ISOVALENT

🥈 Isovalent Enterprise for Cilium: Network Policies

A Hands-on lab by Isovalent

Version 1.2.1

⚠️ **DISCLAIMER**

This document serves as a reference guide summarizing the content and steps of the hands-on lab. It is intended to help users recall the concepts and practices they learned when taking the lab.

Please note that the scripts, tools, and environment configurations required to recreate the lab are not included in this document.

For a complete hands-on experience, please refer to the live lab environment instead.

You can access the lab at https://isovalent.com/labs/cilium-network-policies.

The World of Cilium Map at https://labs-map.isovalent.com also lets you access all labs and view your progress through the various learning paths.

Challenges

⎈ A cloud native challenge

Imagine you have to share resources between multiple tenants. How do you ensure that resources can only be accessed by the tenants they are meant for?

Following that thought, how would you secure each tenant individually? Limit outgoing traffic, secure against incoming traffic?

IP addresses and ports cannot help you here! So how do you do it?

👮 Network Policies

The answer is: with cloud native network policies. Policies which are not trying to use ephemeral IPs and random ports, but policies which are identity aware, which "understand" the cloud native environment natively.

🪪 Cloud Native Identity

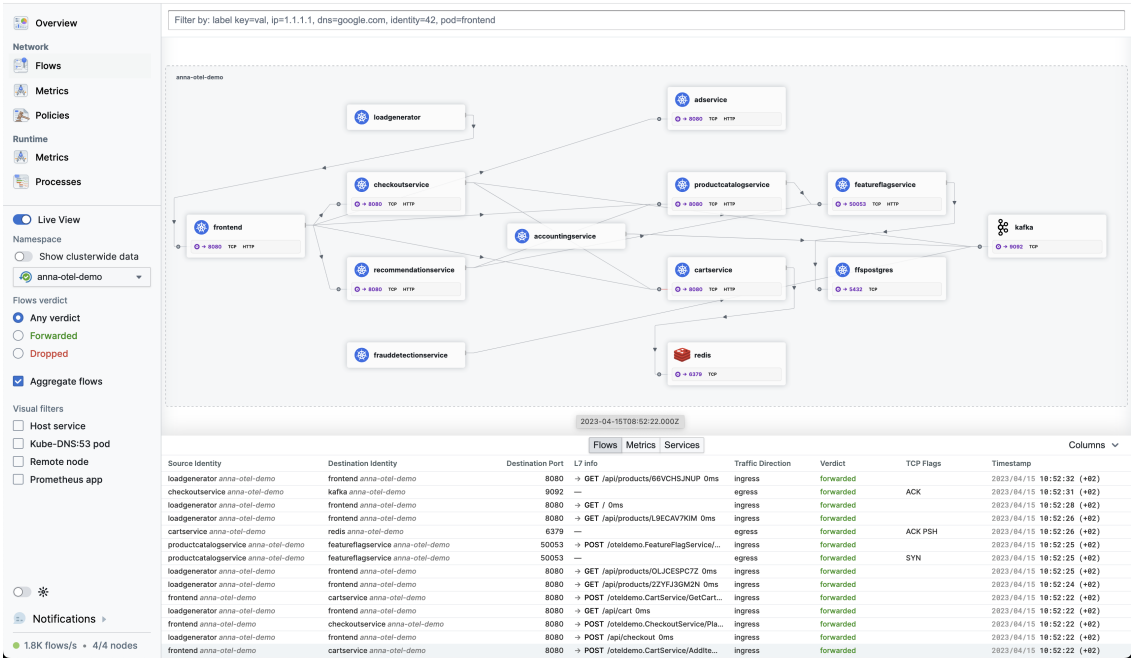
Cloud native and identity aware mean understanding

- tenant namespaces
- cluster names
- ingress and egress
- DNS names

🛠️ Visualize, create and modify Network Policies

And how can you create, modify and enforce such cloud native policies?

With the right toolset: Isovalent Enterprise for Cilium, the eBPF-powered CNI, and Isovalent Hubble Enterprise, our observability platform!



