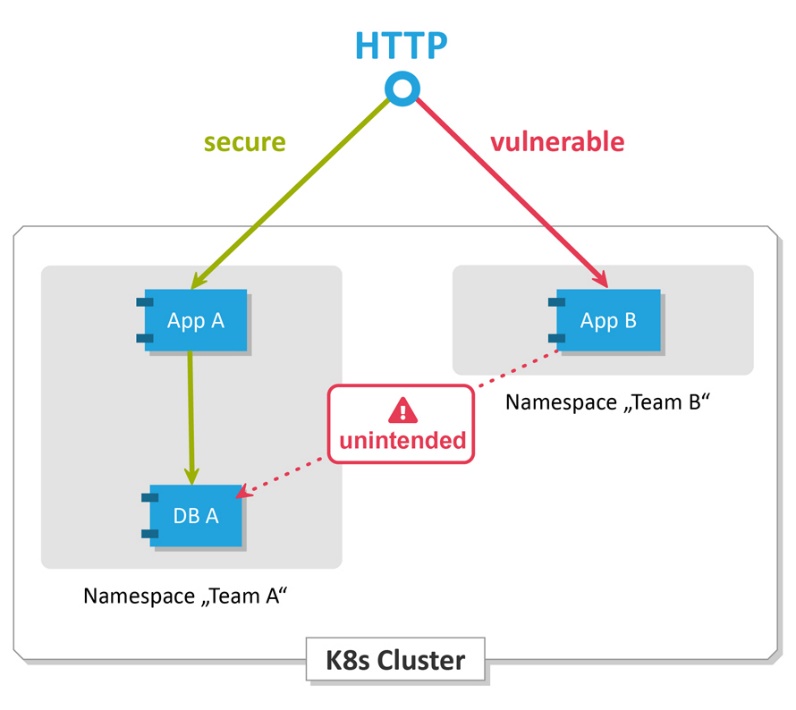
Securing Traffic using Network Policies

* Pods and Services have cluster-wide IP addresses and can communicate across different namespaces and nodes.
* That makes it easy to model distributed applications, but it means you can't have segregated networks within the cluster or stop applications in Pods reaching outside of the cluster.
* Those are security gaps which the Network Policy API fills, allowing you to model network access as part of your application deployment.



**Network Policies:**

* If you want to control traffic flow at the IP address or port level (OSI layer 3 or 4), then you might consider using Kubernetes NetworkPolicies for particular applications in your cluster.
* NetworkPolicies are an application-centric construct which allow you to specify how a [pod](https://kubernetes.io/docs/concepts/workloads/pods/) is allowed to communicate with various network "entities" over the network.
* NetworkPolicies apply to a connection with a pod on one or both ends, and are not relevant to other connections.

Note: Network policies are implemented by the **network plugin**. To use network policies, you must be using a networking solution which supports NetworkPolicy.

Creating a NetworkPolicy resource without a controller that implements it will have no effect.

**Example**

|  |  |
| --- | --- |
| apiVersion: networking.k8s.io/v1  kind: NetworkPolicy  metadata:    name: test-network-policy    namespace: default  spec:    podSelector:      matchLabels:        role: db    policyTypes:      - Ingress      - Egress    ingress:      - from:          - ipBlock:              cidr: 172.17.0.0/16              except:                - 172.17.1.0/24          - namespaceSelector:              matchLabels:                project: myproject          - podSelector:              matchLabels:                role: frontend        ports:          - protocol: TCP            port: 6379    egress:      - to:          - ipBlock:              cidr: 10.0.0.0/24        ports:          - protocol: TCP            port: 5978 | 1. Isolates "role=db" pods in the "default" namespace for both ingress and egress traffic (if they weren't already isolated) 2. (Ingress rules) allows connections to all pods in the "default" namespace with the label "role=db" on TCP port 6379 from:    * Any pod with IP addresses in the ranges 172.17.0.0–172.17.0.255 and 172.17.2.0–172.17.255.255 (ie, all of 172.17.0.0/16 except 172.17.1.0/24)    * Any pod in a namespace with the label "project=myproject"    * Any pod in the "**default**" (local) namespace with the label "role=frontend" 3. (Egress rules) allows connections from any pod in the "default" namespace with the label "role=db" to CIDR 10.0.0.0/24 on TCP port 5978 |

**And vs Or Rule**

|  |  |
| --- | --- |
| **AND** | **OR** |
| ingress:  - from:    - namespaceSelector:        matchLabels:          project: myproj1      podSelector:        matchLabels:          role: client | ingress:  - from:    - namespaceSelector:        matchLabels:          project: myproj1    - podSelector:        matchLabels:          role: client |
| Contains a **single** from element.  Allowing connections from Pods with the label **role=client** in namespaces with the label **project: myproj1** | Contains two elements in the from array.  Allows connections from Pods in the local Namespace with the label **role=client, or** from any Pod in any namespace with the label **project: myproj1** |

Lab

**~~Step1: Install k3d CLI~~**

**~~# On Windows using Chocolatey:~~**

~~choco install k3d~~

**~~# On MacOS using brew:~~**

~~brew install k3d~~

**~~# On Linux:~~**

~~curl -s https://raw.githubusercontent.com/rancher/k3d/main/install.sh | bash~~

~~OR~~

~~Use the install instructions~~ [~~https://k3d.io/v5.0.0/#installation~~](https://k3d.io/v5.0.0/#installation)

**~~Verify installation:~~**

~~k3d version~~

~~Note: k3d requires Docker - Docker Engine on Linux or Docker Desktop on Mac/Windows, so you can't use it with any other container runtime.~~

~~k3d clusters use the~~ **~~Flannel CNI plugin~~** ~~by default (like most clusters), but you can configure a new cluster with no network plugin at all.~~

