# Practice M3: Security and Policies

For the purpose of this practice, we will assume that we are working on a machine with either a Windows 10/11 or any recent Linux distribution and there is a local virtualization solution (like VirtualBox, Hyper-V, VMware Workstation, etc.) installed

***Please note that long commands may be hard to read here. To handle this, you can copy them to a plain text editor first. This will allow you to see them correctly. Then you can use them as intended***

## Part 1: Authentication, Authorization and Admission Control

Let's start with a simple (two or three nodes) custom made cluster (**not** Minikube or KIND based)

### Default Cluster User

Log on to the control plane node (we assume that the cluster is created with Vagrant)

**vagrant ssh node1**

Check the cluster configuration file

**cat ~/.kube/config**

We can see that there are several notable sections – ***clusters***, ***contexts***, ***current-context***, and ***users***

**Clusters** section contains currently registered clusters with their certificate authority data, control plane and name

**Contexts** section contains list of currently registered contexts that combine clusters with users

**Current context** specifies against which context our commands will be fired by default and without the need to explicitly specifying it

**Users** section contains the list of registered users with their name, certificate and key

Should we want to control more than one cluster or interact as different users, then we must alter this file or use another copy

### Additional Cluster Users

We will create:

* two namespaces – one for **Production** (**demo-prod**) and one for **Development** (**demo-dev**)
* two users – **John** (**john**) and **Jane** (**jane**)

**John** will have **Edit** access to **Production** and **View** access to **Development** and **Jane** will have the opposite – **View** access to **Production** and **Edit** access to **Development**

There are multiple ways to achieve the above. This applies to most of the sub steps as well

We will do it by utilizing the option for using **X.509** client certificates for cluster authentication

#### OS Users Creation and Preparation

Let's start with the users

Create the **John** OS user

**sudo useradd -m -s /bin/bash john**

And set some password

**sudo passwd john**

Switch to its home folder

**cd /home/john**

Create a folder for the certificate related files and change to it

**sudo mkdir .certs && cd .certs**

Create a private key

**sudo openssl genrsa -out john.key 2048**

Create a certificate signing request

**sudo openssl req -new -key john.key -out john.csr -subj "/CN=john"**

Sign the **CSR** with the **Kubernetes CA** certificate

**sudo openssl x509 -req -in john.csr -CA /etc/kubernetes/pki/ca.crt -CAkey /etc/kubernetes/pki/ca.key -CAcreateserial -out john.crt -days 365**

Return to the home folder of our user

**cd**

Create the user in **Kubernetes** *(in fact, with this approach, we are changing the* ***kubeconfig*** *file that is in use, our file)*

**kubectl config set-credentials john --client-certificate=/home/john/.certs/john.crt --client-key=/home/john/.certs/john.key**

Create context for the user as well

**kubectl config set-context john-context --cluster=kubernetes --user=john**

We can prove that indeed we changed our configuration file by executing any of these

**kubectl config get-users**

**kubectl config get-contexts**

**kubectl config view**

Now, let’s create a personal configuration file to be used only by **John**

We will base it on the one we just changed (our configuration file)

Create a folder to store the user configuration

**sudo mkdir /home/john/.kube**

Create a copy first

**sudo cp ~/.kube/config /home/john/.kube/config**

Open it for editing

**sudo vi /home/john/.kube/config**

Reuse (keep) the **clusters** section. Substitute (replace) everything from **contexts** onwards with the lines bellow

**contexts:**

**- context:**

**cluster: kubernetes**

**user: john**

**name: john-context**

**current-context: john-context**

**kind: Config**

**preferences: {}**

**users:**

**- name: john**

**user:**

**client-certificate: /home/john/.certs/john.crt**

**client-key: /home/john/.certs/john.key**

Save and close the file

*Please note that in the above example we are using* ***client-certificate*** *and* ***client-key*** *and not* ***client-certificate-data*** *and* ***client-key-data*** *as with the* ***admin.conf*** *file (or our own configuration file). We may go with the second pair, but then we must first encode the content of both files using* ***base64*** *and then use the result here*

*Of course, now knowing what we produced, we could have done it differently by creating the file from scratch*