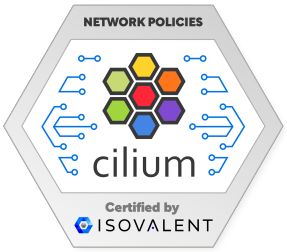
Hubble UI

To showcase how this all works together, in this demo we create a simple multi-tenant environment. Next we use Hubble UI's network policy editor and service map / flow details pages to create, troubleshoot, and update a complex network policy that includes label and DNS-aware network policies.

🏅 Get a Badge!

By completing this lab, you will be able to earn a badge.

Make sure to finish the lab in order to get your badge!



🏛️ The Lab Environment

🏛️ The Kind Cluster

Let's have a look at this lab's environment.

We are running a Kind Kubernetes cluster, and on top of that Cilium.

While the Kind cluster is finishing to start, let's have a look at its configuration:

```
yq /etc/kind/${KIND_CONFIG}.yaml
```

🖥️ Nodes

In the `nodes` section, you can see that the cluster consists of three nodes:

- 1 `control-plane` node running the Kubernetes control plane and etcd
- 2 `worker` nodes to deploy the applications

We are exposing ports on the control plane node so we can access them from the work machine:

- port 31234, used to access Hubble Relay (from the CLI)
- port 31235, used to access the Hubble UI

Note that since the standard port for Hubble Relay is 4245, we have exported the `HUBBLE_SERVER` variable in the current shell to use the 31234 port instead. Verify its value with the following command:

```
echo $HUBBLE_SERVER
```

🔀 Networking

In the `networking` section of the configuration file, the default CNI has been disabled so the cluster won't have any Pod network when it starts. Instead, Cilium is being deployed to the cluster to provide this functionality.

To see if the Kind cluster is ready, verify that the cluster is properly running by listing its nodes:

```
kubectl get nodes
```

You should see the three nodes appear. If not, the worker nodes might still be joining the cluster.

Now that we have a working Kind cluster, let's move on to deploy a sample application on it!

🚀 A Demo Application

In the following challenge, we will deploy a demo application for the lab.

🚀 Deploying a demo app

🚀 Deploy the demo

Let's deploy a simple demo application to explore the Isovalent Enterprise for Cilium network security capabilities. We will create three namespaces, and on top of them deploy three services:

```
kubectl create ns tenant-a
kubectl create ns tenant-b
kubectl create ns tenant-c
kubectl create -f https://docs.isovalent.com/public/tenant-services.yaml -n tenant-a
kubectl create -f https://docs.isovalent.com/public/tenant-services.yaml -n tenant-b
kubectl create -f https://docs.isovalent.com/public/tenant-services.yaml -n tenant-c
```

⬢ Cilium and Hubble status

While the application is starting, let's check if all Cilium components have been properly deployed. Note that it might take a few seconds to display the results!

```
cilium status --wait
```

If all is well, the 3 first lines should indicate OK. Some services might not be available yet. You can wait a bit and try again.

You can also verify that you can properly connect to Hubble relay (using port 31234 in our lab) with:

```
hubble status
```

and that all nodes are properly managed in Hubble:

```
hubble list nodes
```

✅ Verify the application deployment

Before we continue, let's check if all pods have been deployed:

```
kubectl get pods --all-namespaces | grep "tenant"
```

If the services are not all up and running, just wait a few seconds and execute the command again.

When all is finally up we can head over to the next step!

😶 Default Rules

With the demo application in place now, how does the traffic situation look like? What rules are in place?

🚦 All allowed by default

There is no Network Policy applied in the `tenant-a` namespace, so all network traffic will be allowed based on the specification of Kubernetes.

🧪 Testing default policy

Let's check out how this looks like, and just execute a few commands to verify that the default "allow all" policy is in place.

👮 What policies are in place by default?

🌐 Generate HTTP requests

From within `tenant-a` we can connect to various services with the help iof `curl`.

First, let's see if the `frontend-service` pod has access to the `backend-service` service in the same (`tenant-a`) namespace:

```
kubectl exec -n tenant-a frontend-service -- \
  curl -sI backend-service.tenant-a
```

We are getting a `HTTP/1.1 200 OK` response, indicating that traffic is flowing without restriction.

