JTC16 Kubernetes Security Basics

Lab0 - Lab information

With Kubernetes rapidly taking over the IT world it is key that the Kubernetes operator and developer knows about basic Kubernetes security concepts.

This Lab provides you with a hands-on with several of those topics.

Lab sources

All the source code for the lab is available here:

https://github.com/niklaushirt/training

Lab overview

In this Lab you will learn about some basic Kubernetes security paradigms.

1. NetworkPolicies
2. Security Tooling

Lab0 - Lab semantics

Nomenclatures

Shell Commands

The commands that you are going to execute to progress the Labs will look like this:

THIS IS AN EXAMPLE - DO NOT EXECUTE THIS!

```Bash
kubectl create -f redis-slave-service.yaml

 > Output Line 1
 > Output Line 2
 > Output Line 3
 ...
```

> **IMPORTANT NOTE:** The example output of a command is prefixed by ">" in order to make it more distinguishable.
>
> So in the above example you would only enter/copy-paste `kubectl create -f redis-slave-service.yaml` and the output from the command is "Output Line 1" to "Output Line 3"

Code Examples

Code examples are presented like this:

```YAML
apiVersion: lab.ibm.com/v1beta1
kind: MyResource
metadata:
  name: example
spec:
  size: 3
  image: busybox
```

This is only for illustration and is not being actively used in the Labs.

Lab 0 - Prepare the Lab environment

Termnial icon in the bottom dock Before starting the Labs, let's make sure that we have the latest source code from the GitHub repository:

https://github.com/niklaushirt/training

1. Open a Terminal window by clicking on the Termnial icon in the left sidebar - we will use this extensively later as well

2. Execute the following commands to initialize your Training Environment

   ```bash
   ./welcome.sh
   ```

   This will

   - pull the latest example code from my GitHub repository
   - start minikube if not already running
   - installs the registry
   - installs the Network Plugin (Cilium)
   - starts the Personal Training Environment

   > During this you will have to provide a name (your name) that will be used to show your progress in the Instructor Dashboard in order to better assist you.

3. Start the demo application

   ```bash
   kubectl create -f ~/training/deployment/demoapp.yaml
   kubectl create -f ~/training/deployment/demoapp-service.yaml
   kubectl create -f ~/training/deployment/demoapp-backend.yaml
   kubectl create -f ~/training/deployment/demoapp-backend-service.yaml
   ```

4. Wait for the demo application to be available (the status must be 1/1)

   ```bash
   kubectl get pods

   > NAME                               READY   STATUS    RESTARTS   AGE
   > k8sdemo-backend-5b779f567f-2rbgj   1/1     Running   0          21s
   > k8sdemo-backend-5b779f567f-p6j76   1/1     Running   0          21s
   > k8sdemo-bd6bbd548-jcb6r            1/1     Running   0          21s
   ```

5. Open the demo application in the browser

```bash
minikube service k8sdemo-service

> |-----------|-----------------|-------------|-------------------------|
> | NAMESPACE |      NAME       | TARGET PORT |           URL           |
> |-----------|-----------------|-------------|-------------------------|
> | default   | k8sdemo-service |             | http://10.0.2.15:32123 |
> |-----------|-----------------|-------------|-------------------------|
> 🎉  Opening kubernetes service  default/k8sdemo-service in default browser...
```

> If you get the following error
>
> ```yaml
> 💣  Error getting machine status: load: filestore "minikube": open /home/training/.minikube/machines/minikube/config.json: permission denied
>
> 😿  Sorry that minikube crashed. If this was unexpected, we would love to hear from you:
> 👉  https://github.com/kubernetes/minikube/issues/new/choose
> ```
>
> Please execute:
>
> ```
> ~/training/tools/own.sh
> ```

Lab 1 - Network Policies

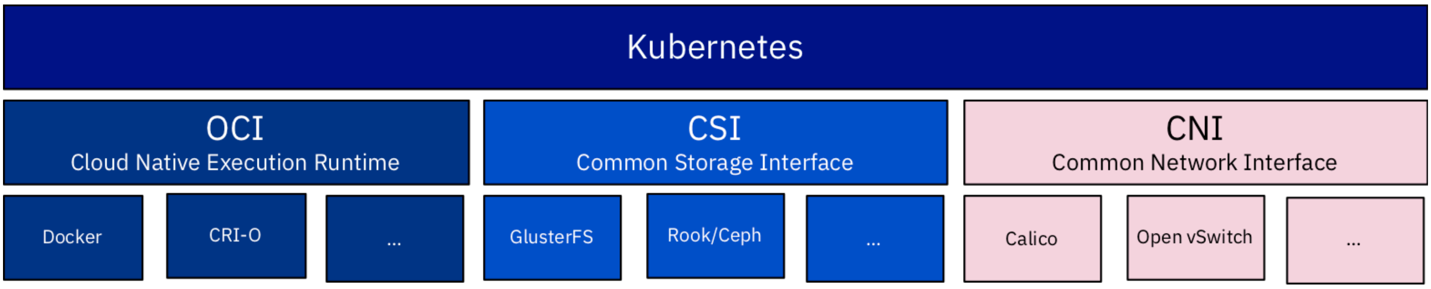
Kubernetes network policies specify how pods can communicate with other pods and with external endpoints. By default, no network policies are set up. If you have unique security requirements, you can create your own network policies.

The following network traffic is allowed by default:

- A pod accepts external traffic from any IP address to its NodePort or LoadBalancer service or its Ingress resource.
- A pod accepts internal traffic from any other pod in the same cluster.
- A pod is allowed outbound traffic to any IP address.

Network policies let you create additional restrictions on what traffic is allowed. For example you may want to restrict external inbound or outbound traffic to certain IP addresses.

For this lab we'll use a network policy to restrict traffic between pods. Let's say that we want to limit access to the `k8sdemo-backend` server to just expose the `k8sdemo` application. First we can observe that the `k8sdemo-backend` server is open to any pod by spinning up a Linux shell.



