ISOVALENT
now part of cisco

Tutorial: Cilium Network Policy in Practice (Part 2)

Paul Arah | Published: Mar 15, 2024 | Updated: Mar 15, 2024 | Cilium

Table of contents

Just as a city's traffic smoothly flows when guided by clear rules and regulations, our Kubernetes cluster is about to undergo a transformation guided by Cilium Network Policies. Imagine navigating a city: certain areas might be restricted to pedestrians only, while others might allow limited car access. Similarly, when designing Network Policies, we need to consider: Who can access what? Define which applications (represented by "pods" in Kubernetes) can communicate with each other, on which ports, and using what protocols. Where can they go? This determines which resources (internal services or external resources) each pod can access.

In this second part of the network policy series, we'll navigate the complexities of designing these rules, exploring some common user stories that shed light on the practical implementation of Cilium Network Policies. The first part of this series introduces Cilium network policy and compares Kubernetes network policy with the Cilium network policy.

To help demonstrate how to use network policy to secure an application, we have prepared a hypothetical three-tier demo app. Our demo application comprises a frontend, a backend HTTP API, and a database. You can find a link to the Kubernetes manifest file on Github.

Prerequisites

To follow along with this tutorial, we assume you have the following tools set up:

- Kubectl installed locally.
- Kind installed locally.
- The Cilium CLI installed.
- Hubble CLI installed.

- And a warm cup of your favorite beverage.😎

Make a quick detour and install these tools if you haven't.

Prepping Our Kubernetes Environment

We will provision our Kubernetes environment with Kind; you can follow along using Kind like we've done here or any other Kubernetes environment with Cilium installed. The Kind config file below creates a Kind Kubernetes cluster with one control plane node and one worker node. The `disableDefaultCNI` field is set to true to start our cluster with the Kind default network plugin disabled.

```
#kind-config.yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
- role: worker
networking:
disableDefaultCNI: true
```

Save this config to a file named `config. yaml` and create the cluster using the command `kind create cluster --config config.yaml`

Installing Cilium

To install Cilium, ensure that you have the Cilium CLI installed first and then run the command `cilium install`. The Cilium CLI will install Cilium automatically with the best configuration for our environment. Confirm your Cilium installation works with the `cilium status –wait` command. You should see an output similar to the one below. Check the Cilium installation guide for more information if you're using a different

Kubernetes environment. If you run into issues setting up Kind with Cilium, there's a Kind installation guide in the Cilium documentation.

```
$ cilium status --wait
/¯¯\
/¯¯\__/¯¯\ Cilium: OK
\__/¯¯\__/ Operator: OK
/¯¯\__/¯¯\ Envoy DaemonSet: disabled (using embedded mode)
\__/¯¯\__/ Hubble Relay: disabled
\__/ ClusterMesh: disabled
```

Setting Up Hubble for Observability

Hubble is the observability component of Cilium. Setting up Hubble will come in handy for inspecting, debugging, and visualizing network flows while crafting Cilium network policies that implement our user stories. Hubble already comes bundled with the default Cilium installation. To enable Hubble, run the command below:

```
$ cilium hubble enable
```

To confirm and validate that Hubble is enabled and running, run the `cilium status –wait` command. The *Hubble Relay* field from the command output should now display "OK".

Hubble provides both a CLI and a UI client for observing and querying network traffic flow data. We will use the Hubble CLI in this tutorial. Install the Hubble CLI if you have not already done so.

To enable access to the Hubble API from our local machine, we create a port forward to our local machine. Run the `cilium hubble port-forward&` command. Run the `hubble status` to validate that the Hubble CLI client can access the hubble API.

```
$ hubble status
Healthcheck (via localhost:4245): Ok
Current/Max Flows: 2,391/8,190 (29.19%)
Flows/s: 5.87
Connected Nodes: 2/2
```

Deploying the Demo Workloads

Deploy the demo application on the cluster using the command:

```
$ kubectl apply -f
https://raw.githubusercontent.com/networkpolicy/demos/main/blog
-1/demo.yaml
```

Check the created resources. You should have 3 deployments, 3 services, and 2 pods per deployment.

```
$ kubectl get deploy,svc,pod

NAME READY UP-TO-DATE AVAILABLE AGE
deployment.apps/backend 2/2 2 2 15h
deployment.apps/database 2/2 2 2 15h
deployment.apps/frontend 2/2 2 2 15h
```