Now let's test traffic to the `backend-service` service in `tenant-b` of the cluster:

```
kubectl exec -n tenant-a frontend-service -- \
  curl -sI backend-service.tenant-b
```

Again, traffic is allowed. Finally, check access to a service outside of the cluster, such as `api.twitter.com`:

```
kubectl exec -n tenant-a frontend-service -- \
  curl -sI api.twitter.com
```

This one returns a `301` response, which also shows that traffic is flowing.

We can see that, by default, all traffic from a pod in the namespace of `tenant-a` is allowed:

- within the tenant-a namespace
- to services in other namespaces (e.g. `tenant-b`)
- to external endpoints outside the Kubernetes cluster (e.g. `api.twitter.com`)

👓 Observing Flows

The `hubble` CLI connects to the Hubble Relay component in the cluster and retrieves logs called "Flows". This command line tool then allows you visualize and filter the flows.

Visualize the TCP traffic sent by the `frontend-service` pod in the `tenant-a` namespace with:

```
hubble observe --from-pod tenant-a/frontend-service --
protocol tcp
```

You should see a list of logs, each with:

- a timestamp
- a source pod, along with its namespace, port, and Cilium identity
- the direction of the flow (->, <-, or at times <> if the direction could not be determined)
- a destination pod, along with its namespace, port, and Cilium identity
- a trace observation point (e.g. `to-endpoint`, `to-stack`, `to-overlay`)
- a verdict (e.g. `FORWARDED` or `DROPPED`)
- a protocol (e.g. `UDP`, `TCP`), optionally with flags

Identify the three requests (to `backend-service.tenant-a`, `api.twitter.com`, and `backend-service.tenant-b`) in the flows.

> 💡 **NOTE**
>
> The TCP replies are missing from these flows as we're only displaying traffic coming from the pod.
>
> Using `--pod` instead of `--from-pod` would display both outgoing and incoming traffic for this pod.

These flows confirm that all three requests were forwarded to their destinations, as all flows are marked as `FORWARDED`.

In the next challenge, we'll see how we can restrict this!

❓ Default Policies Quiz

When no Network Policies are in place in a namespace (select all answers that apply)

- All traffic is allowed from the namespace's pod to other pods in the same namespace
- All traffic is allowed from the namespace's pod to pods in other namespaces
- All traffic is allowed from the namespace's pod to external addresses
- All traffic is allowed from pods in other namespaces to the namespace's pods
- All traffic is allowed from external addresses to the namespace's pods

👮 Adding Policies

We just saw that the default "allow all" policy is in place. To secure the `tenant-a` namespace we need to bring in some policies. One way would be to just write them down in a `yaml` file as we think they fit.

The disadvantage is that this quickly gets overwhelming, hard to overview and details can easily get lost. A better way is a structured visual representation which clearly and quickly shows what policies are in place, what rules are set and so on.

🚓 The Hubble UI

For this we are going to use the Hubble UI.

Hubble is a fully distributed networking and security observability platform. Conveniently, it also contains the network policy editor which allows us to visualize and edit network policies.

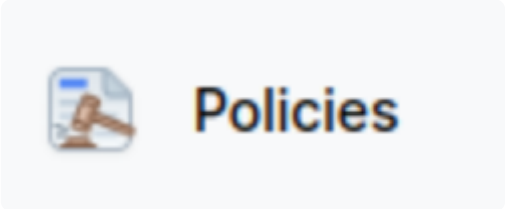
✍️ Edit your policy

Let's use the network policy editor in Hubble UI to modify the policy the way we need it!