Change the ownership of the files

**sudo chown -R john: /home/john/**

Now, repeat the procedure for **Jane** as well

#### Namespaces

Create the two namespaces

**kubectl create namespace demo-dev**

**kubectl create namespace demo-prod**

Now switch to **john** for example

**su - john**

And try to access the cluster by asking for information about it and a few of cluster's resources

**kubectl cluster-info**

**kubectl get nodes**

**kubectl get pods**

**kubectl get pods -n demo-prod**

Neither one of the above won't work (for now). The same will happen if we try with the other user

Exit the user’s session

**exit**

#### Roles and Bindings Setup

The sole reason behind the current situation is that they do not have any roles (on namespace or cluster level) attached yet

To achieve our goal, we can use two of the available **ClusterRole** resources – view and edit

And attach them via a **RoleBinding** to the appropriate user and namespace

We can list them together with one additional cluster role with

**kubectl get clusterroles view edit cluster-admin**

Should we want to explore, for example the **edit** role, we can execute the following

**kubectl describe clusterrole edit**

Okay, let's start with **John**

As he will have access in two namespaces, we should prepare two **RoleBinding** resource manifests

We can combine them in one file named **part1/john-role-bindings.yaml** with the following content

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

  name: john

  namespace: demo-prod

subjects:

- kind: User

  name: john

  apiGroup: rbac.authorization.k8s.io

roleRef:

  kind: ClusterRole

  name: edit

  apiGroup: rbac.authorization.k8s.io

---

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

  name: john

  namespace: demo-dev

subjects:

- kind: User

  name: john

  apiGroup: rbac.authorization.k8s.io

roleRef:

  kind: ClusterRole

  name: view

  apiGroup: rbac.authorization.k8s.io

Save and close the file and push it to the cluster

**kubectl apply -f part1/john-role-bindings.yaml**

Repeat the steps for **Jane** as well but of course, adjusted to match our goal

Create a file **part1/jane-role-bingings.yaml** with the following content

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

  name: jane

  namespace: demo-prod

subjects:

- kind: User

  name: jane

  apiGroup: rbac.authorization.k8s.io

roleRef:

  kind: ClusterRole

  name: view

  apiGroup: rbac.authorization.k8s.io

---

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

  name: jane

  namespace: demo-dev

subjects:

- kind: User

  name: jane

  apiGroup: rbac.authorization.k8s.io

roleRef:

  kind: ClusterRole

  name: edit

  apiGroup: rbac.authorization.k8s.io

Save and close the file and push it to the cluster

**kubectl apply -f part1/jane-role-bindings.yaml**

Okay, we are done for now

#### Explore and Test the Access

Acting as the **administrator** user, let's create two pods – one in the **Production** and another in the **Development** namespace

**kubectl run pod-prod --image=shekeriev/k8s-oracle --labels=app=oracle --namespace demo-prod**

**kubectl run pod-dev --image=shekeriev/k8s-oracle --labels=app=oracle --namespace demo-dev**

Let's switch to **John** and check if he can create resources in **Production** and view resources in **Development**

**su - john**

Now, check if he can create pods in both namespaces

**kubectl run pod-prod-john --image=shekeriev/k8s-oracle --labels=app=oracle --namespace demo-prod**

**kubectl run pod-dev-john --image=shekeriev/k8s-oracle --labels=app=oracle --namespace demo-dev**

And then services

**kubectl expose pod pod-prod --port=5000 --name=svc-prod-john --namespace demo-prod**

**kubectl expose pod pod-dev --port=5000 --name=svc-dev-john --namespace demo-dev**

Then if he can list them

**kubectl get pods,svc -n demo-prod**

**kubectl get pods,svc -n demo-dev**

And finally, if he can delete resources

**kubectl delete pod pod-prod-john -n demo-prod**

**kubectl delete pod pod-dev -n demo-dev**

As we can see, everything behaves just like we wanted

Should we want, we can do some checks for **Jane** as well, but we should not see any surprises

Do not forget to exit the respective user’s session

**exit**

#### Easier Permission Check

Okay, isn't there another way to check who can do what? Yes, there is

In fact, there are multiple ways. They vary and include native functionality in **kubectl**, specialized security-related plugins for **kubectl**, or third-party tools and applications