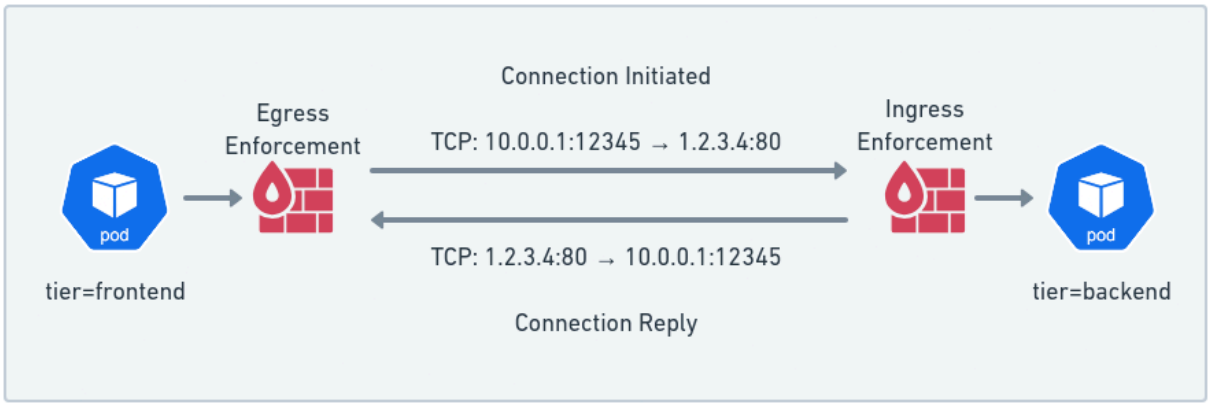
Categorizing Ingress and Egress Traffic

To put it simply, ingress refers to incoming/inbound traffic, and egress refers to outgoing/outbound traffic. Depending on which pod's

perspective we're evaluating the traffic from, this could vary. Let's go through a couple of scenarios.

When a user accesses our frontend application, this is ingress traffic from the frontend pods perspective. When our frontend application makes an HTTP call to the backend API, this is egress traffic from the frontend pod perspective, but from the backend pod perspective, this is ingress traffic. To allow traffic between two pods, the egress side of the sending pod and the ingress side of the receiving pod must both allow traffic. This ensures compliance with zero-trust principles by allowing each pod to enforce its policy instead of relying on the policies of other pods.

When our API pod queries the database pod and receives the results, a common mistake is to evaluate the response from the database as egress traffic leaving the database pod, but this isn't the case. The request and reply all happen under the same underlying TCP connection, which counts only as egress traffic from the backend pod and ingress traffic to the database pod.

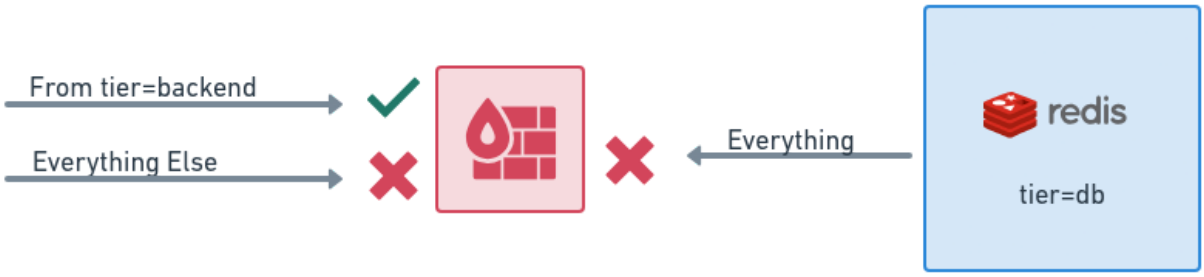


Writing Cilium Network Policies

We've been given the requirements of implementing the following traffic rules in our cluster:

User story 1 – Lockdown the Database, Allow Ingress Only From The Backend

As an administrator, I want to ensure my database pods only receive traffic from the backend pods and can't send traffic anywhere else



This is a common use case, and most managed database providers implement some sort of ingress traffic filtering. To implement this user story, we want our database pods to receive ingress traffic from only the frontend pods, and we also want our database pod to block all egress traffic from leaving the pods. When a Pod is created, Cilium creates an endpoint for the pod(s) and their containers and assigns the endpoint an IP address. We can apply endpoints-based policies using the pod labels.