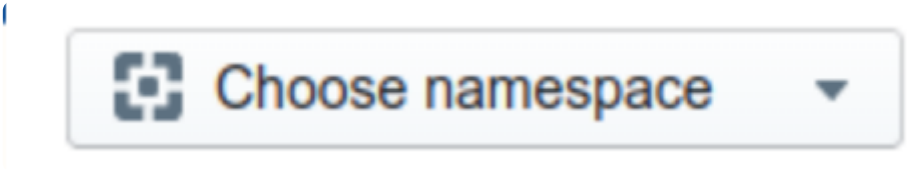
🛠️ Create a policy in the Hubble UI

👁️ The Network Policy Visualizer

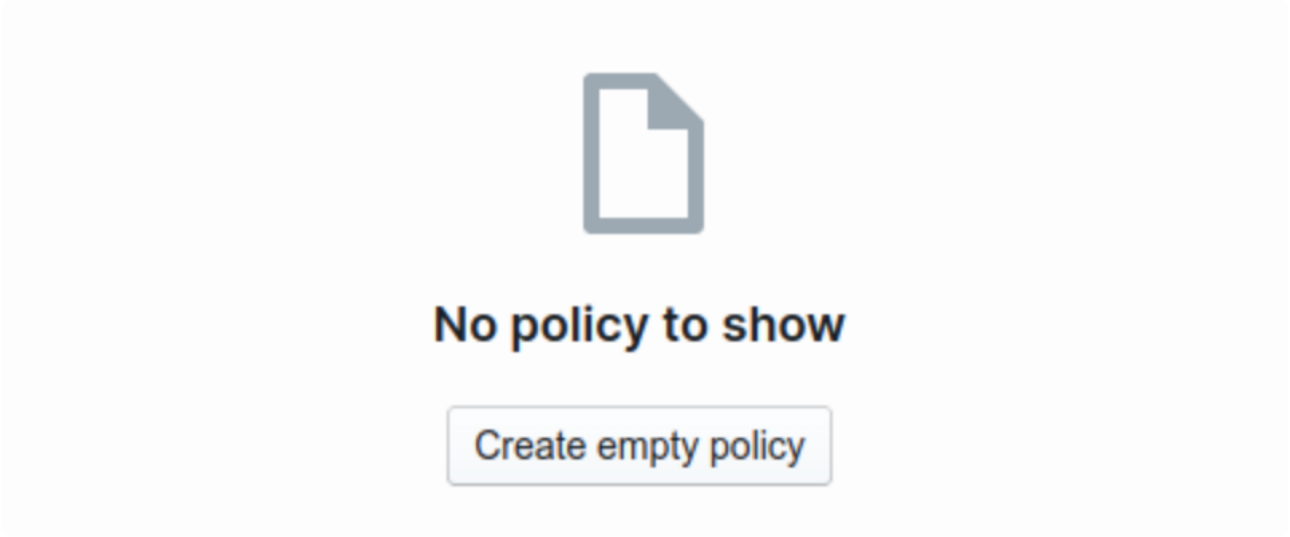
Let's now have a look at the Hubble UI (it might take a few seconds to load). Click on the **Policies** menu entry on the left.



On the left side of the menu there is now a drop-down menu to select a namespace.



Pick the `tenant-a` namespace. Since there is no network policies in this namespace yet, the main pane is empty, and the policy editor has a note saying **"No policy to show"**.



In the lower right hand corner, you can see a list of flows, all marked `forwarded`, corresponding to the traffic Hubble knows about this namespace.