First let's create a `Pod` that will assist you in testing the reachability of the different elements.

1. **Open a new Terminal Window or Tab** and run (this can take some time):

   ```bash
   kubectl run -it --rm --restart=Never alpine -n default --image=alpine sh

   > If you don't see a command prompt, try pressing enter.
   > / #
   ```

2. Now from **inside** the Pod run the following commands.

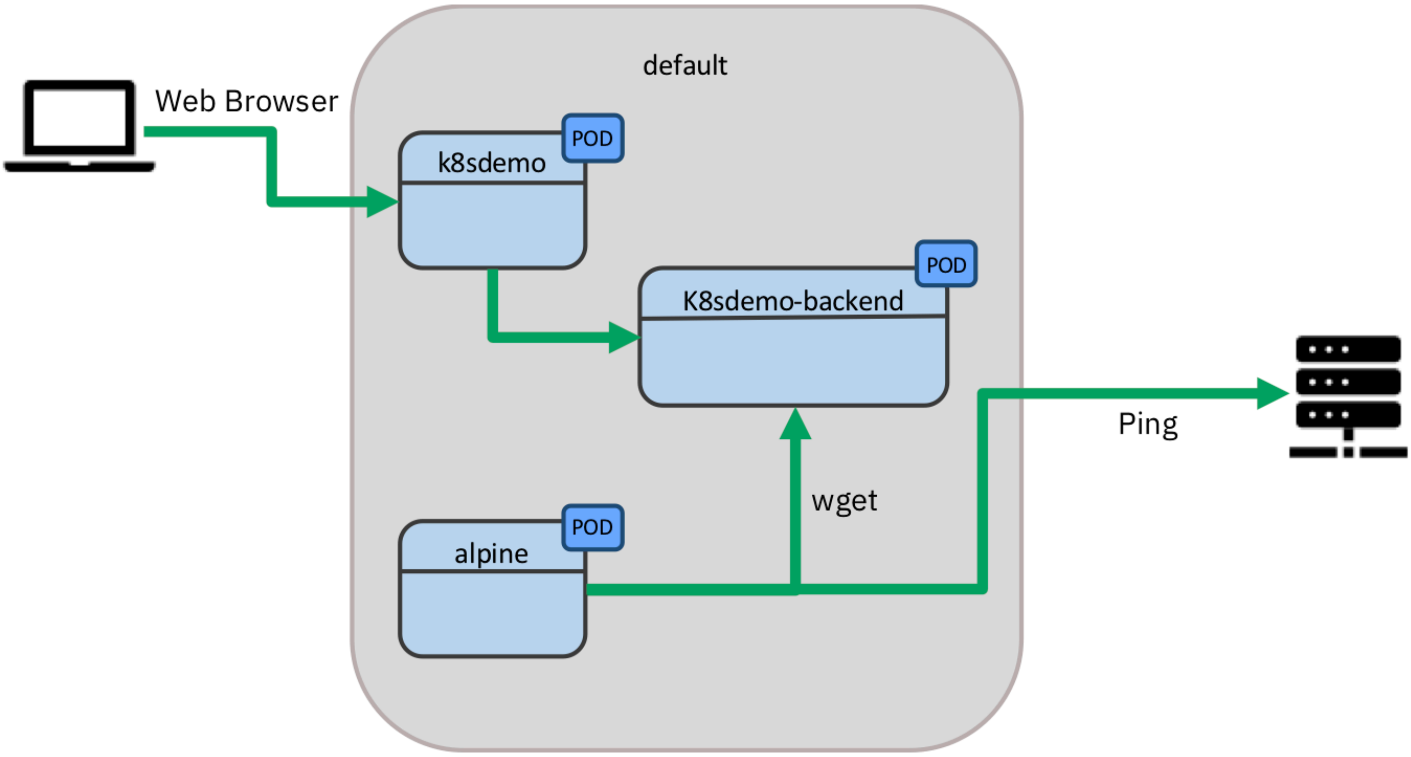
3. The `alpine~> prompt` indicates that the commands must be executed inside the running Pod.

   ```bash
   alpine~> wget -O-  k8sdemo-backend-service.default.svc:3000

   > Connecting to k8sdemo-backend-service.default.svc:3000 (10.103.242.14:3000)
   > K8s Demo Backend                   100% |***********************************
   ***************************|   197  0:00:00 ETA
   ```

   You should get the HTML response from the backend server.

4. And you should be able to ping external adresses (45.55.44.56 is Google)

   ```bash
   alpine~> ping 45.55.44.56

   > PING 45.55.44.56 (45.55.44.56): 56 data bytes
   > 64 bytes from 45.55.44.56: seq=0 ttl=59 time=133.476 ms
   > 64 bytes from 45.55.44.56: seq=1 ttl=59 time=136.036 ms
   > 64 bytes from 45.55.44.56: seq=2 ttl=59 time=125.471 ms
   ```


Web Browser
default
k8sdemo
POD
K8sdemo-backend
POD
Ping
wget
alpine
POD

Lab 1 - Control incoming traffic

Now let's create the first `NetworkPolicy` that simply blocks all traffic coming into all pods.

1. Run the following command

   ```bash
   kubectl create -f ~/training/networkpolicies/deny-all-ingress.yaml
   ```

   This creates the following `NetworkPolicy` which blocks incoming traffic to all Pods.