Currently, if we make a request to the database service or pod from any pod, the request goes through.

Sidenote: Kubernetes generates unique names for pods. While following along, you should replace our autogenerated Kubernetes pod names with yours.

```
kubectl exec backend-784c669968-q6w9w -- curl -s database/PING

{"PING":[true,"PONG"]}
```

We can observe the network traffic flow on the backend pods using Hubble. Conveniently, the hubble CLI client allows us to filter the network traffic flow data it gathers by pod labels. Let's filter network traffic data on the pod with labels `tier=backend`.

```
Mar 14 13:55:24.606: default/backend-784c669968-srl9z:53200
(ID:40136) -> default/database-6fcd88f48-sr8bz:7379
(ID:29609) to-endpoint FORWARDED (TCP Flags: ACK)
Mar 14 13:55:24.606: default/backend-784c669968-srl9z:53200
(ID:40136) -> default/database-6fcd88f48-sr8bz:7379
(ID:29609) to-endpoint FORWARDED (TCP Flags: ACK, PSH)
Mar 14 13:55:24.606: default/backend-784c669968-srl9z:53200
```

The resulting data shows network traffic originating from the backend pods and flowing to database pods.

To implement our network policy, we identify the specific pod(s) to which we want our policy to be applied. In our case, the database pods with the label `tier=database`. Note that if the endpoint selector field is empty, the policy will be applied to all pods in the namespace.

The standard network security posture of Kubernetes allows all network traffic. We can restrict all communication by introducing a “default deny” posture and allowing only the desired network communication. When a rule selects an endpoint, and we specify the traffic direction(i.e. ingress or egress), the endpoint automatically enters a default deny mode.

```
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
name: database-policy
namespace: default
spec:
endpointSelector:
```

Let’s create our Cilium Network Policy in a file named `database-policy.yaml`.

```
$ kubectl apply -f database-policy.yaml

$ kubectl get cnp #cnp is short for the CiliumNetworkPolicy
```

When we try to reach the database pods from any other pods in the cluster or when we try to reach other pods from inside the database pod, we're unable to do so. The default deny posture on the network policy has restricted both ingress and egress traffic on the database pods.

```
$ kubectl exec backend-784c669968-k6m85 -- curl -s
database/PING

#kill the request by pressing CTRL + C
```

Observing the network traffic flow using hubble, we see that connections are dropped due to the database policy in place.

```
Mar 14 14:12:01.154: default/backend-784c669968-srl9z:35866
(ID:40136) <> default/database-6fcd88f48-f9xmg:7379
(ID:29609) Policy denied DROPPED (TCP Flags: SYN)
Mar 14 14:12:34.178: default/backend-784c669968-srl9z:35866
(ID:40136) <> default/database-6fcd88f48-f9xmg:7379
(ID:29609) policy-verdict:none INGRESS DENIED (TCP Flags:
SYN)
```

We now add an ingress rule that allows traffic for the backend pods while leaving the egress rule in its default deny state.

```
  - matchLabels:
    tier: backend
    toPorts:
    - ports:
    - port: "7379"
  egress:
  - {}
```

We can test our ingress traffic ruleset by reaching the database pods from any backend pods. The container image for our database pod uses a minimal image without the curl binary. To test our egress traffic, we'll use a slight workaround with busybox and wget.

```
$ kubectl exec backend-784c669968-vww2g -- curl -s
database/PING
$ kubectl exec database-54dddb86bf-56blf -- busybox wget -O -
frontend
```

Our request from the backend pod to the database pod goes through, while our request from the database pod to the frontend pod fails. If we try to reach any external domains or IP addresses from the database pod, this fails too!