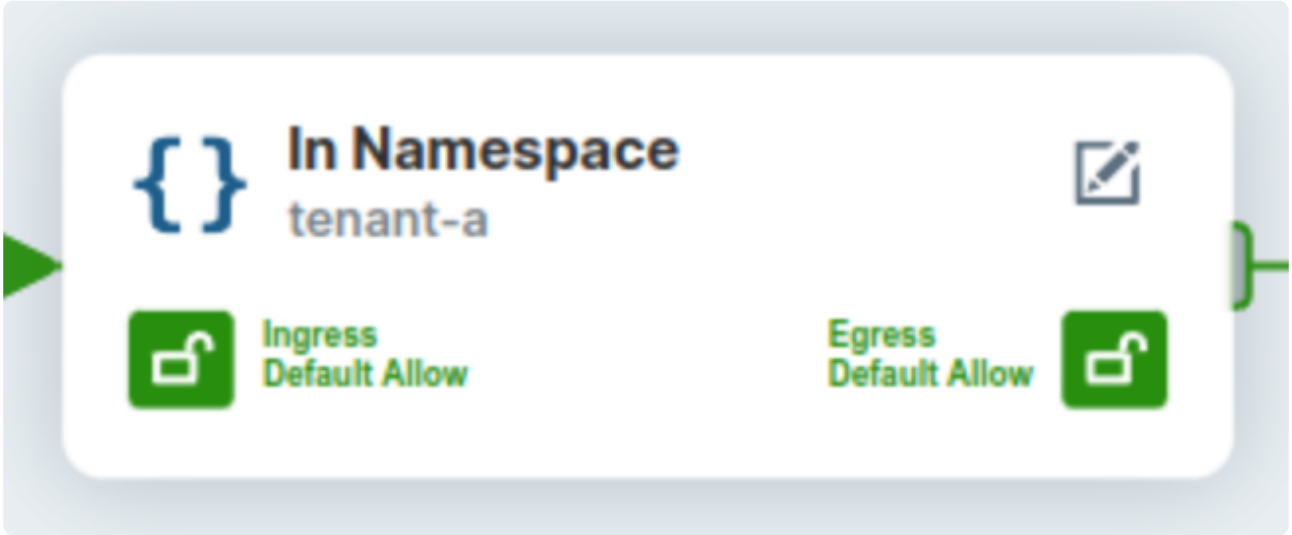
Since everything is currently allowed in this namespace, let's create a policy!

👮 Create a Network Policy

Click the **"Create empty policy"** button.

You will see a new policy being displayed in the main pane, along with its YAML representation in the editor pane in the lower hand corner.

The center box in the visualizer corresponds to the target pods for the policy. All arrows connected to this box are currently green, because the policy allows all traffic at the moment.



Click the 📝 button in the upper right hand corner of the center box and specify the following values:

- Policy name: `default`
- Policy namespace: `tenant-a`
- Endpoint selector: (leave empty - an empty pod selector matches all pods in the namespace)

Click the green **Save** button underneath it.

This updates the YAML document in the editor pane, which should now read:

```
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: default
  namespace: tenant-a
spec:
  endpointSelector: {}
```

If it doesn't, make sure you have selected the ⬢ `Cilium` tab of the editor, not the ⎈ `Kubernetes` one.

> 💡 **NOTE**
>
> The network policy editor generates YAML in both the Kubernetes Network Policy and Cilium Network Policy formats. In this lab however, you MUST select the Cilium Network Policy to download in order to ensure Cilium's DNS-aware network visibility and network policy can be enabled.

🛡️ Adding default deny rules

In the center block, click on the 🔓 **Ingress Default Allow** and **Egress Default Allow** 🔓 buttons, respectively in the lower left and lower right hand corners. This will update the policy to have rules that will drop all inbound and outbound connectivity from any pod in the `tenant-a` namespace.

All arrows in the visualizer have now turned red, and the YAML spec should now be:

```
spec:
  endpointSelector: {}
  ingress:
    - {}
  egress:
    - {}
```

You can see that adding empty rules will enforce a deny policy by default.

🚪 Allowing traffic

Now that we have a default deny, we can start allowing specific traffic in the policy.

We want to allow the following communication patterns:

- Ingress from workloads in the same namespace.
- Egress to workloads in the same namespace.
- Egress from workloads in the namespace to KubeDNS/CoreDNS so pods in the namespace can perform DNS requests.

To do that, on the left side (i.e. Ingress) of the visualizer, find the second box, titled **{} In Namespace** and click on the Any `pod` text. In the pop-up window, click **Allow from any pod**. This adds a green arrow from the **{} In Namespace** box to the center box, as well as a new Ingress rule to the YAML policy manifest:

```
ingress:
  - fromEndpoints:
      - {}
```

Repeat this step on the right side for the Egress **In Namespace box**.

Then on the right side (i.e. Egress), in the **In Cluster** box, click the `Kubernetes DNS` section, and in the pop-up window click the **Allow rule** button. Hover on the same `Kubernetes DNS` section again and toggle the `DNS proxy` option. This adds a whole block to the YAML manifest, which allows DNS (UDP/53) traffic to the `kube-system/kube-dns` pods.

