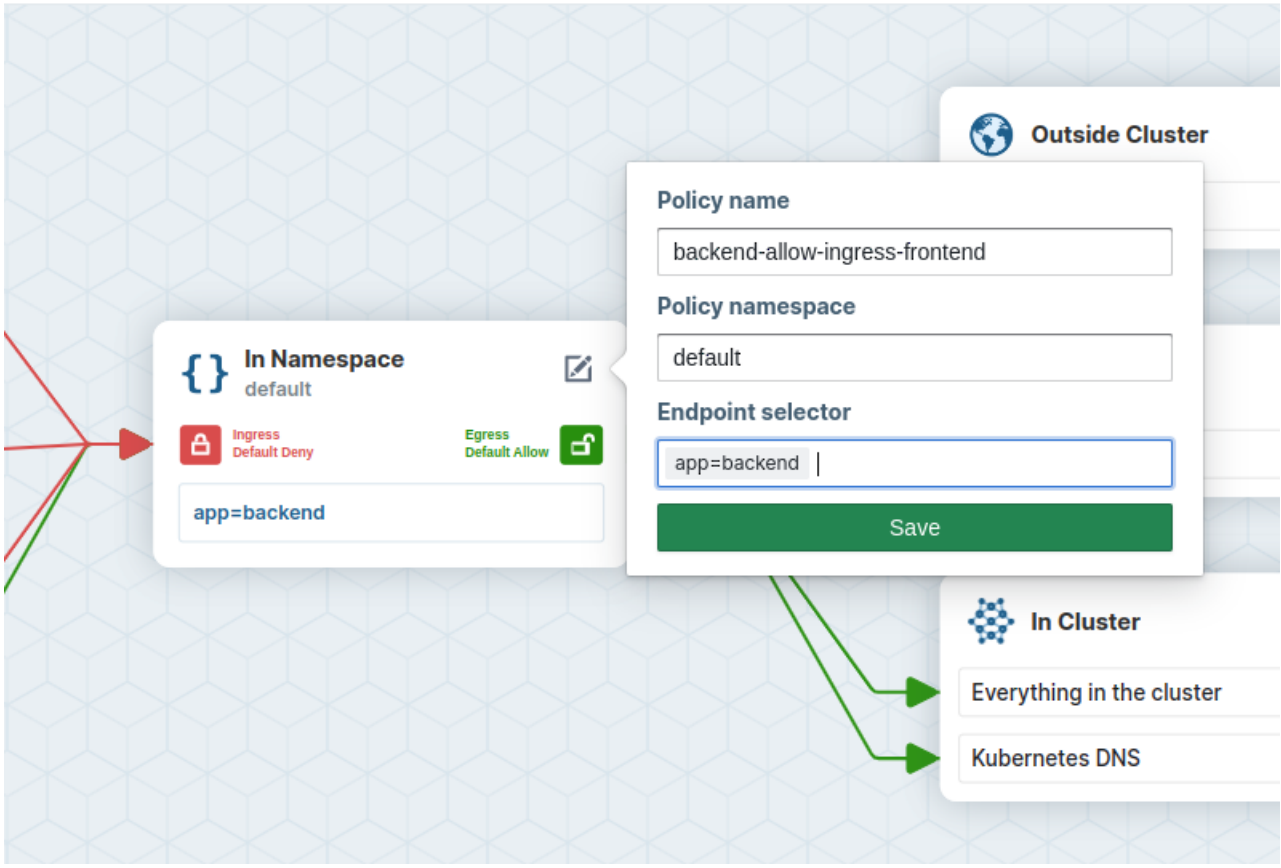
And the create a new empty policy:



We now want to allow traffic from the `frontend` pod to the `backend` pod while traffic from `not-frontend` to `backend` is blocked. In the right panel you the see existing flows. Select the flow from `frontend` to `backend` and then click on the `Add rule to policy` Button. The Network Policy Editor now visualice the policy.