Let's check the native functionality in **kubectl**

Acting as the **admin**, execute the following three commands

**kubectl auth can-i create pods**

**kubectl auth can-i create pods -n demo-prod**

**kubectl auth can-i create pods -n demo-dev**

Of course, the **administrator** should receive **yes** as output of all three commands

Now, how can we check for example for **John**? We can switch of course, but let's try another way

**kubectl auth can-i create pods --as john**

**kubectl auth can-i create pods -n demo-prod --as john**

**kubectl auth can-i create pods -n demo-dev --as john**

We can try with **Jane** as well. In the same manner, we can try with other actions (verbs) like **get**, **list**, **watch**, **delete**, **update**, etc.

Furthermore, instead of asking for individual actions, we can ask for everything one can do in a namespace

For example, for **John** in the **Development** namespace, we can use

**kubectl auth can-i --list --namespace demo-dev --as john**

In the result, we can see that our users may be allowed to interact not only with resources, but with API endpoint as well (**Non-Resource URLs**)

#### Clean Up

Acting as the administrator user, execute the following command to remove all artefacts of our tests

**kubectl delete namespace demo-prod demo-dev**

### Service Accounts and API

Access restrictions may be applied not only to human users but to service accounts used by pods as well

The easiest way to demonstrate this is to try to access the **apiserver** from a pod

#### Preparation

Let's create a few resources to experiment with

**kubectl create namespace rbac-ns**

**kubectl run rbac-pod --image=shekeriev/k8s-oracle --namespace=rbac-ns**

**kubectl expose pod rbac-pod --port=5000 --name=rbac-svc --namespace=rbac-ns**

Check which service account has been assigned to the pod

**kubectl get pod rbac-pod -n rbac-ns -o yaml | grep serviceAccount**

It is the one named **default** as we expected

#### First Attempt

Now, jump into the pod

**kubectl exec -it rbac-pod -n rbac-ns -- bash**

Install the missing **curl** command

**apt-get update && apt-get install -y curl**

Prepare a set of environment variables

**APISERVER=https://kubernetes.default.svc**

**SERVICEACCOUNT=/var/run/secrets/kubernetes.io/serviceaccount**

**NAMESPACE=$(cat ${SERVICEACCOUNT}/namespace)**

**TOKEN=$(cat ${SERVICEACCOUNT}/token)**

**CACERT=${SERVICEACCOUNT}/ca.crt**

Check that we can access the **apiserver**

**curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" -X GET ${APISERVER}/api**

Okay, we can. But can we list for example the pods or the services in the same namespace?

Let's check for the pods

**curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" \**

**-X GET ${APISERVER}/api/v1/namespaces/rbac-ns/pods**

And then for the services

**curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" \**

**-X GET ${APISERVER}/api/v1/namespaces/rbac-ns/services**

One more try. Let's check if we can list anything (for example pods) in the **default** namespace

**curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" \**

**-X GET ${APISERVER}/api/v1/namespaces/default/pods**

Again, no luck ☹

Leave the pod

**exit**

The reason is that this **default** service account doesn't have any permissions granted by default

We could validate this using the **auth** command from earlier

**kubectl auth can-i get pods --namespace rbac-ns --as system:serviceaccount:rbac-ns:default**

**kubectl auth can-i create pods --namespace rbac-ns --as system:serviceaccount:rbac-ns:default**

We can give additional permissions to the service account in use (**default**) or create a new one

Let's first extend the permissions of the existing one

#### Extend Permissions

Before we continue with the permissions, let's ask for the service account

**kubectl get serviceaccount -n rbac-ns**

We can shorten the above to

**kubectl get sa -n rbac-ns**

For this task we will create a **Role** and a **RoleBinging** as we are operating in a namespace

To create a role which will allow **get**, **list**, and **watch**, we can execute

**kubectl create role view-pods --verb=get,list,watch --resource=pods --namespace=rbac-ns**

Then, we can bind the role to the service account with

**kubectl create rolebinding view-pods --role=view-pods --serviceaccount=rbac-ns:default --namespace=rbac-ns**

Now, we can repeat part of the procedure from the first attempt

Jump into the pod

**kubectl exec -it rbac-pod -n rbac-ns -- bash**

Prepare a set of environment variables

**APISERVER=https://kubernetes.default.svc**

**SERVICEACCOUNT=/var/run/secrets/kubernetes.io/serviceaccount**

**NAMESPACE=$(cat ${SERVICEACCOUNT}/namespace)**

**TOKEN=$(cat ${SERVICEACCOUNT}/token)**

**CACERT=${SERVICEACCOUNT}/ca.crt**

Let's check for the pods

**curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" \**

**-X GET ${APISERVER}/api/v1/namespaces/rbac-ns/pods**

Wow, we can see them 😊

And what about the services?

**curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" \**

**-X GET ${APISERVER}/api/v1/namespaces/rbac-ns/services**

Still no luck ☹

Exit the pod

**exit**

But if we remember correctly, we did not include this type in the role, so that is why we cannot access it

So, everything is working as expected

We can validate it again with the **auth** command

**kubectl auth can-i get pods --namespace rbac-ns --as system:serviceaccount:rbac-ns:default**

**kubectl auth can-i get services --namespace rbac-ns --as system:serviceaccount:rbac-ns:default**

But what if we want a more complex role, covering for example **pods** and **services**?

Let's dump the existing one in **YAML**

**kubectl get role view-pods -n rbac-ns -o yaml**

So, it seems quite natural. It should be enough to enlist the services there as well

We can either **edit** the role or create a new **YAML** file and **apply** it. Let's go with the **edit**

**kubectl edit role view-pods -n rbac-ns**

Now, we can repeat the procedure and check

We should be able to apply the same set of actions now to services as well

#### New Service Account

Let's go over the procedure of creating a new service account and grant some permissions to it

We will create **demo-sa** in the same namespace – **rbac-ns**

It will have **get**, **list**, **create** and **delete** permissions for **pods** and **get**, **list**, and **create** for **services**

So, it won't happen with one rule as we saw earlier. We should find a way around this

First, let's create the service account

**kubectl create serviceaccount demo-sa --namespace rbac-ns**

Check that it doesn't have some of the permissions that we are about to grant

**kubectl auth can-i get pods --namespace rbac-ns --as system:serviceaccount:rbac-ns:demo-sa**

**kubectl auth can-i get services --namespace rbac-ns --as system:serviceaccount:rbac-ns:demo-sa**

Okay, now let's prepare the role manifest **demo-role.yaml** with the following content

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

  name: demo-role

  namespace: rbac-ns

rules:

- apiGroups:

  - ""

  resources:

  - pods

  verbs:

  - get

  - list

  - create

  - delete

- apiGroups:

  - ""

  resources:

  - services

  verbs:

  - get

  - list

  - create

Save and close the file

Send the **role** to the cluster

**kubectl apply -f demo-role.yaml**

Then create a **RoleBinding** for it

**kubectl create rolebinding demo-role --role=demo-role --serviceaccount=rbac-ns:demo-sa --namespace=rbac-ns**

Check that it got some of the permissions that we wanted to grant

**kubectl auth can-i get pods --namespace rbac-ns --as system:serviceaccount:rbac-ns:demo-sa**

**kubectl auth can-i get services --namespace rbac-ns --as system:serviceaccount:rbac-ns:demo-sa**

**kubectl auth can-i delete pods --namespace rbac-ns --as system:serviceaccount:rbac-ns:demo-sa**

**kubectl auth can-i delete services --namespace rbac-ns --as system:serviceaccount:rbac-ns:demo-sa**

Now, let's start a new pod that will use this service account and check from there

For this we will use the following manifest (**demo-pod.yaml**)

apiVersion: v1

kind: Pod

metadata:

  name: demo-pod

  namespace: rbac-ns

spec:

  containers:

  - image: shekeriev/k8s-oracle

    name: demo-pod

  serviceAccount: demo-sa

  serviceAccountName: demo-sa

Send it to the cluster

**kubectl apply -f demo-pod.yaml**

Now, open a session to the pod

**kubectl exec -it demo-pod -n rbac-ns -- bash**

Install the missing **curl** command

**apt-get update && apt-get install -y curl**

Prepare a set of environment variables

**APISERVER=https://kubernetes.default.svc**

**SERVICEACCOUNT=/var/run/secrets/kubernetes.io/serviceaccount**

**NAMESPACE=$(cat ${SERVICEACCOUNT}/namespace)**

**TOKEN=$(cat ${SERVICEACCOUNT}/token)**

**CACERT=${SERVICEACCOUNT}/ca.crt**

Let's check for the pods

**curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" \**

**-X GET ${APISERVER}/api/v1/namespaces/rbac-ns/pods**

Wow, we can see them 😊

And what about the services?

**curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" \**

**-X GET ${APISERVER}/api/v1/namespaces/rbac-ns/services**

We can see them too 😊

Let's try to delete the service

**curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" \**

**-X DELETE ${APISERVER}/api/v1/namespaces/rbac-ns/services/rbac-svc**

No, we can't. Which is normal. The role dictates it

Let's try to delete one of the pods

**curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" \**

**-X DELETE ${APISERVER}/api/v1/namespaces/rbac-ns/pods/rbac-pod**

Wow, we did it 😊

Exit the pod

**exit**

So, our setup is working

#### Clean Up

It would be enough to delete the namespace

**kubectl delete namespace rbac-ns**

## Part 2: Resource Requirements, Limits and Quotas

### Requests and Limits

Let's check the available resources on the nodes

**kubectl describe nodes**

Focus on the sections **Capacity**, **Allocatable**, **Non-terminated Pods**, and **Allocated resources**

Let's see how we can state how much resource a pod needs to start

We will accomplish this by using the **requests** block

First, create a namespace to accommodate the resources

**kubectl create namespace reslim**

#### Unrestricted Pods

We can start a pod without any restrictions

**kubectl run pod-1 --image=alpine --restart=Never --namespace reslim -- dd if=/dev/zero of=/dev/null bs=16M**

Check where the pod was scheduled and see if there is a change with the resources of the node

**kubectl get pods -n reslim -o wide**

**kubectl describe node <node>**

No, at least on first sight there aren't any significant changes

Let's enter the pod and see how busy it is

**kubectl exec -it pod-1 -n reslim -- top**

We can see that it varies but at times reaches almost 50% CPU, which in our case (a node with 2 CPUs) is equal to 1 CPU

Let's delete this one

**kubectl delete pod pod-1 -n reslim**

#### Resource Requests

And start another but from a manifest and with a reservation of resources

Check the **part2/pod-2.yaml** manifest and apply it

**kubectl apply -f part2/pod-2.yaml**

Again, we can check where it was scheduled

**kubectl get pods -n reslim -o wide**

Then check the situation in the pod

**kubectl exec -it pod-2 -n reslim -- top**

And then the resources of the node

**kubectl describe node <node>**

Finally, we can see that there is resource allocation

It appears to be different that the actual amount of utilized resources

This is due to the fact that we have specified just the amount of resources required to start the pod successfully

It is not limited in any way

Try to start one more but this time using the following deployment manifest

**kubectl apply -f part2/deployment-res.yaml**

If will attempt to schedule 3 replicas

Again, we can check where they were scheduled

**kubectl get pods -n reslim -o wide**

Then check the resources of the nodes

**kubectl describe node <node>**

Okay, we are exhausting the resources

Now, increase the replica count to 5 and check again

**kubectl scale deployment res -n reslim --replicas=5**

**kubectl get deployment -n reslim**

**kubectl get pods -n reslim -o wide**

If there are still free resources, scale to 6, 7, etc. until there is no room. You will see a pod or pods that stay in **Pending** status

Then, if we check the resource utilization of the nodes, we will quickly understand why this is happening

We managed to exhaust all resources (CPU in our case) of the nodes

Should we want details, we can ask for the pending pod's description

**kubectl describe pod <pod-id> -n reslim**

We should see a warning for the failed scheduling with a reason for insufficient CPU

Let's clean a bit

**kubectl delete -f part2/deployment-res.yaml**

**kubectl delete -f part2/pod-2.yaml**

#### Resource Limits

Requesting resources is a way to reserve resources for a particular workload

There are situations in which we would like to limit the amount of resources a pod may use

This is where we use **limits**

Let's use a modified and extended version of the pod manifest (**part2/pod-3a.yaml**)