> 💡 **NOTE**
>
> This DNS rule actually proxies DNS queries through a DNS proxy provided with Cilium, and will also enable DNS visibility in Hubble.
>
> For this reason, you will now be able to see `api.twitter.com` in the following challenges instead of the corresponding IP.

You should now have three green arrows in the visualizer, and the YAML manifest should look like this:

```
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: default
  namespace: tenant-a
spec:
  endpointSelector: {}
  ingress:
    - fromEndpoints:
        - {}
  egress:
    - toEndpoints:
        - {}
    - toEndpoints:
        - matchLabels:
            io.kubernetes.pod.namespace: kube-system
            k8s-app: kube-dns
      toPorts:
        - ports:
            - port: "53"
              protocol: UDP
          rules:
            dns:
              - matchPattern: "*"
```

📄 Save the Policy to file

Now we want to save the policy to our cluster. In the editor pane, select all the YAML code and copy it.

At the top of your browser window, select the **</> Editor** tab then:

1. select the file called `tenant-a-default-policy.yaml` in the left column
2. paste the content you copied from the Network Policies editor into the file

Now that we have generated our policy, let's enforce it in the next challenge!

❓ Base Policy Quiz

Select all answers that apply

- [x] Cilium supports standard Kubernetes Network Policies
- [ ] The Hubble UI only allows you create Cilium Network Policies
- [x] Adding an empty ingress rule blocks incoming traffic
- [ ] Adding an empty egress rule blocks incoming traffic
- [x] Cilium Network Policies allow to filter DNS requests to Kube DNS

🧪 Testing the policy

Let us now apply the Network Policy we have just created.

And what do we do once we applied the new policy? We test it, of course!

For this we can just re-execute the same commands we used previously to check if there are any policies in place at all.

🛡️ Enforce and test new policy

⎈ Apply the policy

Let's apply the policy:

```
kubectl apply -f tenant-a-default-policy.yaml
```

Did it work? The best way to find out is to replay the connection tests we used earlier!

✅ Test forwarded policies

Let's test the connectivity between the frontend-service and backend-service pod in the `tenant-a` namespace:

```
kubectl exec -n tenant-a frontend-service -- \
  curl -sI backend-service.tenant-a
```

We can see that the command succeeds as we get an HTTP reply, demonstrating that the communication within the `tenant-a` namespace and to KubeDNS is working.

We can visualize this traffic using `hubble`:

```
hubble observe --from-pod tenant-a/frontend-service
```

> 💡 **NOTE**
>
> Since we didn't use the `--protocol tcp` for this command, you are also seeing all the DNS requests.
>
> The details of the DNS query are also displayed because DNS requests are now proxied and analyzed by the Cilium DNS Proxy.

❌ Test denied policies

Now let's test denied policies.

Test connections to `api.twitter.com`:

```
kubectl exec -n tenant-a frontend-service -- \
  curl -sI --max-time 5 api.twitter.com
```

The connection now hangs (since it is blocked at L3/L4), and you will get a timeout after 5 attempts.

Similarly, let's test internal cluster services with:

```
kubectl exec -n tenant-a frontend-service -- \
  curl -sI --max-time 5 backend-service.tenant-b
```

Again, the connection hangs and times out.

This confirms that policies are properly denied to both external services, as well as other Kubernetes namespaces.

In the next challenge, we'll see how we can visualize these drops using both the Hubble CLI and UI!

🔍 Visualizing dropped traffic

We have successfully applied new rules and verified that connections are now not allowed.

But that means traffic was dropped - and as people in charge we surely want to see when traffic is dropped, right?

🛰️ Hubble to the rescue

Both the Hubble CLI and UI offer insight into connection drops.

In this short challenge we will guide you through the visualization, as a preparation to the following challenge where we are going to act on it.

🔭 Let's watch those connectivity drops

👓 Observing Flows

Using the `hubble` CLI as before, we can see all the requests in the `tenant-a` namespace:

```
hubble observe --namespace tenant-a
```

You will see flows marked as either `FORWARDED` or `DROPPED`.