Edit the Policy Name to `backend-allow-ingress-frontend` and also add `app=backend` as the endpoint selector:



Afterwards download the CiliumNetworkPolicy which should look like:

```
---
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: backend-allow-ingress-frontend
  namespace: default
spec:
  endpointSelector:
    matchLabels:
      app: backend
  ingress:
    - fromEndpoints:
        - matchLabels:
            k8s:app: frontend
            k8s:io.cilium.k8s.namespace.labels.kubernetes.io/metadata.name: default
            k8s:io.kubernetes.pod.namespace: default
      toPorts:
        - ports:
            - port: "8080"
```

Task 13.2.2: Apply Network Policy

Apply the file with:

```
kubectl apply -f backend-allow-ingress-frontend.yaml
```

and then execute the connectivity test again:

```
kubectl exec -ti ${FRONTEND} -- curl -I --connect-timeout 5 backend:8080
```

and

```
kubectl exec -ti ${NOT_FRONTEND} -- curl -I --connect-timeout 5 backend:8080
```

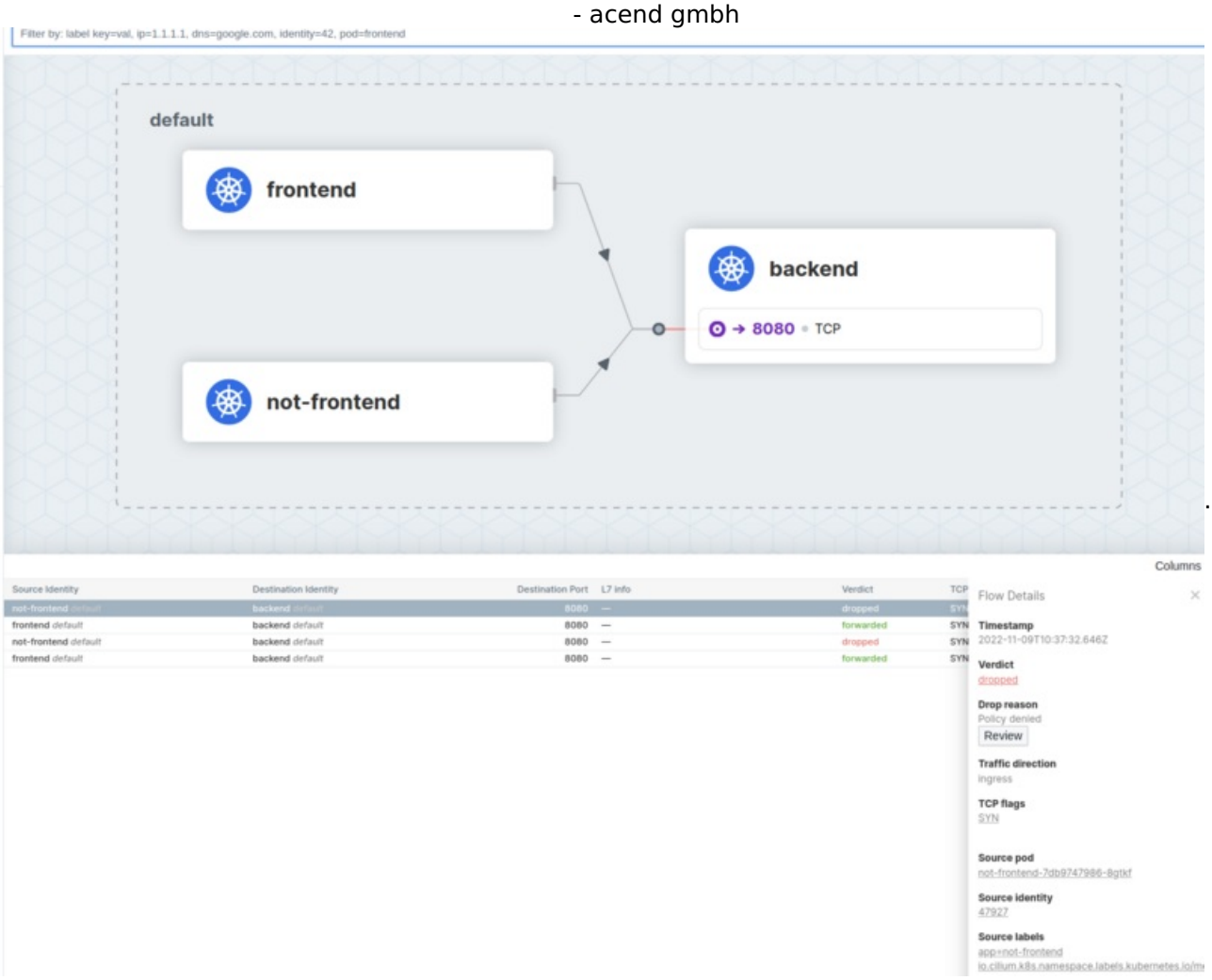
And you see the `frontend` application is able to connect to the `backend` but the `not-frontend` application cannot connect to the `backend` :

```
# Frontend
HTTP/1.1 200 OK
X-Powered-By: Express
Vary: Origin, Accept-Encoding
Access-Control-Allow-Credentials: true
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Last-Modified: Sat, 26 Oct 1985 08:15:00 GMT
ETag: W/"83d-7438674ba0"
Content-Type: text/html; charset=UTF-8
Content-Length: 2109
Date: Tue, 23 Nov 2021 13:08:27 GMT
Connection: keep-alive

# Not Frontend
curl: (28) Connection timed out after 5001 milliseconds
command terminated with exit code 28
```