   ```yaml
   apiVersion: networking.k8s.io/v1
   kind: NetworkPolicy
   metadata:
     name: default-deny-ingress
     namespace: default
   spec:
     podSelector: {}
     policyTypes:
     - Ingress
   ```

2. Now from inside the Pod run:

   ```bash
   alpine~> wget -O-  k8sdemo-backend-service.default.svc:3000

   > Connecting to k8sdemo-backend-service.default.svc:3000 (10.103.242.14:3000)
   ...
   ```

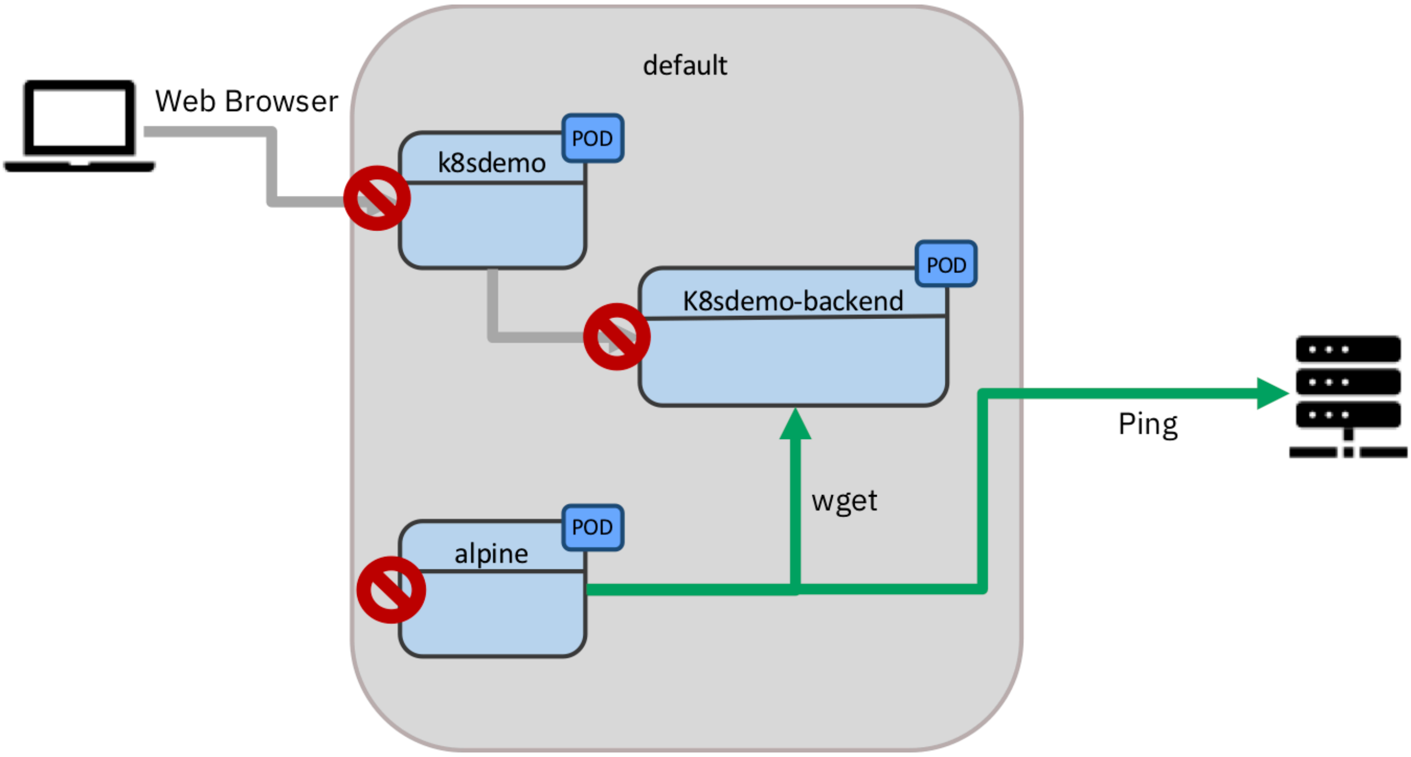
   You should get no response from `k8sdemo-backend`.

3. But you should still be able to ping external adresses

   ```bash
   alpine~> ping 45.55.44.56

   > PING 45.55.44.56 (45.55.44.56): 56 data bytes
   > 64 bytes from 45.55.44.56: seq=0 ttl=59 time=133.476 ms
   > 64 bytes from 45.55.44.56: seq=1 ttl=59 time=136.036 ms
   > 64 bytes from 45.55.44.56: seq=2 ttl=59 time=125.471 ms
   ```

4. Reload the web application. It should not load!

We have just blocked all traffic coming into the pods, but not the outgoing.



Clean-up

Delete the `NetworkPolicy` in order to go back to normal.

```bash
kubectl delete NetworkPolicy -n default default-deny-ingress
```

Lab 1 - Control outgoing traffic

Now let's create a `NetworkPolicy` that simply blocks all outgoing traffic from all pods.

1. Run the following command

   ```bash
   kubectl create -f ~/training/networkpolicies/deny-all-egress.yaml
   ```

   This creates the following `NetworkPolicy` which blocks all outgoing traffic from an Pod.

   ```yaml
   apiVersion: networking.k8s.io/v1
   kind: NetworkPolicy
   metadata:
     name: default-deny-egress
     namespace: default
   spec:
     podSelector: {}
     policyTypes:
     - Egress
   ```

2. Now from inside the Pod run:

   ```bash
   alpine~> wget -O-  k8sdemo-backend-service.default.svc:3000

   > Connecting to k8sdemo-backend-service.default.svc:3000 (10.103.242.14:3000)
   ...
   ```

   You should get no response from the `k8sdemo-backend` as the web frontend `k8sdemo` outgoing traffic is blocked.