You can filter on this criteria with the `--verdict` flag, for example executing:

```
hubble observe --namespace tenant-a --verdict DROPPED
```

You should be able to view the requests that were dropped in the previous challenge.

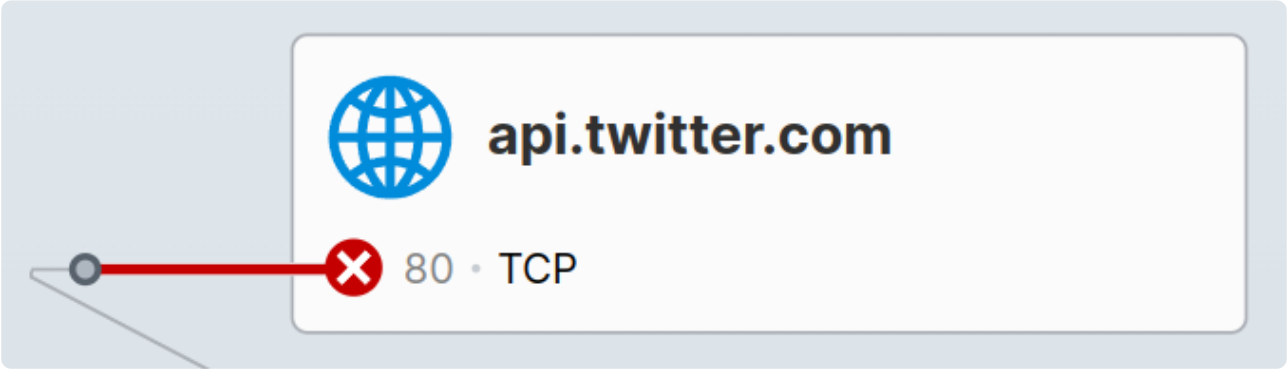
> 💡 **NOTE**
>
> Since DNS visibility is now enabled, you will actually see `api.twitter.com` in the logs.

🔭 The Hubble UI

Now head to the 🔗 🛰️ **Hubble UI** tab, click on **Connections** and select the `tenant-a` namespace.

This will show you how Hubble UI simplifies the understanding of service connectivity, and shows what connectivity has failed because of drops due to a Network Policy.

In the service map, the red lines at the end of the arrows indicate a dropped flow, while gray indicates a successful flow.



The flows table at the bottom of the pane also shows a simplified view of the connectivity for this namespace, including when the flow was last seen.

Have a look around, then head to the next challenge were we will use this interface to add the missing traffic policies!

❓ Hubble Observability Quiz

When using Hubble (select all answers that apply)

- The Hubble CLI allows to observe all Kubernetes traffic
- Hubble (CLI & UI) always display external DNS names
- The Hubble service map displays connection drops
- The Hubble CLI output can be filtered by pod

🛠️ Acting on the dropped traffic

Now that we see that the interface shows us what traffic is dropped, let's use that information!

Imagine we were just told that something is not running right in the application. We look at the network traffic and see that traffic is dropped which should not be dropped.

✅ Identify and allow traffic

We will allow traffic from the frontend to the backend, as well as to the backend of tenant-b, and to the Twitter API!

Afterwards we will of course test if everything is working as it should!

🆕 Update the network policy based on Hubble flows

👓 Visualize current policy

In the Hubble UI, go to the Policies view and pick the `tenant-a` namespace.

In the lower right-hand corner we see that Hubble has identified the set of flows observed in the `tenant-a` namespace that are not allowed by the current policy, and marked them as `dropped`.

🆕 Create a new policy

In order to allow additional traffic, we could add rules to the existing Network Policy.

Instead, let's create a new one so the scopes can be clearly separated.

Click the + New button in the top left hand corner of the editor pane.

Then click the 📝 icon in the center box and rename the policy to `extra`.

Look at the flows table in the bottom right hand pane. Two of them have a `dropped` verdict, the requests to `backend-service` in `tenant-b` and to `api.twitter.com`.

Click on the line corresponding to the `backend-service` in `tenant-b` and select **Add rule to policy**. The YAML manifest is now updated to accept this traffic!

Repeat the operation to allow traffic to `api.twitter.com`.