**~~Step2: Create a new cluster with no networking:~~**

~~k3d cluster create labs-netpol -p "30000-30040:30000-30040@server:0" --k3s-arg "--flannel-backend=none@server:0" --k3s-arg "--disable=servicelb@server:0" --k3s-arg "--disable=traefik@server:0" --k3s-arg "--disable=metrics-server@server:0"~~

* ~~This creates a single-node cluster without the~~ **~~Flannel CNI~~** ~~installed~~
* ~~Ports are published to the local machine, so you can use localhost with NodePort Services~~
* ~~Extra k3d features like LoadBalancer support and metrics are turned off~~

~~The node is in the~~**~~NotReady~~**~~state, because there's no network installed.~~

~~kubectl get nodes~~

~~Dig a bit deeper and you'll see the DNS server isn't running:~~

~~kubectl get deploy -n kube-system~~

**~~Step3: DNS requires a network plugin.~~**

[~~Calico~~](https://docs.projectcalico.org/getting-started/kubernetes/)~~is a network plugin which supports NetworkPolicy. It's open-source and very commonly used where network policy is required.~~

~~The network plugin runs as a DaemonSet, but it also uses privileged init containers to modify the network configuration of OS on the node. That's why we're using k3d, so we don't impact the networking on your main cluster:~~

* [~~k3d/calico.yaml~~](https://kubernetes.courselabs.co/labs/networkpolicy/specs/k3d/calico.yaml)~~- is from the Calico docs, it includes the network plugin and RBAC rules~~

~~kubectl apply -f https://kubernetes.courselabs.co/labs/networkpolicy/specs/k3d/calico.yaml~~

~~Note: You'll see various Calico Pods starting up in the kube-system namespace~~

~~Your node should be ready now, and the coredns~~

~~kubectl get nodes~~

**If using Azure, enable Network Policy while creating the cluster.**

az aks create --resource-group myResourceGroup --name myAKSCluster --network-plugin azure **--enable-network-policy**

**Connecting nginx from another pod without Network Policies**

**nginx.yaml**

apiVersion: v1

kind: Pod

metadata:

  name: nginx

  labels:

    app: nginx

spec:

  containers:

  - name: nginx-con

    image: nginx:latest

    ports:

    - containerPort: 80

**Execute the following commands**

kubectl apply -f nginx.yaml

kubectl get pods -o wide

kubectl run tmp-shell1 --rm -it --labels="app=client" --restart=Never --image nicolaka/netshoot -- /bin/bash

# curl http://<IPofNginx> Note: It connects and returns result.

# ctrl + c

# exit

**Deny All Network Policy:**

There are no ingress or egress permissions in the rules, so this is a policy which allows nothing - effectively blocks all outgoing and incoming communication to all Pods

**deny-all.yaml**

kind: NetworkPolicy

apiVersion: networking.k8s.io/v1

metadata:

  name: deny-all

spec:

  podSelector: {}

  policyTypes:

  - Ingress

  - Egress

kubectl apply -f deny-all.yaml

kubectl run tmp-shell1 --rm -it --labels="app=client" --restart=Never --image nicolaka/netshoot -- /bin/bash

# curl http://<IPofNginx> Note: It fails to connect.

# ctrl + c

# exit

**kubectl delete -f deny-all.yaml**

**Create Allow all Network Policy:**

**allow-all.yaml**

kind: NetworkPolicy

apiVersion: networking.k8s.io/v1

metadata:

  name: allow-all

spec:

  podSelector: {}

  policyTypes:

  - Ingress

  - Egress

  ingress:

  - {} # allow all

  egress:

  - {} # allow all

**Execute following commands**

kubectl apply -f allow-all.yaml

kubectl run tmp-shell1 --rm -it --labels="app=client" --restart=Never --image nicolaka/netshoot -- /bin/bash

# curl http://<IPofNginx> Note: It connects and returns result.

# ctrl + c

# exit

kubectl delete -f allow-all.yaml

**nginx-policy.yaml:** Allows ingress from any pod with **label app=client**;

kind: NetworkPolicy

apiVersion: networking.k8s.io/v1

metadata:

  name: nginx-policy

spec:

  podSelector:

    matchLabels:

**app: nginx**

  policyTypes:

  - Ingress

  ingress:

  - from:

    - podSelector:

        matchLabels:

**app: client**

**Following connects**

k run client1 --image nicolaka/netshoot --labels "app=client" -it --rm --restart=Never -- sh

# curl <IP of POD>

**Following fails to connect**

k run client2 --image nicolaka/netshoot --labels "app=client1" -it --rm --restart=Never -- sh

# curl <IP of POD>

**client-policy.yaml**

kind: NetworkPolicy

apiVersion: networking.k8s.io/v1

metadata:

  name: client-policy

spec:

  podSelector:

    matchLabels:

**app: client**

  policyTypes:

  - Egress

  egress:

  - to:

    - podSelector:

        matchLabels:

**app: nginx**

  - to:

    - namespaceSelector:

        matchLabels:

          kubernetes.io/metadata.name: kube-system

      podSelector:

        matchLabels:

          k8s-app: kube-dns

    ports:

    - protocol: TCP

      port: 53

    - protocol: UDP

      port: 53

Allows egress to the two pod with label app=nginx and the DNS server (so the app can get the IP addresses of the API Pods)

**Execute following commands**

kubectl apply -f deny-all.yaml

kubectl apply -f nginx-policy.yaml

kubectl apply -f client-policy.yaml

kubectl run tmp-shell1 --rm -it **--labels="app=client"** --restart=Never --image nicolaka/netshoot -- /bin/bash

# curl http://<IPofNginx> Note: It connects and returns result.

# ctrl + c

# exit