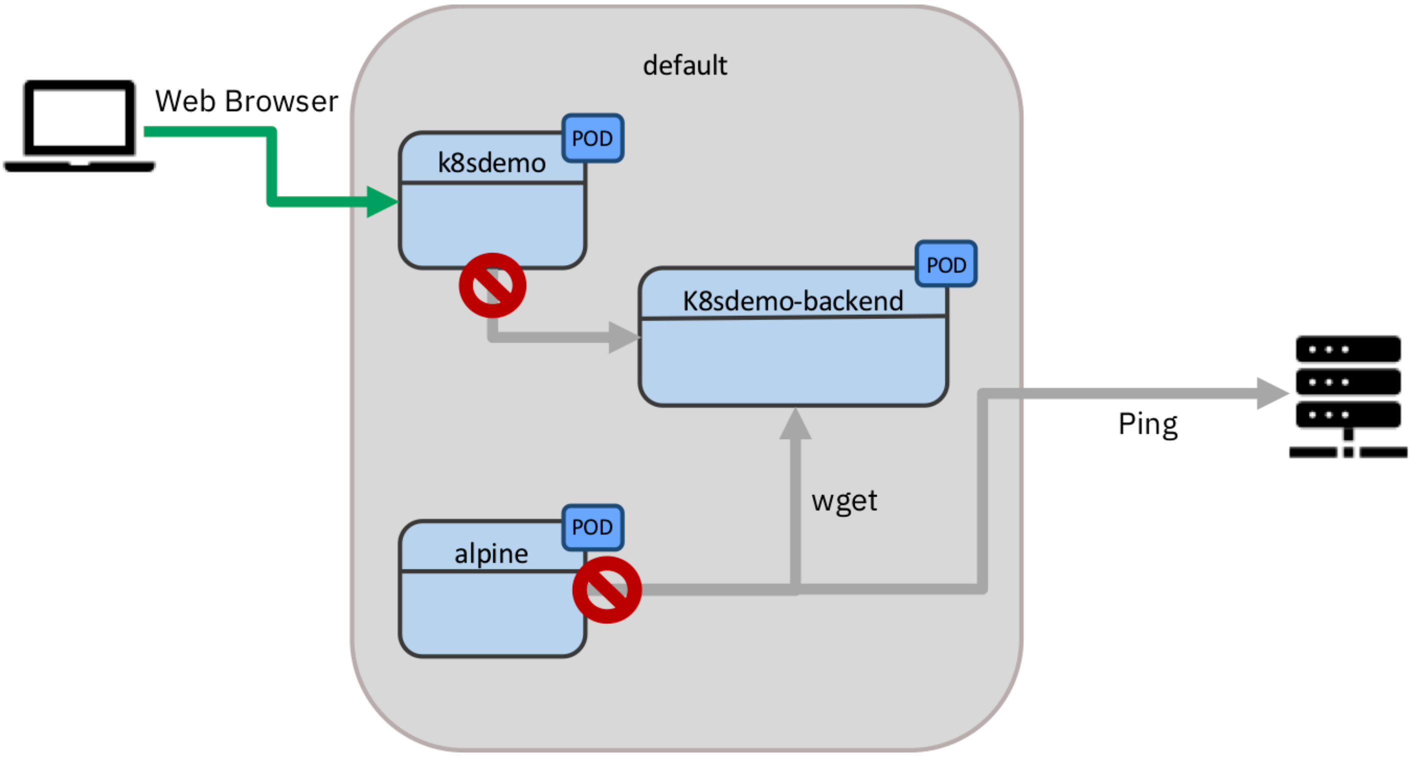
3. And you should not be able to ping external adresses as the `alpine` pod outgoing traffic is blocked.

   ```bash
   alpine~> ping 45.55.44.56

   ...
   ```

4. Reload the web application. It should now load again, but with the error from the backend:

   Testing DEMO_API STATUS: ERROR
   Trying to reach backend ....

We have just blocked all traffic going out of the pods, but not the incoming.



Clean-up

Delete the `NetworkPolicy` in order to go back to normal.

```bash
kubectl delete NetworkPolicy -n default default-deny-egress
```

Lab 1 - Control Pod to Pod communication

Now let's create a `NetworkPolicy` that simply blocks all incoming traffic for the backend (`k8sdemo-backend`) except the one coming from the web frontend (`k8sdemo`).

1. Run the following command

   ```bash
   kubectl create -f ~/training/networkpolicies/deny-except-web.yaml
   ```

   This creates the following `NetworkPolicy` . This policy allows for all traffic, except incoming for the backend Pod.

   ```yaml
   apiVersion: networking.k8s.io/v1
   kind: NetworkPolicy
   metadata:
     name: k8sdemo-web-backend
     namespace: default
   spec:
     podSelector:
       matchLabels:
         app: k8sdemo-backend
     policyTypes:
     - Ingress
     ingress:
     - from:
       - podSelector:
           matchLabels:
             app: k8sdemo
   ```

2. Now from inside the Pod run:

   ```bash
   # wget -O-  k8sdemo-backend-service.default.svc:3000

   > Connecting to k8sdemo-backend-service.default.svc:3000 (10.103.242.14:3000)
   ...
   ```

   You should get no response from `k8sdemo-backend` as only `k8sdemo` is allowed to access it.