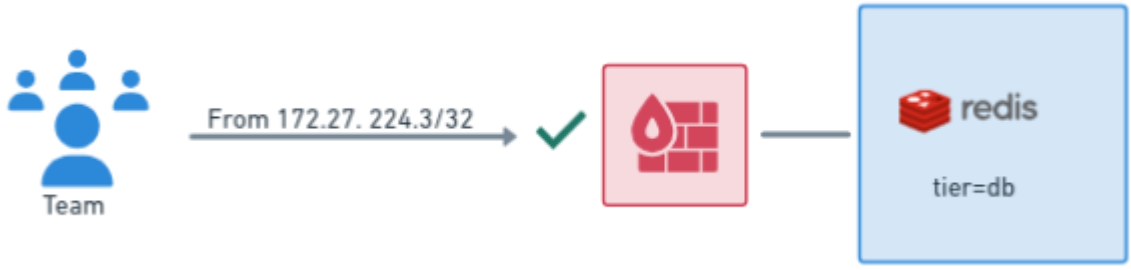
Observing the network traffic flow with hubble, we see that packets are now being allowed only from the backend pods to the database pods.

```
Mar 14 14:16:07.889: default/backend-784c669968-srl9z:38510
(ID:40136) -> default/database-6fcd88f48-f9xmg:7379
(ID:29609) to-endpoint FORWARDED (TCP Flags: ACK, PSH)
Mar 14 14:16:07.889: default/backend-784c669968-srl9z:38510
(ID:40136) -> default/database-6fcd88f48-f9xmg:7379
(ID:29609) to-endpoint FORWARDED (TCP Flags: ACK, FIN)
Mar 14 14:16:07.889: default/backend-784c669968-srl9z:38510
```

User Story 2 – IP/CIDR Ingress Traffic Filter for Database Access

We have a team of developers and administrators who need to access the cluster from outside the office, and we're routing their incoming traffic through a VPN with the IP address `172.27.224.3`*, as an administrator, I want to ensure all developers and administrators using this VPN can access the database pods.*

Since all the incoming external traffic to our database pod is routed through a VPN, this user story entails that the only external IP address that can access our database pods is the IP address of the VPN server. Cilium Network Policy provides CIDR-based policies for controlling traffic to known IP addresses and CIDRs. The `fromCIDR` and `fromCIDRSet` `fields` in the spec define ingress traffic rules, and `toCIDR` and `toCIDRSet` define egress traffic rules. The toCIDRSet field allows us to define exclusion rules for subnets within a CIDR.

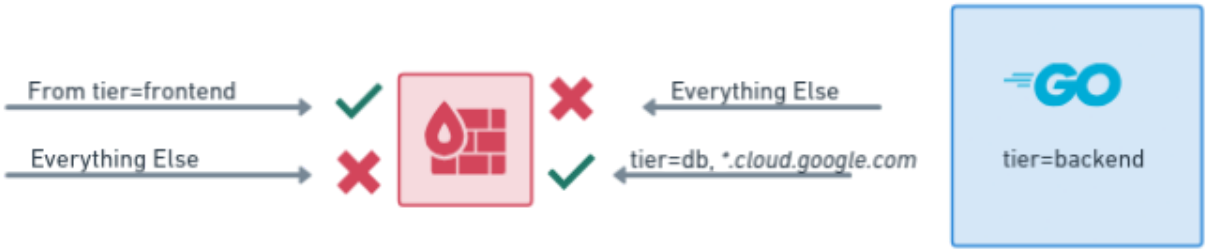


We can update our database policy to include the ingress rule that implements our user story.

```
  - fromCIDRSet:
  - cidr: 102.213.50.174/32
  toPorts:
  - ports:
  - port: "443"
  egress:
  - {}
```

User Story 3 – Filter Egress Traffic by DNS Name

*As an administrator, I want the backend pods to accept ingress traffic only from the frontend pods. I want backend pods to be able to only reach the database pods and also be able to access our external resources on Google Cloud with the domain name wildcard pattern *.cloud.google.com.*



Let's break down what this user story entails. We want to apply a policy on the backend pods that only allows ingress traffic from the frontend pod and egress traffic to the database pods and the external domain name pattern *.cloud.google.com.*