Task 13.2.3: Observe the Network Flows

In the Hubble UI Service map you see now some `dropped` flows.



By clicking on the `Review` button, the enterprise Hubble UI allows you to see which Network Policy was the reason for the `dropped` verdict.

13.3. Process Visibility

Task 13.3.1: Enable Process Visibility

Edit your cilium-enterprise-values.yaml file so that it reads:

```
cilium:
  (...)
hubble-enterprise:
  enabled: true
  enterprise:
    enabled: true
  (...)
```

Then, run `helm upgrade` command to apply the new configuration:

```
helm upgrade cilium-enterprise isovalent/cilium-enterprise --version 1.12.7
  --namespace kube-system -f cilium-enterprise-values.yaml --wait
```

Task 13.3.2: Validate the Installation

First, please run:

```
kubectl get ds -n kube-system hubble-enterprise
```

and ensure that all the pods for `hubble-enterprise` daemonset are in `READY` state.

Run `hubble-enterprise` command to validate that Cilium Enterprise is configured with process visibility enabled:

```
kubectl exec -n kube-system ds/hubble-enterprise -c enterprise -- hubble-enterprise getevents
```

and you will see process events from one of the `hubble-enterprise` pods in JSON format.

Task 13.3.3: Export logs and visualize in Hubble UI Process Tree

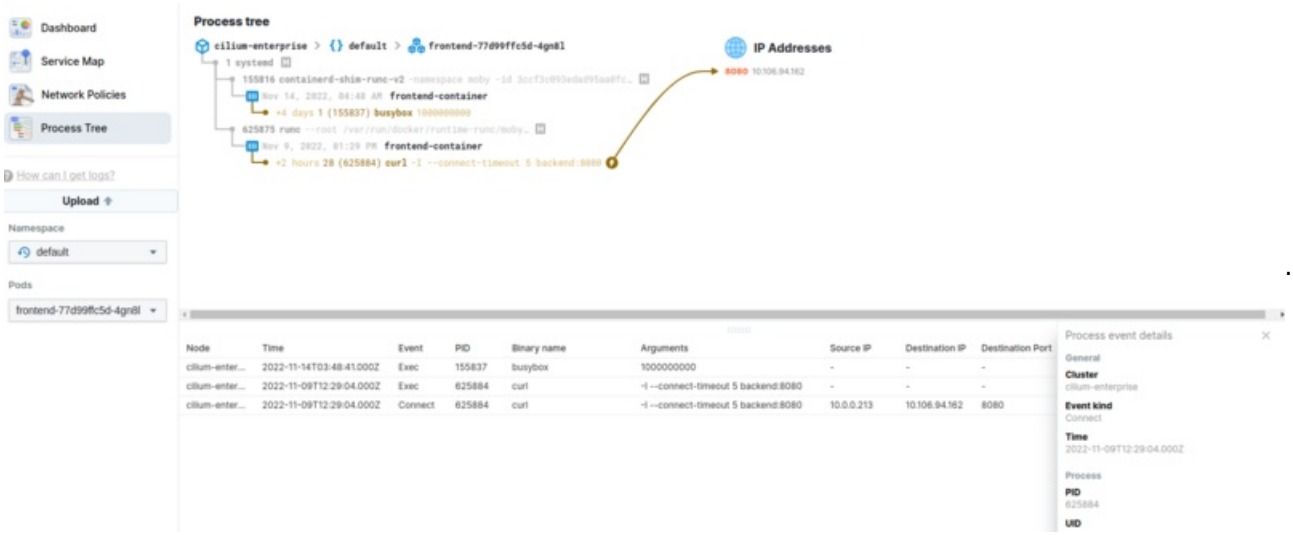
Execute the connectivity test from `frontend` to `backend` again to make sure we have some data to visualize:

```
kubectl exec -ti ${FRONTEND} -- curl -I --connect-timeout 5 backend:8080
kubectl exec -ti ${NOT_FRONTEND} -- curl -I --connect-timeout 5 backend:8080
```

Then, use the following command to export process events from hubbe-enterprise:

```
kubectl logs -n kube-system ds/hubble-enterprise -c export-stdout --since=1h > export.log
```