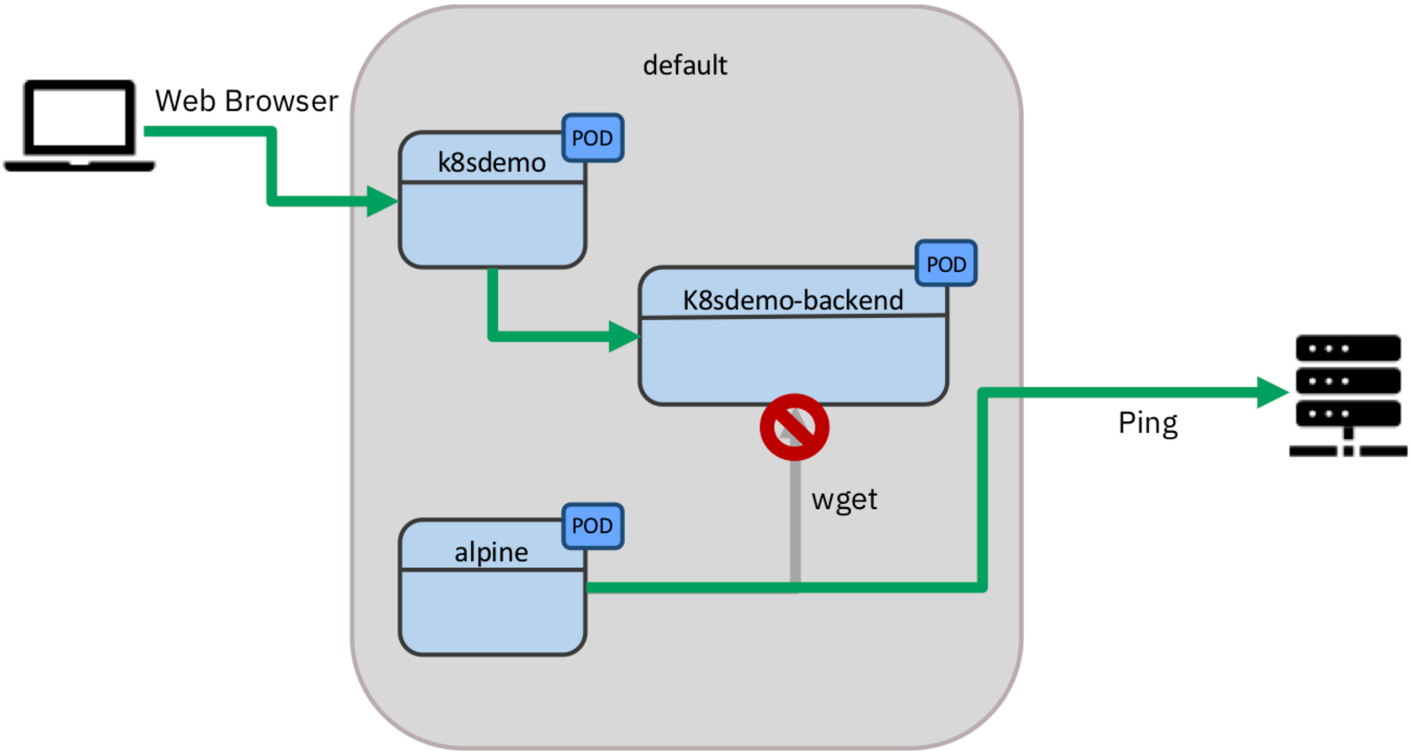
3. You should be able to ping external adresses as outgoing traffic is not blocked.

```bash
# ping 45.55.44.56

> PING 45.55.44.56 (45.55.44.56): 56 data bytes
> 64 bytes from 45.55.44.56: seq=0 ttl=59 time=143.152 ms
> 64 bytes from 45.55.44.56: seq=1 ttl=59 time=120.875 ms
> 64 bytes from 45.55.44.56: seq=2 ttl=59 time=130.981 ms
```

4. Reload the web application. It should now load again, without error from the backend:

   Testing DEMO*API STATUS: OK*
   Message from the Backend The IP Address is <IPADDRESS>

We have just blocked all traffic going to `k8sdemo-backend` , except the one coming from `k8sdemo` thus isolating and securing the communication.



In this Lab we have seen how NetworkPolicies enable us to isolate and control access to and from Pods in the Cluster.

Clean-up

Delete the `NetworkPolicy` in order to go back to normal.

```bash
kubectl delete NetworkPolicy -n default k8sdemo-web-backend
```

Congratulations!!!

This concludes Lab 1 on Network Policies

Lab 2 - Security Tooling

Polaris

Polaris runs a variety of checks to ensure that Kubernetes pods and controllers are configured using best practices, helping you avoid problems in the future.

You can get more details here.

1. Install Polaris Dashboard by running:

```bash
kubectl apply -f ~/training/tools/polaris.yaml

> namespace/polaris created
> configmap/polaris created
> serviceaccount/polaris-dashboard created
> clusterrole.rbac.authorization.k8s.io/polaris-dashboard created
> clusterrolebinding.rbac.authorization.k8s.io/polaris-dashboard created
> service/polaris-dashboard created
> deployment.apps/polaris-dashboard created
```

2. Wait until the pod si running:

```bash
kubectl get pods -n polaris

> NAME                                READY     STATUS     RESTARTS    AGE
> polaris-dashboard-69f5bc4b5d-8jz24  1/1       Running    0           66s
```

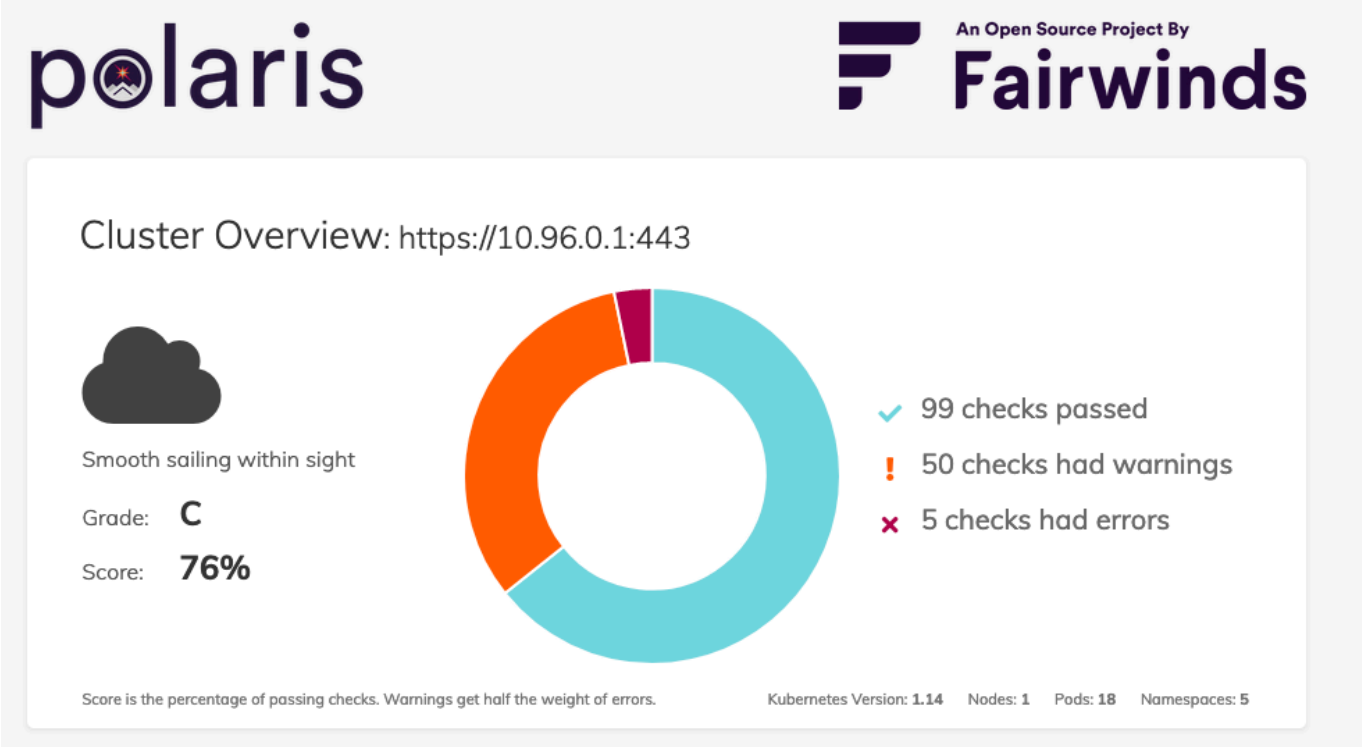
3. Once the status reads `Running`, we need to expose the Dashboard as a service so we can access it:

```bash
kubectl expose deployment polaris-dashboard -n polaris --name polaris-dashboard-service --type="NodePort" --port=8080

> service/polaris-dashboard-service exposed
```