We've written policy rules based on endpoints using matching pod labels in the previous user story. The wildcard domain requirement of this user story introduces a new type of ruleset, DNS-based policies. The CilumNetworkPolicy allows us to define rules based on DNS names and patterns for endpoints not managed by Cilum. The DNS names get

resolved to their IP addresses. DNS-based rules are handy in scenarios where IPs change or are not known beforehand. We define the rules in the `toFQDNs` field of the spec. These rules can be based on DNS names using the `matchName` field or DNS wildcard patterns using the `matchPattern` field.

```
- toEndpoints:
- matchLabels:
io.kubernetes.pod.namespace: kube-system
k8s-app: kube-dns
toPorts:
- ports:
- port: "53"
protocol: UDP
```

For egress traffic to internal and external domain names, the pods rely on kube-dns to resolve DNS queries. We need to explicitly allow egress traffic to port 53/UDP only to kube-dns pods in the cluster.

```
kubectl apply -f backend-policy.yaml
kubectl get cnp

NAME AGE
backend-policy 4h28m
database-policy 4h28m
```

When we try to reach our backend pods from the database pod, the request fails because our policy doesn't allow ingress from any other source except the frontend pods. Our egress policy only allows egress traffic to the database pods and resources in the Google Cloud domain on port 80 for HTTP and port 443 for HTTPS. Suppose a malicious workload in the container tries to exfiltrate data to an external destination, our egress policy now prevents this.

```
$ kubectl exec database-54dddb86bf-56blf -- busybox wget -O -
backend
$ kubectl exec database-54dddb86bf-56blf -- busybox wget -O -
example.com
```

Our policy now prevents our backend pods from sending egress traffic to unknown destinations while simultaneously accepting traffic from the fronted pods.
Let's have a look at the network traffic flow data from Hubble when we filter on the `tier=database` label.

```
$ hubble observe --from-label tier=database

kube-system/coredns-5d78c9869d-mmxnp:53 (ID:941) policy-
verdict:none EGRESS DENIED (UDP)
Mar 14 14:36:51.838: default/database-6fcd88f48-sr8bz:54632
(ID:29609) <> kube-system/coredns-5d78c9869d-mmxnp:53 (ID:941)
Policy denied DROPPED (UDP)
```

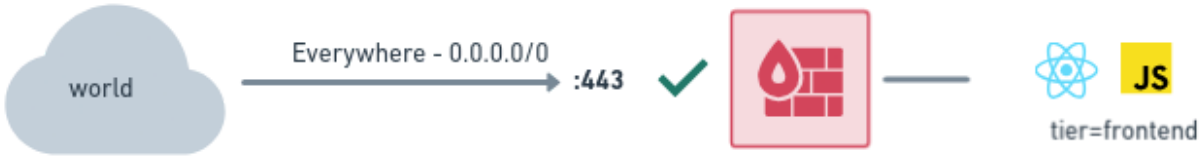
We see that our existing database Cilium network policy prevents egress traffic from even leaving the database pods to the backend pods

When we filter on the `tier=backend` label, our backend Cilium network policy prevents egress traffic from leaving the backend pods to any other domain name apart from the one we've specified.

```
$ hubble observe --from-label tier=backend

Mar 14 14:55:13.538: default/backend-784c669968-srl9z:49770
(ID:40136) <> example.com:80 (world) Policy denied DROPPED
(TCP Flags: SYN)
Mar 14 14:55:15.554: default/backend-784c669968-srl9z:49770
(ID:40136) <> example.com:80 (world) policy-verdict:none
```

User Story 4 – Frontend Accepts World Ingress Traffic via TCP Only

As an administrator, I want our frontend pods to be accessible from any external IP address but only on ports 80 and 443. The only protocol allowed should be TCP.



For public-facing applications like our frontend pods, we can't possibly account for every source IP/CIDR range ingress traffic will originate from. To implement the first part of this user story, we want the frontend pods to accept ingress traffic from every external IP address. Entities-based policies offer abstractions for defining rulesets in scenarios like this. Entities are used to describe remote peers which can be categorized without necessarily knowing their IP addresses. The entity that satisfies the requirement of this user story is the world entity. The world entity represents all endpoints outside the cluster. This corresponds to 0.0.0.0/0 in CIDR notation. Cilium also provides Layer 4 policies that can be specified separately or in addition to an existing Layer 3 policy. Next, the protocol requirement of this user story can be implemented with a Layer 4 policy that enforces a rule that only allows connections on the TCP protocol.