This results in a fine-grained network policy that allows the required connectivity, while retaining the default deny aspects of a zero-trust network policy.

The changes are also reflected in the policy visualization. For example check the lower box on the right side called **In Cluster** where now underneath the DNS rule another rule appeared.

👀 Viewing all policies

Since we are working on a new Network Policy, the main pane only shows the rules for this specific policy.

At the bottom of the left column, you can see a list of the policies for this namespace, with `extra` selected in a bold font, allowing you to switch between policies.

Toggle the **Visualize all** button immediately on top of the list. The main pane now displays the result of all policies applied simultanously.

In the policy code overview, click on ⬢ **Cilium** and copy the entire code.

📄 Save & Enforce Policy

Hop over to the **</> Editor** tab to the left, select the `tenant-a-extra-policy.yaml` file and paste the code.

Then go to the **>_ Terminal** tab and apply the new rule:

```
kubectl apply -f tenant-a-extra-policy.yaml
```

✅ Test the Policy

Let's verify that our policy works properly. Execute the same curl commands we run earlier:

Tenant-internal test:

```
kubectl exec -n tenant-a frontend-service -- \
  curl -sI --max-time 5 backend-service.tenant-a
```

External service test:

```
kubectl exec -n tenant-a frontend-service -- \
  curl -sI --max-time 5 api.twitter.com
```

Other tenant services test:

```
kubectl exec -n tenant-a frontend-service -- \
  curl -sI --max-time 5 backend-service.tenant-b
```

As you can see, while we still maintain an overall deny policy, we now also have carefully adjusted rules to allow just the traffic we want.

❌ Test denied rules

Let's check that other external destinations are still denied:

Another external service:

```
kubectl exec -n tenant-a frontend-service -- \
  curl -sI --max-time 5 www.google.com
```

Another internal service:

```
kubectl exec -n tenant-a frontend-service -- \
  curl -sI --max-time 5 backend-service.tenant-c
```

As you will see, those are still not reachable, and we can check the flows with:

```
hubble observe --namespace tenant-a
```

❓ Extra Rules Quiz

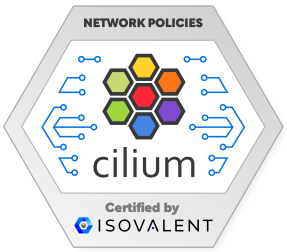
Adding Network Policies rules (select all answers that apply)

- Rules can be added to an existing Network Policy
- Rules can be added by creating a new Network Policy
- The Hubble Network Policy editor allows to edit existing Kubernetes Network Policies
- Modifying Network Policies in Hubble automatically applies them to the cluster
- Hubble cannot let you view all policies applying to namespace at the same time

🏆 Final Exam Challenge

This last challenge is an exam that will allow you to win a badge.

After you complete the exam, you will receive an email from Credly with a link to accept your badge. Make sure to finish the lab!



🥋 Exam Challenge

📝 Exam Instructions

For this practical exam, you will need to:

1. create a policy called `default-exam` in namespace `tenant-b` (use the `default-exam.yaml` file)
2. allow traffic to `google.com` on port 443 from all pods in namespace `tenant-b`
3. allow Kubernetes DNS traffic in `tenant-b`
4. allow traffic to pod backend-service in namespace `tenant-c` on port `80`
5. apply the policy

Notes:

- You have access to both the Hubble UI and the graphic Editor
- The terminal has auto-completion for most commands, so <Tab> is your friend!
  - The `tenant-c` rule is quite complex. To find the right rule, it helps to first attempt a connection with a deny-all rule, and then add the dropped flow as traffic.
  - A command that you might find helpful is: `kubectl exec -n tenant-b frontend-service -- curl -sI --max-time 5 backend-service.tenant-c`
- The Cilium Network Policy documentation on https://docs.cilium.io/en/stable/security/policy/

Good luck!

🎓 You've done it!

On completion of this lab, you will receive a badge. Feel free to share your achievement on social media!

We hope that this lab gave you a first insight into what Cilium and Hubble can bring to your Kubernetes cluster security with Network Policies.

Don't forget to rate this lab and head over to our home page if you want to learn more!