**kubectl apply -f part2/pod-3a.yaml**

Now, check where the pod went

**kubectl get pods -n reslim -o wide**

And then the resource utilization of the node

**kubectl describe node <node>**

Then check the situation from within the pod

**kubectl exec -it pod-3a -n reslim -- top**

Even though there may be some fluctuations, we will see that it is close to 25%

This, in our case (500m) is correct, because we have 2 CPUs

Now, delete this one

**kubectl delete pod pod-3a -n reslim**

And try another version of the manifest (**part2/pod-3b.yaml**)

**kubectl apply -f part2/pod-3b.yaml**

Then watch the progress

**kubectl get pods -n reslim -o wide -w**

After a while we will see that the pod goes over **ContainerCreating**, **Running** and **OOMKilled**

And then the cycle (without **ContainerCreating**) repeats

This happens because we limited the memory amount that the containers in the pod may use

And once this limit is reached containers in the pod are being restarted (as this is the default policy)

We managed to see how limits are working and how they differ from requests

Now, let's clean

**kubectl delete -f part2/pod-3b.yaml**

### Limit Ranges

There is a better way to specify request and limits on every pod (or its containers)

For this, we can use the **LimitRange** resource

It sets default, min and max values per resource

Let's explore the **part2/limits.yaml** manifest

And then apply it

**kubectl apply -f part2/limits.yaml**

Then check with

**kubectl describe limitrange limits -n reslim**

Now try to create a few pods

First, one that does not specify anything

**kubectl apply -f part2/pod-4a.yaml**

Then check its resource requests and limits and compare them with the default values that we specified via the policy

Now, start another one (**part2/pod-4b.yaml**)

**kubectl apply -f part2/pod-4b.yaml**

It failed. Why?

And then another one (**part2/pod-4c.yaml**)

**kubectl apply -f part2/pod-4c.yaml**

It failed. Why?

Now, let's clean a bit

**kubectl delete pod pod-4a -n reslim**

**kubectl delete -f part2/limits.yaml**

### Quotas

Quotas behave similarly to **Limit Ranges**. They do not operate on pod level but on namespace level

Explore the **part2/quota.yaml** manifest

And then apply it

**kubectl apply -f part2/quota.yaml**

Check the state

**kubectl get quota -n reslim**

**kubectl describe quota quota -n reslim**

Now spin one pod

**kubectl apply -f part2/pod-5a.yaml**

No success. Why?

Try another one

**kubectl apply -f part2/pod-5b.yaml**

It works. Why?

Check again the stats

**kubectl describe quota quota -n reslim**

It appears that **Limit Ranges** and **Quotas** are better together 😉

Clean by deleting the whole namespace

**kubectl delete namespace reslim**

## Part 3: Network Policies

For this part we must make sure that our cluster is using a **Network plugin** that supports **Network Policies**

Most of the popular plugins support network policies together with many other features

Partial list of them can be found here:

<https://kubernetes.io/docs/tasks/administer-cluster/declare-network-policy/>

### Installation

Their installation process doesn't differ much from the one of the **Flannel** plugin (which does not support network policies)

#### Antrea

Assuming that we created our cluster with

**kubeadm init --apiserver-advertise-address=<cp-address> --pod-network-cidr 172.16.0.0/16**

Then the latest version of the **Antrea** plugin can be installed with

**kubectl apply -f https://raw.githubusercontent.com/antrea-io/antrea/main/build/yamls/antrea.yml**

Should we want to install a particular version (for example, 1.14.0), we can do it with

**kubectl apply -f https://github.com/antrea-io/antrea/releases/download/v1.14.0/antrea.yml**

More information can be found here:

<https://github.com/antrea-io/antrea/blob/main/docs/getting-started.md>

#### Calico

Assuming that we created our cluster with

**kubeadm init --apiserver-advertise-address=<cp-address> --pod-network-cidr 172.16.0.0/16**

Then the **Calico** plugin can be installed with

**kubectl apply -f https://docs.projectcalico.org/manifests/tigera-operator.yaml**

Then download the custom resources manifest

**curl https://docs.projectcalico.org/manifests/custom-resources.yaml -O**

Adjust the configuration if needed by editing the **custom-resources.yaml** manifest *(especially if you used something different than 192.168.0.0/16 for pod network)*

