ISOVALENT

🥉 Getting Started with Cilium

A Hands-on lab by Isovalent

Version 1.1.1

⚠ **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-getting-started.

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

🏛️ The Lab Environment

⬢ Install Cilium

❓ Cilium Install Quiz

🚀 Deploying a demo app

🔍 Check Current Access

👮 A first L3/L4 policy

🌐 Apply and Test HTTP-aware L7 Policy

❓ Network Policies Quiz

🥋 Exam Challenge

🧪 Isovalent labs

You can access the whole list of Isovalent labs at https://isovalent.com/labs.

The labs can also be explored in an interactive way using the interactive map at https://labs-map.isovalent.com.

👮 Ensuring Security on Kubernetes

In a Cloud Native/Kubernetes environment, how do you

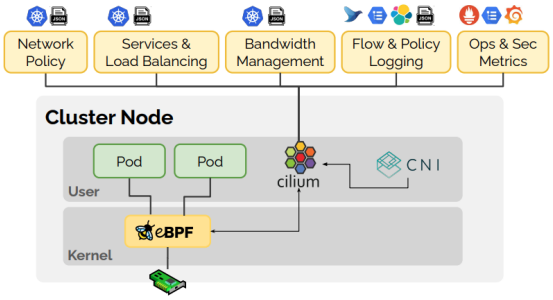
- enforce policies?
- troubleshoot the network?
- stay secure with minimal effort?

After all, in the highly dynamic and complex world of microservices, IP addresses and ports are no longer relevant.

A new approach is needed — meet Cilium!

Cilium Overview

Cilium provides Connectivity, Observability and Security capabilities in a Cloud Native World, and is based on eBPF.



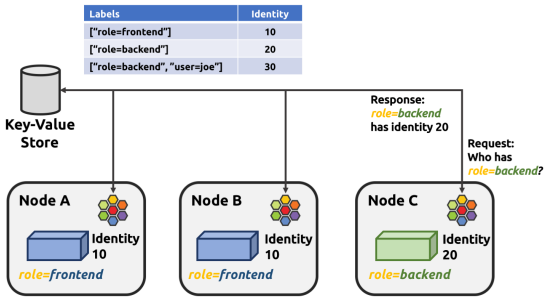
Thanks to eBPF, a revolutionary new kernel extensibility mechanism of Linux, we have the opportunity to rethink the Linux networking and security stack for the age of microservices.

Identities, Protocol parsing & Observability

From inception, Cilium was designed for large-scale, highly-dynamic containerized environments.

Cilium:

- natively understands container identities
- parses API protocols like HTTP, gRPC, and Kafka
- and provides visibility and security that is both simpler and more powerful than traditional approaches