4. The Polaris Dashboard is now running in your cluster, and exposed to the internet. You can open it by typing:

```bash
minikube service polaris-dashboard-service -n polaris
```

5. Look around the Dashboard to get familiar with the checks.



Cluster Overview: https://10.96.0.1:443

Smooth sailing within sight

Grade: C

Score: 76%

✓ 99 checks passed

! 50 checks had warnings

× 5 checks had errors

Score is the percentage of passing checks. Warnings get half the weight of errors.

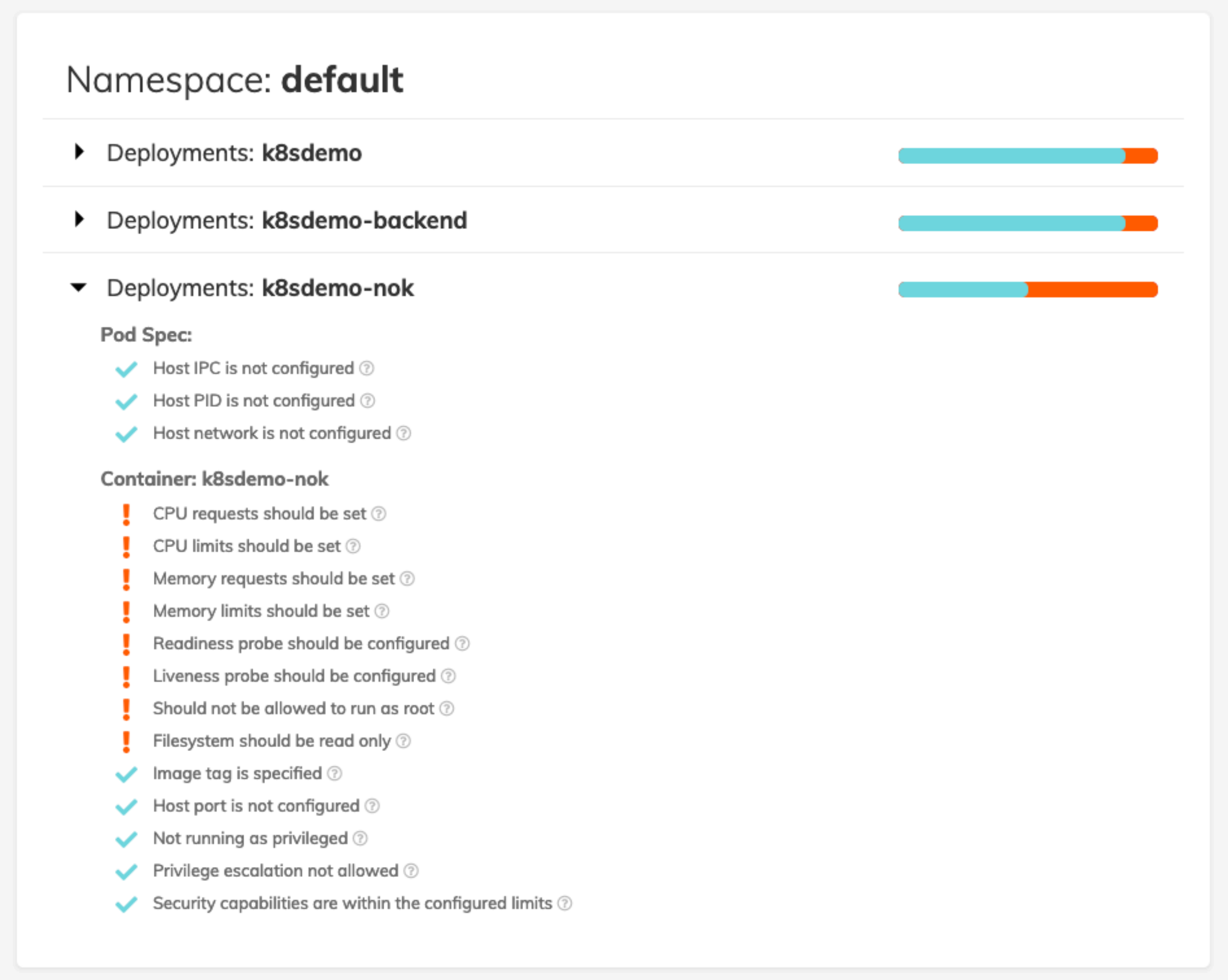
Kubernetes Version: 1.14 Nodes: 1 Pods: 18 Namespaces: 5

6. Let's deploy a version of `k8sdemo` that has some more problems by running:

```bash
kubectl create -f ~/training/deployment/demoapp-errors.yaml
```

This action will take a bit of time. To check the status of the running application, you can use `kubectl get pods`.

