Lesson Plan
Network Policy Fundamentals

Summary

1. Lab Setup
2. Simulate a Compromise
3. Protect the Database
4. Default Deny
5. Allow DNS
6. Add Remaining Network Policies

Lab Setup

This lab assumes you already have the "Yet Another Online Bank" (yaobank) installed, as created in the “Installing the Sample Application” module in Week 1.

If you don’t already have it installed then go back and do so now.

For reference, this is architecture of YAOBank:

If you are not already on host1, you can enter host1 by using the multipass shell command.

```
multipass shell host1
```

You can validate the yaobank installation by using kubectl get pods.

```
kubectl get pods -n yaobank
```

Example output:

```
NAME                        READY   STATUS    RESTARTS   AGE
database-6c5db58d95-slsph   1/1     Running   0          26m
summary-85c56b76d7-28chk    1/1     Running   0          26m
summary-85c56b76d7-l7rsv    1/1     Running   0          26m
customer-574bd6cc75-5cwnn   1/1     Running   0          26m
```

Simulate a compromise

To simulate a compromise of the customer pod we will exec into the pod and attempt to access the database directly from there.

Enter the customer pod

First we will find the customer pod name and store it in an environment variable to simplify future commands.

```
CUSTOMER_POD=$(kubectl get pods -n yaobank -l app=customer -o name)
```

Note that the CUSTOMER_POD environment variable only exists within your current shell, so if you exit that shell you must set it again in your new shell using the same command as above.

Now we will exec into the customer pod, and run bash, to give us a command prompt within the pod:

```
kubectl exec -it $CUSTOMER_POD -n yaobank -c customer -- /bin/bash
```

Access the database

From within the customer pod, we will now attempt to access the database directly, simulating an attack. As the pod is not secured with NetworkPolicy, the attack will succeed and the balance of all users will be returned.

```
curl http://database:2379/v2/keys?recursive=true | python -m json.tool
```

Leaving the customer pod

To return from the pod back to our original host command line shell, use the `exit` command:

```
exit
```

Protect the Database

To protect the yaobank database, we will be using kubernetes policy.

Deploy the K8s Network Policy

```
cat <<EOF | kubectl apply -f -
---
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: database-policy
  namespace: yaobank
spec:
  podSelector:
    matchLabels:
      app: database
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: summary
      ports:
      - protocol: TCP
        port: 2379
  egress:
    - to: []
EOF
```

Try the attack again

Now let's repeat the simulated attack.

Execute bash command in customer container

```
kubectl exec -ti $CUSTOMER_POD -n yaobank -c customer -- /bin/bash
```

Try to access the database

```
curl --connect-timeout 3 http://database:2379/v2/keys?recursive=true
```

The curl will be blocked and return no data.

Then remember to `exit` the pod exec and return to the host terminal.

```
exit
```

Default Deny

Now you have seen how easy it is to protect a microservice using network policy, let's look at how we switch the cluster to a default-deny behavior. This is a best practice that ensures that every new microservice deployed needs to have an accompanying network policy and no microservices are accidentally left wide open to attack.

You can do this on a per-namespace scope using Kubernetes Network Policy, but that requires each namespace to have its own default-deny policy, and relies on remembering to create a default-deny policy each time a new namespace is created. This is where Calico policy can be useful. Since Calico's GlobalNetworkPolicy policies apply across all namespaces, you can write a single default-deny policy for the whole of your cluster.

calicoctl

The calicoctl utility is used to create and manage Calico resource types, as well as allowing you to run a range of other Calico specific commands. In this specific case, we are creating a GlobalNetworkPolicy (GNP).

Deploy the network policy

```
cat <<EOF | calicoctl apply -f -
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: default-app-policy
spec:
  namespaceSelector: has(projectcalico.org/name) && projectcalico.org/name
not in {"kube-system", "calico-system"}
  types:
  - Ingress
  - Egress
EOF
```

There a couple of things worth noting:

- For this lab we've chosen to exclude kube-system and calico-system namespaces, since we don't want the policy to impact the Kubernetes or Calico control planes. (This is a good best practice which avoids accidentally breaking the control planes, in case you have not already set up appropriate network policies and/or Calico failsafe port rules that allows control plane traffic. You can then separately write network policy for each control plane component.)
- As the policy contains no rules, it doesn't actually matter what precedence it has, so we did not specify an order value. But for completeness of your learning, omitting the order field on a network policy means it has the lower precedence compared to any Calico network policy which does specify an order, or Kubernetes network policies which have an implicit order of 1000.