We create a frontend policy file and include an entity-based policy that allows ingress traffic from every external source.

```
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
name: frontend-policy
namespace: default
spec:
endpointSelector:
```

Notice that we included an egress rule to allow traffic to the backend pods? This bidirectional rule is important when writing network policies. We allow egress traffic on the source endpoint and allow ingress traffic on the destination endpoint.

```
$ kubectl apply -f frontend-policy.yaml
$ kubectl get cnp

NAME AGE
backend-policy 22h
database-policy 22h
frotnend-policy 18s
```

We can access our frontend pods externally by using the port forwarding feature from kubectl. We test that our network policy rules work as expected by opening a TCP and UDP connection then seeing if the connection goes through.

To access our frontend pods externally, let's create a port forward from our cluster to our local machine.

```
$ kubectl port-forward service/frontend 8000:80
```

We can now access our frontend service locally.

```
$ curl localhost:8000
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
```

We open TCP connection with netcat and send a random message and get a valid reply

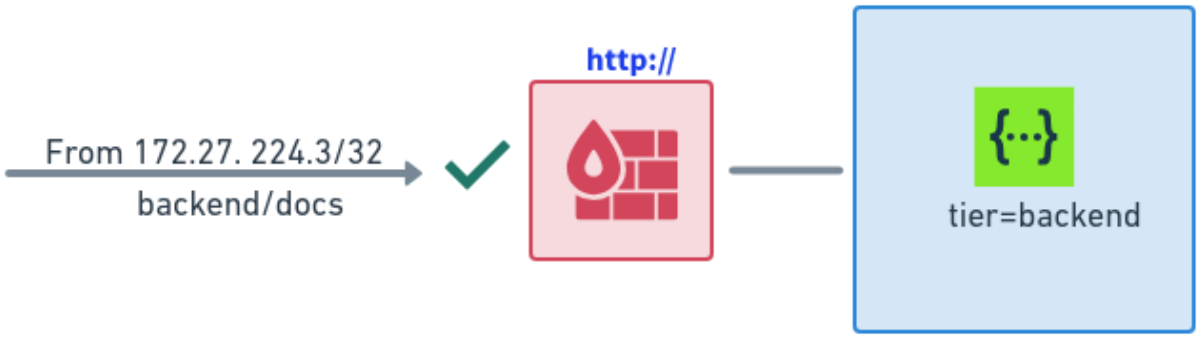
```
nc localhost 8000 #send a random message

HTTP/1.1 400 Bad Request
Server: nginx/1.25.3
Date: Wed, 14 Feb 2024 12:22:32 GMT
Content-Type: text/html
Content-Length: 157
```

When we open a UDP connection our policy prevents this and the connection is abruptly terminated.

```
$ nc -u localhost 8000
```

User Story 5 – Docs HTTP Route Accessible Only From Specific IP Address

We have internal documentation for our backend Rest APIs and we auto-generate the OPEN API spec from our backend codebase and serve the UI via swagger on the `/docs` *HTTP route. We want this route to be accessible to developers and administrators as long as they are using our VPN.*



In addition to the CIDR-based policies we've seen in the previous user story, this user story introduces Layer 7 policies to satisfy the HTTP route requirement. On Layer 3, we need a CIDR rule to allow ingress traffic from 172.224.3/32 (i.e the IP address of the VPN server), and on the Layer 7 side, we need a rule that allows access to the HTTP route `/docs` only.

Let's update our existing backend policy file to include these new rulesets

```
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
name: backend-policy
spec:
endpointSelector:
matchLabels:
```

Combining Policies

The network policy specification dictates that the rules are logically OR'ed (not AND'ed). Following the YAML syntax, when we place rules as separate sequences(arrays/lists) indicated with the "-" symbol, they are evaluated individually as different rules.

Suppose we want a policy that accepts ingress traffic from the CIDR range *172.17.0.0/16* on port *443* only.