In the Hubble-UI open the Process Tree and click on the `Upload` Button. Upload the previously created `export.log`. Now you can select the `default` Namespace and one of the Pods, e.g. `frontend-xxxxx-xxx`.



We see our previously executed `curl` command and that the process opened a connection to an IP on Port 8080.

By clicking on one of the event, e.g. the `Connect` event for the `curl` command, you get some more details for the selected event.

13.4. Exporting Events

Task 13.4.1: Export Network Events

Edit your `cilium-enterprise-values.yaml` file and include export-file-path field to export network events:

```
cilium:
  (...)
  extraConfig:
    # Enable network event export
    export-file-path: "/var/run/cilium/hubble/hubble.log"
  (...)
hubble-enterprise:
  enabled: true
```

Then, run helm upgrade command to apply the new configuration:

```
helm upgrade cilium-enterprise isovalent/cilium-enterprise --version 1.12.7
  --namespace kube-system -f cilium-enterprise-values.yaml --wait
```

and restart cilium daemonset for the new filters to take effect:

```
kubectl rollout restart -n kube-system ds/cilium
```

Task 13.4.2: Export Process Events

Edit your `cilium-enterprise-values.yaml` file and include exportFilename field to export process events:

```
cilium:
  (...)
hubble-enterprise:
  enabled: true
  enterprise:
    # Enable process event export
    exportFilename: "fgs.log"
  (...)
```

Then, run helm upgrade command to apply the new configuration:

```
helm upgrade cilium-enterprise isovalent/cilium-enterprise --version 1.12.7
  --namespace kube-system -f cilium-enterprise-values.yaml --wait
```

Task 13.4.3: Observe Exported Events

Run the following command to observe exported events in `export-stdout` container logs:

```
kubectl logs -n kube-system -l app.kubernetes.io/name=hubble-enterprise -c export-stdout -f
```

Those exported events can now be sent to Splunk, Elasticsearch or similar.