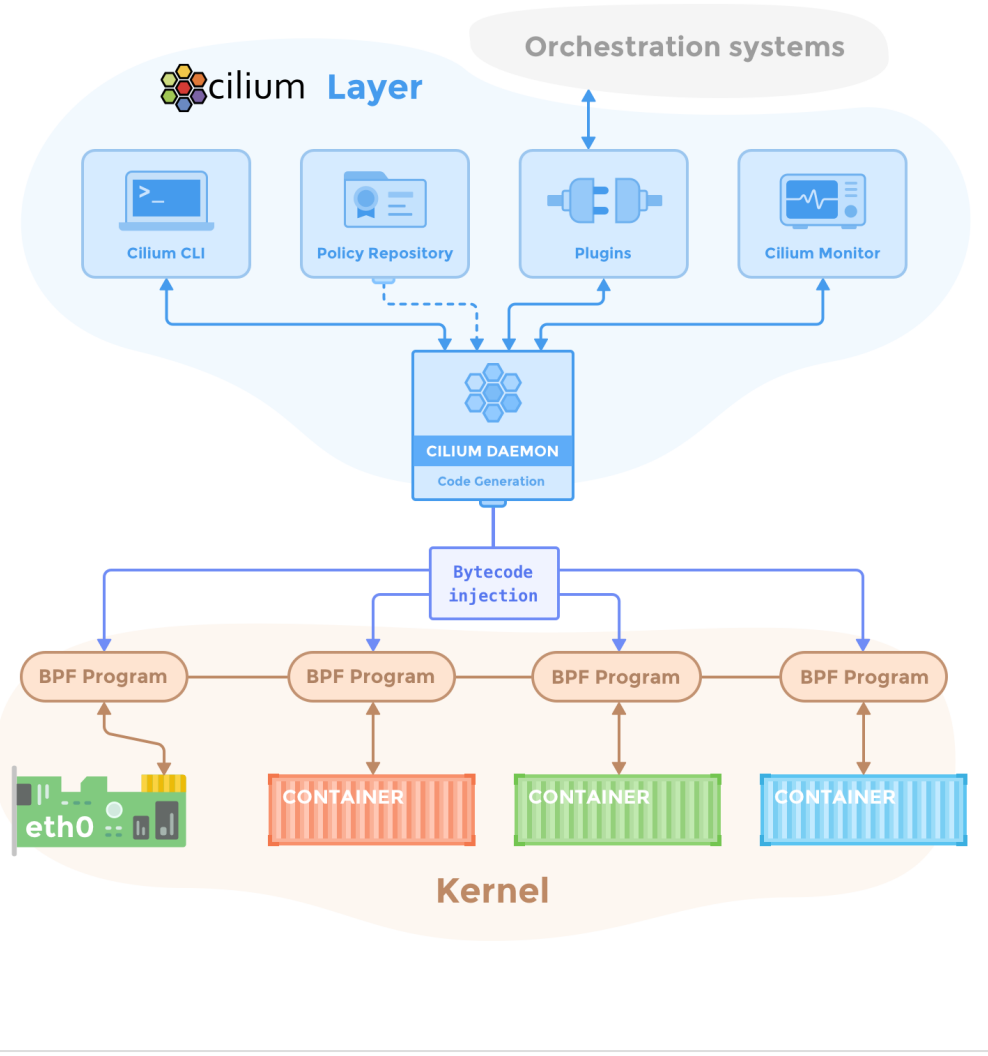


Hubble

Also, meet Hubble: Hubble is a fully distributed networking and security observability platform for Cloud Native workloads.

Hubble is an open source software and 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



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:

```
cat /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

🔀 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, all marked as `NotReady`. This is normal, since the CNI is disabled, and we will install Cilium in the next step. If you don't see

all nodes, the workers nodes might still be joining the cluster. Relaunch the command until you can see all three nodes listed.

Now that we have a Kind cluster, let's install Cilium on it!

🖥️ The Cilium CLI

The `cilium` CLI tool can install and update Cilium on a cluster, as well as activate features —such as Hubble and Cluster Mesh.

```
❯ cilium install
🔮 Auto-detected Kubernetes kind: kind
✨ Running "kind" validation checks
✅ Detected kind version "0.20.0"
ℹ️ Using Cilium version "v1.14.1"
🔮 Auto-detected cluster name: kind-kind
🔮 Auto-detected kube-proxy has been installed
```

⬢ Install Cilium

⬢ The Cilium CLI

The `cilium` CLI tool is provided in this environment to install and check the status of Cilium in the cluster.

Let's start by installing Cilium on the Kind cluster:

```
cilium install
```

Wait for the installation to finish —usually about a minute— and check the status with:

```
cilium status --wait
```

Now that Cilium is functional on our cluster, let's deploy a demo application on it!

? Cilium Install Quiz

The Cilium CLI allows to (select all answers that apply)

- Install Cilium on a cluster
- Check the Cilium status on a cluster
- Install Kubernetes

🛸 Star Wars Demo

To learn how to use and enforce policies with Cilium, we have prepared a demo example.

In the following Star Wars-inspired example, there are three microservice applications: `deathstar`, `tiefighter`, and `xwing`.

🌐 The `deathstar` service

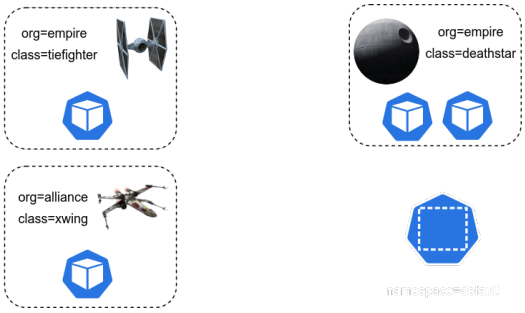
The deathstar runs an HTTP webservice on port 80, which is exposed as a Kubernetes Service to load-balance requests to `deathstar` across two pod replicas.

The `deathstar` service provides landing services to the empire's spaceships so that they can request a landing port.

👮 Allowing ship access

The `tiefighter` pod represents a landing-request client service on a typical empire ship and `xwing` represents a similar service on an alliance ship.

With this setup, we can test different security policies for access control to `deathstar` landing services.



🚀 Deploying a demo app

🚀 Deploy the demo

Let's deploy a simple empire demo application. It is made of several microservices, each identified by Kubernetes labels:

- the Death Star: `org=empire, class=deathstar`
- the Imperial TIE fighter: `org=empire, class=tiefighter`
- the Rebel X-Wing: `org=alliance, class=xwing`

The deployment also includes a deathstar-service, which load-balances traffic to all pods with label `org=empire, class=deathstar`.

Let's install everything via the manifest `http-sw-app.yaml`:

```
kubectl apply -f https://raw.githubusercontent.com/cilium/
cilium/HEAD/examples/minikube/http-sw-app.yaml
```

In the output, note the newly created objects. To verify that everything is properly deployed, run:

```
kubectl get pods,svc
```

Each pod will go through several states until it reaches `Running` at which point the pod is ready. Re-launch the command until all pods are in a `Running` state.

Each pod will also be represented in Cilium as an Endpoint. To retrieve a list of all endpoints managed by Cilium, the Cilium Endpoint (or `cep`) resource can be used:

```
kubectl get cep --all-namespaces
```

As you can see the demo application is properly installed now. Let's launch our first test!

🌐 Death Star access

From the perspective of the deathstar service, only the ships with label `org=empire` are allowed to connect and request landing!

🛡️ No rules enforced

But we have no rules enforced, so what will happen if not only `tiefighter` but also `xwing` request landing?

Let's find out!

🔍 Check Current Access

To simulate our connectivity tests, we will be executing simple API calls using curl.

Let's test if we can land our TIE fighter on the Death Star by running the following command:

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

The command above lets us get a shell on the tiefighter pod and run a HTTP POST request to the deathstar Service to request landing.

The command should work —as the TIE fighter and the Death Star are on the same side of the galactic wars (i.e. the bad guys).

Now test if you can land your X-wing (i.e. the good guys) with:

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

So far, it seems access is allowed! This is good for the rebel alliance —unfettered access to the DeathStar— but this should not be allowed, right?

There is a security policy missing!

Identities and Cloud Native

IP addresses are no longer relevant for Cloud Native workloads. Security policies need something else.

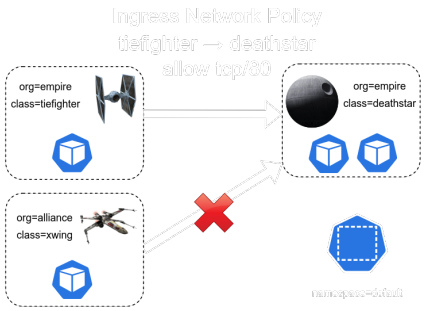
Cilium provides this: Cilium uses the labels assigned to pods to define security policies.

Writing Network Policies

We'll start with a basic policy restricting `deathstar` landing requests to only the ships that have the label `org=empire`.

This blocks any ships without the `org=empire` label to even connect to the `deathstar` service.

This is a simple policy that filters only on network layer 3 (IP protocol) and network layer 4 (TCP protocol), so it is often referred to as a L3/L4 network security policy.



👮🏻 A first L3/L4 policy

✍️ Crafting a Network Policy

We'll start with the basic policy to restrict deathstar landing requests to the ships that have label `org=empire` only.

Before we start changing the policies right away, let's think about what a corresponding network policy should look like.

We need to match on empire ships only, so we need to match on that label:

```
spec:
  description:
"L3-L4 policy to restrict deathstar access to empire ships
only"
  endpointSelector:
    matchLabels:
      org: empire
      class: deathstar
```

Furthermore, we have to make sure that ingress from endpoints with the label `empire` is allowed to port `80` for protocol `tcp`:

```
  ingress:
  - fromEndpoints:
    - matchLabels:
        org: empire
    toPorts:
    - ports:
      - port: "80"
        protocol: TCP
```

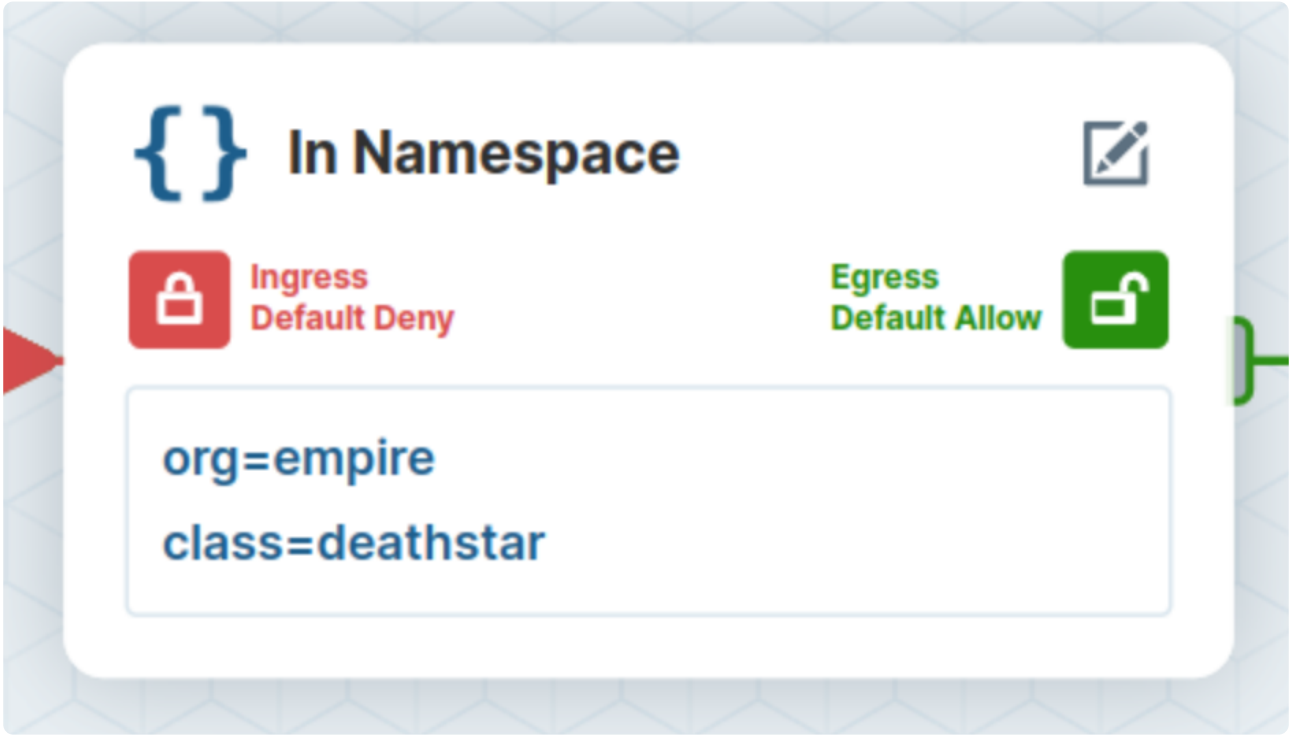
Visualizing the Network Policy

While this example is relatively simple, Kubernetes operators sometimes find it difficult to understand and to build network policies.

Cilium has a solution for this: alongside your current server tab, there is another tab, called **Network Policy Editor**. Click on it to see a visual and interactive representation of your policy, and change some values if you want to better understand what the policy is doing.

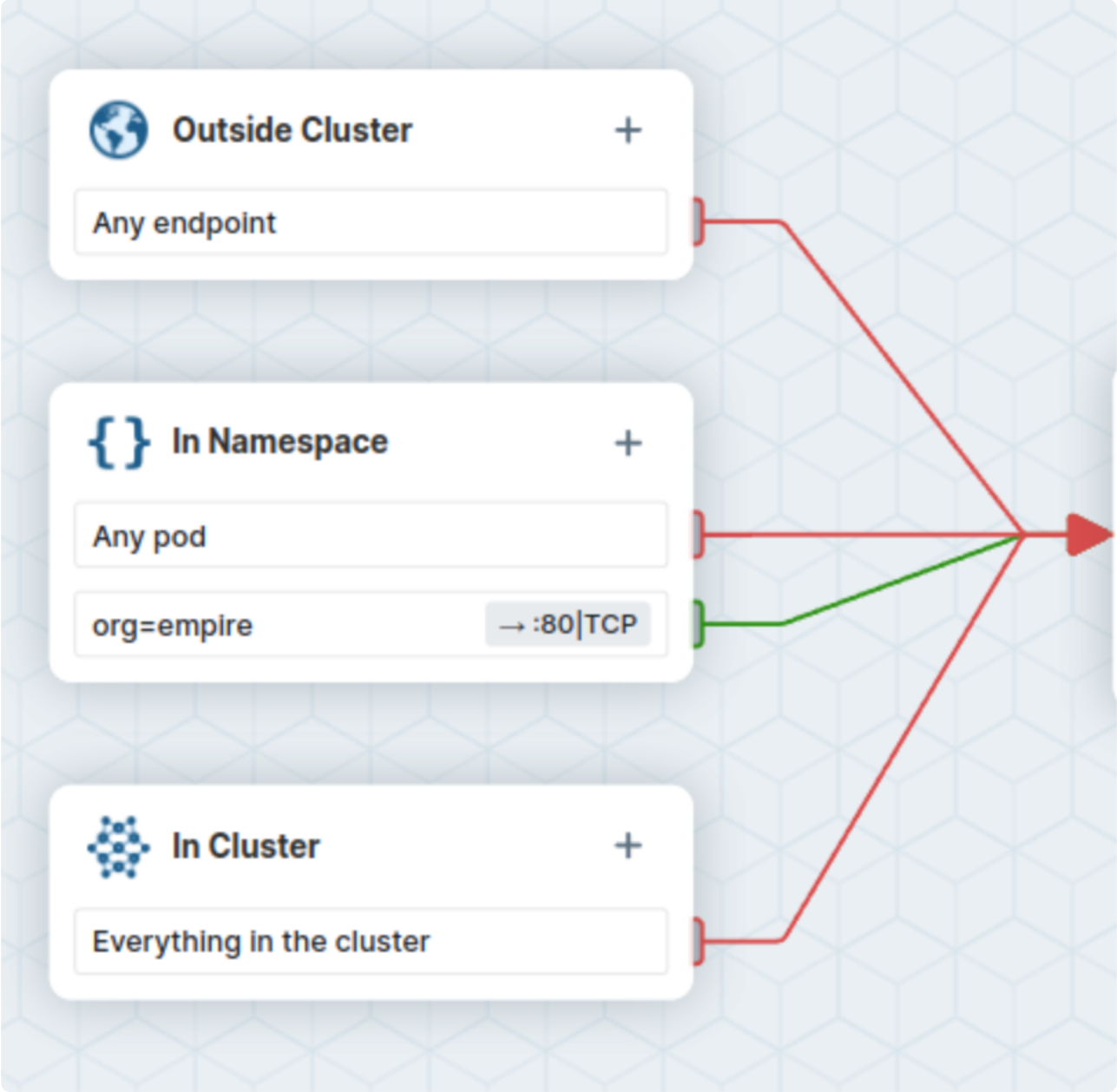
> **NOTE**
>
> If the Editor doesn't load properly, you can open it in a new browser tab

The central part of the diagram represents the Network Policy selector:



Each side shows respectively which ingress and egress are allowed for this workload.

Verify that the current selector (`org=empire, class=deathstar`) is only allowed ingress traffic from pods labeled as `org=empire`:



Check the corresponding Network Policy manifests in the editor below. Note that they can be viewed either in the standard Kubernetes Network Policy format, or using the Cilium Network Policy specification, which allows for extra features.

Kubernetes Network Policy | Cilium Network Policy

> 💡 **NOTE**
>
> This interface is public and can be accessed on 🔗 editor.cilium.io

Enforcing the Network Policy

Once you are done visualizing the policy in the editor, change back to the `>_ Terminal` tab. There we can apply a preconfigured network policy with the values discussed above to our demo system:

```
kubectl apply -f https://raw.githubusercontent.com/cilium/
cilium/HEAD/examples/minikube/sw_l3_l4_policy.yaml
```

Now let's try to land the empire tiefighter again (HTTP POST from tiefighter to deathstar on the /v1/request-landing path):

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

This still works, which is expected.

In comparison, if you try to request landing from the xwing pod, you will see that the request will eventually time out:

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

Kill the request with `Ctrl+C` once you realize that it hangs.

We have successfully blocked access to the deathstar from an X-Wing ship. Let's now see how we could make this policy a bit more fine-grained using L7 rules.

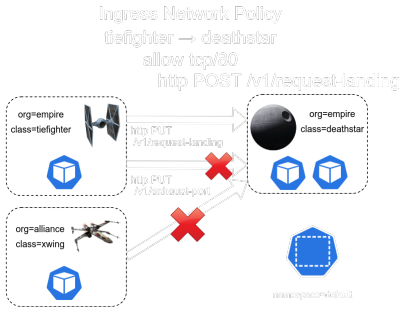
Tighter Rules

So far it was sufficient to either give `tiefighter`/`xwing` full access to deathstar's API or no access at all. But are you absolutely sure that you can trust all thousands of tiefighter pilots of the entire empire?

We must 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.

Filtering on HTTP

So we need to filter on a higher level: we need to filter the actual HTTP requests!



🌐 Apply and Test HTTP-aware L7 Policy

📻 A Radio Contact

The Death Star is now well secured to only allow Imperial vessels to access it.

What if the Rebellion was able to take control of an Imperial Tie Figther though? You —a Rebel officer- have just taken control of a Tie Fighter and are approaching the Death Star.

A radio signal from the Death Star comes through and is asking for your credentials and intentions. Quick, answer the call with the following command:

```
starcom --interactive
```

🚥 Filtering Paths

Consider that the deathstar service exposes some maintenance APIs which should not be called by random empire ships. To see why those APIs are sensitive, run:

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

Yes, there is a `Panic`: the Death Star just exploded!

As you can see, this leads to rather unwanted results. While this is an illustrative example, unauthorized access such as above can have adverse security repercussions. We need to enforce policies on the HTTP layer, at layer 7, to limit what exact APIs the tiefighter is allowed to call —and which are not.

We need to extend the existing policy with an HTTP rule such as:

```
rules:
  http:
  - method: "POST"
    path: "/v1/request-landing"
```

This will restrict API access to only the `/v1/request-landing` path and will thus prevent users from accessing the `/v1/exhaust-port`, which caused a crash as we saw earlier.

👁️ Visualizing the Network Policy

Let's have a look at how such an extended policy might look like. Switch to the 🔗 **Network Policy Editor** tab.

> 💡 **NOTE**
>
> If the Editor doesn't load properly, you can open it in a new browser tab

The new L7 network policy is now loaded, including the HTTP API filter.

Verify in the Kubernetes manifest that this policy will filter access to the deathstar by path.

```
- fromEndpoints:
    - matchLabels:
        org: empire
  toPorts:
    - ports:
        - port: "80"
          protocol: TCP
      rules:
        http:
          - method: POST
            path: /v1/request-landing
```

Enforcing the Network Policy

Once you have done so, switch back to the `>_ Terminal` tab and apply this updated rule to your demo system:

```
kubectl apply -f https://raw.githubusercontent.com/cilium/
cilium/HEAD/examples/minikube/sw_l3_l4_l7_policy.yaml
```

Run the same test as above, and see the different outcome:

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

As you can see, with Cilium L7 security policies, we are able to restrict tiefighter's access to only the required API resources on deathstar, thereby implementing a “least privilege” security approach for communication between microservices.

? Network Policies Quiz

Network Policies (select all answers that apply)

- Network Policies can block or allow traffic between pods
- L3/L4 Network Policies can filter HTTP requests
- L7 Network Policies can filter on HTTP paths
- Cilium supports standard Kubernetes Network Policies

🏆 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!

🗺️ Keep Learning!

Did you like this lab?

This is just the beginning of a journey to learn eBPF-based tools!

Check out the Cilium World map to discover where it could lead you!

Exam Challenge

Exam Instructions

The death star was destroyed by an X-Wing. And what does the empire do? They built another death star. And this time we really have to ensure that no X-Wing can ever access it.

Your task is to create a rule file under the name `/root/policies/sneak.yaml`, containing a rule called `rule1` which restricts access to the death star via L3-L4 policies to empire ships only.

ⓘ **Notes:**

- In the **</> Editor** tab, the `sneak.yaml` file has been pre-created but is missing the right parameters.
- The policy has been loaded in the **🔗 Network Policy Editor** tab. You can also open it in a new tab if necessary.
- Make sure to apply the policy file in the **>_ Terminal** tab!
- Your organization is called `empire`, you only want to allow ships of the class `tiefighter`
- The endpoint the ships should reach is of the class `deathstar`, and of the organization `empire`
- We also want to limit to port 80 and the TCP protocol.
- Test TIE fighter access with `kubectl exec tiefighter -- curl -s -XPOST deathstar.default.svc.cluster.local/v1/request-landing`
- Test X-Wing access with `kubectl exec xwing -- curl -s -XPOST deathstar.default.svc.cluster.local/v1/request-landing`
- In the second tab, you can find the network policy editor which can help you creating the necessary rule.
- You might find Cilium documentation about L3/L4 policies helpful
- When the policy is updated, apply it with `kubectl apply -f /root/policies/sneak.yaml`

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 Cilium Network Policies on your Kubernetes cluster.

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