The policy below actually allows ingress traffic from the CIDR range 172.17.0.0/16 and also allows ingress traffic from any endpoints to 443. This is different from what we originally intended. By specifying the port rule with a "-" in front of it, we now have a rule that allows ingress traffic from any endpoint to our pod using port *443*.

```
metadata:
name: egress-to-private-vm-443
spec:
endpointSelector:
matchLabels:
app: foo
egress:
```

Notice how a single extra character entirely changes how our policy is interpreted? Our policy now allows more connectivity than we originally intended. To fix this remove the "-" character in front of the port rule and nest the port rule under the toCIDRSet rule.

```
metadata:
name: egress-to-private-vm-443
spec:
endpointSelector:
matchLabels:
app: foo
egress:
```

Cleaning Up

Let's delete all the resources we've created so far.

```
$ kubectl delete -f
https://raw.githubusercontent.com/networkpolicy/demos/main/blog
-1/demo.yaml

$ kubectl delete cnp frontend-policy backend-policy database-
policy
```

Summary

Cilium network policies act as a kind of "border control" and "traffic light" for our Kubernetes city, first by regulating who can enter or leave the city. Then, once you're in the city, network policies determine which zones(endpoints) within the city you can access, through which entry points(ports) can you access these zones, and using what type of vehicles(protocol). As a city's traffic flows well when guided by clear rules and regulations; in the same vein, Cilium network policies ensure predictable and secure communication pathways in our Kubernetes cluster. Cilium network policies enable us to define fine-grained network security policies within Kubernetes. In the various user stories we've explored, we've written policies based on endpoint names, entities,

IP/CIDR, DNS names, and HTTP. Beyond the scenarios covered in this blog post, Cilium supports defining ICMP/ICMPv6 rules at layer 4. At layer 7, Cilium goes beyond HTTP by supporting defining connectivity rules based on other high-level API protocols like gRPC, Kafka, etc. Cilium also features cluster-wide network policies for defining security policies that are scoped to the entire cluster. Host policies that can enforce connectivity rules on the node hosts and they take the form of Cilium cluster-wide network policies. We did not cover this in the blog post. Check out the host firewall lab if you would like to learn more about this feature.

All the YAML manifest files used in the tutorial can be found here on GitHub.

If you have questions or feedback, reach out via the network policy channel in the Cilium Slack.

Dive into Cilium using the hands-on lab, Getting Started with Cilium.

Get hands-on practice with the Network policy lab.

Try out the host policy feature with the Cilium Host Firewall lab.

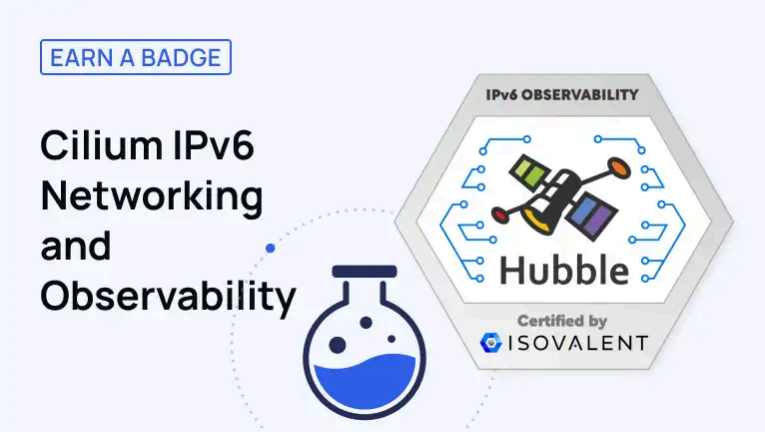
AUTHOR

Paul Arah

Community Builder, Security

Paul is a security-focused community builder at Isovalent – the company behind Cilium & Tetragon. Before joining Isovalent, Paul worked in the startup world as a backend and infrastructure-focused software engineer using various cloud-native technologies, including Kubernetes. Paul has also worked as a software engineering trainer, designing curriculum and content to train budding software engineers. You can find Paul hanging out in various open-source communities and speaking at conferences like the Open Source Festival. Paul enjoys swimming, cycling, and mountain biking.

Related



Labs • Feb 14, 2023

Cilium IPv6 Networking and Observability

Learn how simple IPv6 can be installed and operated with Cilium and Hubble. With Kubernetes' IPv6 support improving in recent releases and Dua...

Briefs • On-demand

Isovalent Enterprise for

Learn about the key features a Isovalent Enterprise for Cilium managing Kubernetes clusters

By Roland Wolters