Verify default deny is in place

In the "Protect the Database" step above, we only defined network policy for the Database. The rest of our pods should now be hitting default deny (for both ingress and egress) since there's no policy defined to say who they are allowed to talk to.

Let's try to see if basic connectivity works, e.g. DNS.

Execute bash command in customer container

```
kubectl exec -ti $CUSTOMER_POD -n yaobank -c customer -- /bin/bash
```

Now let's try to do a DNS lookup.

```
dig www.google.com
```

That should fail, timing out after around 15s, because we've not written any policy to say DNS (or any other egress) is allowed.

After running the command, remember to exit from the kubectl exec of the customer pod.

```
exit
```

Allow DNS

Lets update our default policy to allow DNS to the cluster-internal kube-dns service.

```
cat <<EOF | calicoctl apply -f -
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
```

```
metadata:
  name: default-app-policy
spec:
  namespaceSelector: has(projectcalico.org/name) && projectcalico.org/name
not in {"kube-system", "calico-system"}
  types:
  - Ingress
  - Egress
  egress:
    - action: Allow
      protocol: UDP
      destination:
        selector: k8s-app == "kube-dns"
        ports:
          - 53
EOF
```

Now we'll attempt to do a DNS lookup again to confirm the policy now allows it.

Execute bash command in customer container.

```
kubectl exec -ti $CUSTOMER_POD -n yaobank -c customer -- /bin/bash
```

Try the DNS lookup again.

```
dig www.google.com
```

This should now succeed.

After running the command, remember to exit from the kubectl exec of the customer pod by typing exit.

```
exit
```

Add remaining Network Policies

We've now defined a default app policy across all of the cluster which results in default-deny behavior except for allowed DNS queries to kube-dns. We also defined a policy for the database earlier in the "Protect the Database" step. But we haven't yet defined policies for the customer or summary pods.

Verify we cannot access the Frontend

Try to access the Yaobank front end via the front end service's NodePort:

```
curl --connect-timeout 3 198.19.0.1:30180
```

This should timeout because we haven't provided detailed policy for the customer and summary pods.

Create policy for the remaining pods

Let's create the remaining policies for all the Yaobank services.

```
cat <<EOF | kubectl apply -f -
---
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: customer-policy
  namespace: yaobank
spec:
  podSelector:
    matchLabels:
      app: customer
  ingress:
    - ports:
      - protocol: TCP
        port: 80
  egress:
    - to: []
---
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: summary-policy
  namespace: yaobank
spec:
  podSelector:
    matchLabels:
      app: summary
  ingress:
    - from:
      - podSelector:
```

```
          matchLabels:
            app: customer
      ports:
      - protocol: TCP
        port: 80
  egress:
    - to:
      - podSelector:
          matchLabels:
            app: database
      ports:
      - protocol: TCP
        port: 2379
EOF
```

Verify everything is now working

Now try accessing the Yaobank front end, it should work again. By defining the cluster wide default deny we've forced the user to follow best practices and define network policy for all the necessary microservices.

```
curl 198.19.0.1:30180
```

The resulting output should contain the following at the bottom:

```
  <body>
        <h1>Welcome to YAO Bank</h1>
        <h2>Name: Spike Curtis</h2>
        <h2>Balance: 2389.45</h2>
        <p><a href="/logout">Log Out >></a></p>
  </body>
```

Examine the policy we just applied

Take a little time to examine the policies we just applied and ensure you understand them.

Notice that the egress rule in the customer-policy is excessively liberal, allowing outbound connection to any destination. While this may be required for some workloads, in this case it is a mistake, perhaps reflecting a lazy developer. We will look at how this can be locked down further by the cluster operator or security admin in the next module.