*For example, to change the default CIRD from* ***192.168.0.0/16*** *to* ***172.16.0.0/16****, you can execute*

***sed -i.bak s/192.168/172.16/g custom-resources.yaml***

Then apply it

**kubectl apply -f custom-resources.yaml**

More information can be found here:

<https://docs.projectcalico.org/getting-started/kubernetes/quickstart>

#### Weave Net

Assuming that we created our cluster with

**kubeadm init --apiserver-advertise-address=<cp-address> --pod-network-cidr 10.32.0.0/12**

Then the **Weave Net** plugin can be installed with

**kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"**

Should we want to change the configuration, for example the pod network addresses, we ca change it to

**kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d 'n')&env.IPALLOC\_RANGE=10.200.0.0/16"**

More information can be found here:

<https://github.com/weaveworks/weave>

### Basic Test

Let's do our first test of what network policies can do for us

Create a namespace

**kubectl create namespace basicnp**

Create a simple deployment

**kubectl create deployment oracle --image=shekeriev/k8s-oracle --namespace basicnp**

And then expose it via service

**kubectl expose deployment oracle --port=5000 --namespace basicnp**

Check the resources we have so far

**kubectl get svc,pod -n basicnp**

Now, let's start another pod, which we can use to test the reachability of the service and the pod

**kubectl run tester --image=alpine --namespace basicnp -- sleep 1d**

Enter the additional pod

**kubectl exec -it tester --namespace basicnp -- sh**

Add the **curl** package

**apk add curl**

Then, try to connect to the service we started earlier

**curl --connect-timeout 5 http://oracle:5000**

We should be able to see an answer from the pod which proves that there aren't any restrictions

Close the session

**exit**

Now, let's create a simple network policy that will limit incoming connections to our pod (which is behind the service) only to pods that are labeled in a certain way

The manifest (**part3/oracle-policy.yaml**) has the following content

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

  name: access-oracle

  namespace: basicnp

spec:

  podSelector:

    matchLabels:

      app: oracle

  ingress:

  - from:

    - podSelector:

        matchLabels:

          access: "true"

Send it to the cluster

**kubectl apply -f part3/oracle-policy.yaml**

Ask for details about the policy we just created

**kubectl describe netpol access-oracle -n basicnp**

Enter again the tester pod (which is in the same namespace)

**kubectl exec -it tester --namespace basicnp -- sh**

Then, try to connect to the service we started earlier

**curl --connect-timeout 5 http://oracle:5000**

The connection closes with time out. We are not able to communicate with the pod (behind the service). Why?

Close the session

**exit**

Let's spin another tester pod in the **default** namespace and try from there

**kubectl run tester --image=alpine -- sleep 1d**

Enter the additional pod

**kubectl exec -it tester -- sh**

Add the **curl** package

**apk add curl**

Then, try to connect to the service we started earlier

**curl --connect-timeout 5 http://oracle.basicnp:5000**

No luck again. Why?

Close the session

**exit**

Let's add the missing ***magic*** label to our **tester** pod (the one in the **basicnp** namespace)

**kubectl label pods tester --namespace basicnp access=true**

Establish again a session to it

**kubectl exec -it tester --namespace basicnp -- sh**

And try to connect to the service

**curl --connect-timeout 5 http://oracle:5000**

Wow. Success 😊

Close the session

**exit**

Try from the other **tester** pod (the one in the **default** namespace)

**kubectl exec -it tester -- sh**

Then, try to connect to the service

**curl --connect-timeout 5 http://oracle.basicnp:5000**

No luck again. Why? *Missing label perhaps?*

Close the session

**exit**

What if we add a label to it as well? Let's do it

**kubectl label pods tester access=true**

Try again from the other **tester** pod (the one in the **default** namespace)

**kubectl exec -it tester -- sh**

Then, try to connect to the service

**curl --connect-timeout 5 http://oracle.basicnp:5000**

No luck again. Why? *Because of the different namespace?*

Close the session

**exit**

Edit the policy

**kubectl edit netpol access-oracle -n basicnp**

Add the following row

**- namespaceSelector: {}**

Just before the **podSelector** clause

Save and close. The changes will be applied immediately

Let's test again

**kubectl exec -it tester -- sh**

Then, try to connect to the service

**curl --connect-timeout 5 http://oracle.basicnp:5000**

Success! 😊 Why? 😉 *Perhaps, because of the namespaceSelector?*

Close the session

**exit**

But wait, if we remove the label from the pod (the one in the **default** namespace) what will happen? Will it be able to access it? Let's try it