7. Check out the dashboard for the `k8sdemo-nok` application and you will find that there are a lot more warnings for this deployment.



Namespace: **default**

- ▸ Deployments: **k8sdemo**
- ▸ Deployments: **k8sdemo-backend**
- ▾ Deployments: **k8sdemo-nok**

Pod Spec:

- ✓ Host IPC is not configured
- ✓ Host PID is not configured
- ✓ Host network is not configured

Container: k8sdemo-nok

- ! CPU requests should be set
- ! CPU limits should be set
- ! Memory requests should be set
- ! Memory limits should be set
- ! Readiness probe should be configured
- ! Liveness probe should be configured
- ! Should not be allowed to run as root
- ! Filesystem should be read only
- ✓ Image tag is specified
- ✓ Host port is not configured
- ✓ Not running as privileged
- ✓ Privilege escalation not allowed
- ✓ Security capabilities are within the configured limits

1. Clean-up by running:

```bash
kubectl delete -f ~/training/deployment/demoapp-errors.yaml
kubectl delete -f ~/training/tools/polaris.yaml
```

Now on to the next tool...

Kube Hunter

Kube-hunter hunts for security weaknesses in Kubernetes clusters. The tool was developed to increase awareness and visibility for security issues in Kubernetes environments.

> **IMPORTANT!!! You should NOT run kube-hunter on a Kubernetes cluster you don't own!**

You can get more details here.

Let's examine the list of passive test (non intrusive, aka that do not change the cluster state) that kube-hunter runs:

```bash
~/kube-hunter/kube-hunter.py --list

> Passive Hunters:
> ----------------
> * Mount Hunter - /var/log
>   Hunt pods that have write access to host's /var/log. in such case, the pod can traverse read files on the host machine
>
> * Host Discovery when running as pod
>   Generates ip adresses to scan, based on cluster/scan type
>
> * API Server Hunter
>   Checks if API server is accessible
>
> * K8s CVE Hunter
>   Checks if Node is running a Kubernetes version vulnerable to specific important CVEs
>
> * Proxy Discovery
>   Checks for the existence of a an open Proxy service
>
> * Pod Capabilities Hunter
>   Checks for default enabled capabilities in a pod
>
> * Kubectl CVE Hunter
>   Checks if the kubectl client is vulnerable to specific important CVEs
...
```

Let's examine the list of passive test (non intrusive, aka that do not change the cluster state) that kube-hunter runs:

Bash

```
~/kube-hunter/kube-hunter.py --list --active

> Passive Hunters:
> ----------------
...

> Active Hunters:
> ---------------
> * Kubelet System Logs Hunter
>   Retrieves commands from host's system audit
>
> * Etcd Remote Access
>   Checks for remote write access to etcd- will attempt to add a new key to the
etcd DB
>
> * Azure SPN Hunter
>   Gets the azure subscription file on the host by executing inside a container
>
> * Kubelet Container Logs Hunter
>   Retrieves logs from a random container
>
> * Kubelet Run Hunter
>   Executes uname inside of a random container
>
...
```

Now let's run an active and passive test against our minikube cluster::

Bash

```
~/kube-hunter/kube-hunter.py  --remote $(minikube ip) --active

> ~ Started
> ~ Discovering Open Kubernetes Services...
> |
> | Etcd:
> |   type: open service
> |   service: Etcd
> |_  location: localhost:2379
> |
> | Kubelet API (readonly):
> |   type: open service
> |   service: Kubelet API (readonly)
> |_  location: localhost:10255
...
```

Findings

You should get no findings, meaning that the Minikube instance has been correctly configured.

If you get vulnerabilities like the following, this might be due to the fact that `minikube` API by default allows for access with user `system:anonymous`.

```
+-----------------+----------------------+----------------------+----------------
------+----------+
| LOCATION        | CATEGORY             | VULNERABILITY        | DESCRIPTION     
      | EVIDENCE |
+-----------------+----------------------+----------------------+----------------
------+----------+
| localhost:10250 | Remote Code          | Anonymous            | The kubelet is  
      |          |
|                 | Execution            | Authentication       | misconfigured,  
      |          |
|                 |                      |                      | potentially all
owing |          |
|                 |                      |                      | secure access t
o all |          |
|                 |                      |                      | requests on the
      |          |
|                 |                      |                      | kubelet, withou
t the |          |
|                 |                      |                      | need to authent
icate |          |
+-----------------+----------------------+----------------------+----------------
------+----------+
```

This should (hopefully!) not be the case in your clusters and in this case could be remediated by launching `minikube` with the option `--extra-config=apiserver.anonymous-auth=false`

kubesec

kubesec is a utility that performs security risk analysis for Kubernetes resources and tells you what you should change in order to improve the security of those pods. It also gives you a score that you can use to create a minimum standard. The score incorporates a great number of Kubernetes best practices.

KubeSec has already been installed in your environment.

1. Launch a test against the demo application

   ```Bash
   kubesec scan ~/training/deployment/demoapp.yaml
   ```

   The output is in JSON format that can easily be integrated into a CI/CD process.