Remove the label

**kubectl label pod tester access-**

Let's test again

**kubectl exec -it tester -- sh**

Then, try to connect to the service

**curl --connect-timeout 5 http://oracle.basicnp:5000**

Success! 😊 Why? 😉 *Still, perhaps of the namespaseSelector?*

Close the session

**exit**

This documentation excerpt will help in this case

<https://kubernetes.io/docs/concepts/services-networking/network-policies/#behavior-of-to-and-from-selectors>

*Please note that we should be extremely careful when mixing* ***podSelector*** *(which selects which pods will be allowed but only those from the same namespace where the policy is) and* ***namespaceSelector*** *(which selects namespaces for which all pods will be allowed)*

Knowing the above, let's check again how our policy is read by the cluster

**kubectl describe netpol access-oracle -n basicnp**

Now, edit it and remove the **-** symbol before the **namespaceSelector** or **podSelector** option, whichever is listed second. *There should be dash only before the* ***namespaceSelector*** *or the* ***podSelector*** *in order to be considered together as a single rule*

**kubectl edit netpol access-oracle -n basicnp**

And ask the cluster again for the policy

**kubectl describe netpol access-oracle -n basicnp**

So, a single dash may change how the policy is understood by the cluster …

Let's clean a bit

**kubectl delete pod tester**

**kubectl delete namespace basicnp**

### Advanced Interactive Test

This one really cool demo is borrowed from here:

<https://docs.tigera.io/calico/latest/network-policy/get-started/kubernetes-policy/kubernetes-demo>

It is applicable not only to **Calico** as a network plugin but also to others that offer network policy support

Let's create the **frontend**, **backend**, **client**, and **management-ui** apps

**kubectl create -f https://docs.tigera.io/files/00-namespace.yaml**

**kubectl create -f https://docs.tigera.io/files/01-management-ui.yaml**

**kubectl create -f https://docs.tigera.io/files/02-backend.yaml**

**kubectl create -f https://docs.tigera.io/files/03-frontend.yaml**

**kubectl create -f https://docs.tigera.io/files/04-client.yaml**

Of course, we can first download the manifests and explore them

In any case, after we send the manifests to the cluster, we can monitor resources creation process

**kubectl get pods --all-namespaces --watch**

Once everything is in running state, we can check the components of the **management-ui** namespace

**kubectl get all -n management-ui**

Then we can view the UI by visiting **http://<k8s-node-ip>:30002** in a browser

We can see all three parts – **client** (**C**), **frontend** (**F**) and **backend** (**B**)

They have full connectivity

Normally, we do not want it this way. Instead, we want the client to connect only to the frontend which should connect only to the backend

Let's switch to full isolation

**kubectl create -n stars -f https://docs.tigera.io/files/default-deny.yaml**

**kubectl create -n client -f https://docs.tigera.io/files/default-deny.yaml**

If we return to the **management UI** and refresh, we will notice that it is empty

To tackle this, we will allow the UI to access the services using network policy objects

**kubectl create -f https://docs.tigera.io/files/allow-ui.yaml**

**kubectl create -f https://docs.tigera.io/files/allow-ui-client.yaml**

Let's return to the management UI and refresh. We should see the components but without any activity between them

Now, let's create a policy to allow traffic from the frontend to the backend

**kubectl create -f https://docs.tigera.io/files/backend-policy.yaml**

Return to the management UI and refresh. There should be communication between them

Let's expose the frontend service to the client namespace

**kubectl create -f https://docs.tigera.io/files/frontend-policy.yaml**

Return to the management UI and refresh. Now, everything is just the way we wanted 😊

Once we are done experimenting, we can clean up the demo environment

**kubectl delete ns client stars management-ui**

### Advanced Interactive Test #2

This one is an extended and more detailed version of what you have seen so far

You can check the repository here: <https://github.com/shekeriev/k8s-netpol>