   ```YAML
   [
     {
       "object": "Deployment/k8sdemo.default",
       "valid": true,
       "message": "Passed with a score of 4 points",
       "score": 4,
       "scoring": {
         "advise": [
           {
             "selector": ".metadata .annotations .\"container.apparmor.security.b
   eta.kubernetes.io/nginx\"",
             "reason": "Well defined AppArmor policies may provide greater protec
   tion from unknown threats. WARNING: NOT PRODUCTION READY",
             "points": 3
           },
           {
             "selector": ".spec .serviceAccountName",
             "reason": "Service accounts restrict Kubernetes API access and shoul
   d be configured with least privilege",
             "points": 3
           },
           {
             "selector": ".metadata .annotations .\"container.seccomp.security.al
   pha.kubernetes.io/pod\"",
             "reason": "Seccomp profiles set minimum privilege and secure against
    unknown threats",
             "points": 1
           },
           {
             "selector": "containers[] .securityContext .capabilities .drop",
   ```

```
          "reason": "Reducing kernel capabilities available to a container limits its attack surface",
          "points": 1
        },
        {
          "selector": "containers[] .securityContext .capabilities .drop | index(\"ALL\")",
          "reason": "Drop all capabilities and add only those required to reduce syscall attack surface",
          "points": 1
        },
        {
          "selector": "containers[] .securityContext .readOnlyRootFilesystem == true",
          "reason": "An immutable root filesystem can prevent malicious binaries being added to PATH and increase attack cost",
          "points": 1
        },
        {
          "selector": "containers[] .securityContext .runAsNonRoot == true",
          "reason": "Force the running image to run as a non-root user to ensure least privilege",
          "points": 1
        },
        {
          "selector": "containers[] .securityContext .runAsUser -gt 10000",
          "reason": "Run as a high-UID user to avoid conflicts with the host's user table",
          "points": 1
        }
      ]
    }
  }
]
```

2. Launch a test against a really vulnerable app

```bash
kubesec scan ~/training/deployment/critical.yml
```

The output is in JSON format that can easily be integrated into a CI/CD process.

YAML

```
[
  {
    "object": "Pod/kubesec-test.default",
    "valid": true,
    "message": "Failed with a score of -37 points",
    "score": -37,
    "scoring": {
      "critical": [
        {
          "selector": "containers[] .securityContext .privileged == true",
          "reason": "Privileged containers can allow almost completely unrestricted host access",
          "points": -30
        },
        {
          "selector": "containers[] .securityContext .allowPrivilegeEscalation == true",
          "reason": "",
          "points": -7
        }
      ],
      "advise": [
        {
      ....
```

kubesec gives us a simple tool to check the deployment manifests early on and integrate into a CI/CD process at build-time. KubeSec can run in different ways (commandline, Docker container and even as a Kubernetes admission hook) in order to facilitate that integration.

conftest

conftest is a utility to help you write tests against structured configuration data. For instance you could write tests for your Kubernetes configurations, or Tekton pipeline definitions, Terraform code, Serverless configs or any other structured data.

ConfTest has already been installed in your environment.

1. Launch a test against the demo application

```bash
conftest test -p ~/training/conftest/src/examples/kubernetes/policy ~/training/deployment/demoapp.yaml

> FAIL - /home/training/training/deployment/demoapp.yaml - Containers must not run as root in Deployment k8sdemo
> FAIL - /home/training/training/deployment/demoapp.yaml - Deployment k8sdemo must provide app/release labels for pod selectors
> FAIL - /home/training/training/deployment/demoapp.yaml - k8sdemo must include Kubernetes recommended labels: https://kubernetes.io/docs/concepts/overview/working-with-objects/common-labels/#labels
```

1. Launch a test against all the files for the demo application

The output format here is set to TAP (Test Anything Protocol)

```
conftest test -p ~/training/conftest/src/examples/kubernetes/policy --output=tap
~/training/deployment/*.yaml

> 1..3
> not ok 1 - /home/training/training/deployment/demoapp-backend.yaml - Containers
 must not run as root in Deployment k8sdemo-backend
> not ok 2 - /home/training/training/deployment/demoapp-backend.yaml - Deployment
 k8sdemo-backend must provide app/release labels for pod selectors
> not ok 3 - /home/training/training/deployment/demoapp-backend.yaml - k8sdemo-ba
ckend must include Kubernetes recommended labels: https://kubernetes.io/docs/conc
epts/overview/working-with-objects/common-labels/#labels
> 1..3
> not ok 1 - /home/training/training/deployment/demoapp-errors.yaml - Containers
must not run as root in Deployment k8sdemo-nok
> not ok 2 - /home/training/training/deployment/demoapp-errors.yaml - Deployment
k8sdemo-nok must provide app/release labels for pod selectors
> not ok 3 - /home/training/training/deployment/demoapp-errors.yaml - k8sdemo-nok
 must include Kubernetes recommended labels: https://kubernetes.io/docs/concepts/ove
#labels
> 1..1
> # Warnings
> not ok 1 - /home/training/training/deployment/demoapp-service.yaml - Found serv
ice k8sdemo-service but services are not allowed
> 1..3
> not ok 1 - /home/training/training/deployment/demoapp.yaml - Containers must no
t run as root in Deployment k8sdemo
> not ok 2 - /home/training/training/deployment/demoapp.yaml - Deployment k8sdemo
 must provide app/release labels for pod selectors
> not ok 3 - /home/training/training/deployment/demoapp.yaml - k8sdemo must inclu
de Kubernetes recommended labels: https://kubernetes.io/docs/concepts/overview/wo
rking-with-objects/common-labels/#labels
> 1..1
> # Warnings
> not ok 1 - /home/training/training/deployment/demoapp-backend-service.yaml - Fo
und service k8sdemo-backend-service but services are not allowed
```

1. Launch a test against sample Dockerfile

The output format here is set to JSON.

```
conftest test -p ~/training/conftest/src/examples/docker/policy --output=json ~/t
raining/conftest/src/examples/docker/Dockerfile
```

A blakclisted base image has been detected:

YAML

```yaml
[
    {
        "filename": "/home/training/training/conftest/src/examples/docker/Dockerfile",
        "Warnings": [],
        "Failures": [
            "blacklisted image found [\"openjdk:8-jdk-alpine\"]"
        ],
        "Successes": []
    }
]
```

Congratulations!!!

This concludes Lab 2 on Security Tooling.