CKS Simulator Kubernetes 1.22

https://killer.sh

Pre Setup

Once you've gained access to your terminal it might be wise to spend ~1 minute to setup your environment. You could set these:

```
alias k=kubectl                         # will already be pre-configured

export do="--dry-run=client -o yaml"    # k get pod x $do

export now="--force --grace-period 0"   # k delete pod x $now
```

Vim

To make `vim` use 2 spaces for a tab edit `~/.vimrc` to contain:

```
set tabstop=2
set expandtab
set shiftwidth=2
```

More setup suggestions are in the **tips section**.

Question 1 | Contexts

Task weight: 1%

You have access to multiple clusters from your main terminal through `kubectl` contexts. Write all context names into `/opt/course/1/contexts`, one per line.

From the kubeconfig extract the certificate of user `restricted@infra-prod` and write it decoded to `/opt/course/1/cert`.

Answer:

Maybe the fastest way is just to run:

```
k config get-contexts # copy by hand

k config get-contexts -o name > /opt/course/1/contexts
```

Or using jsonpath:

```
k config view -o jsonpath="{.contexts[*].name}"
k config view -o jsonpath="{.contexts[*].name}" | tr " " "\n" # new lines
k config view -o jsonpath="{.contexts[*].name}" | tr " " "\n" > /opt/course/1/contexts
```

The content could then look like:

```
# /opt/course/1/contexts
gianna@infra-prod
infra-prod
restricted@infra-prod
workload-prod
workload-stage
```

For the certificate we could just run

```
k config view --raw
```

And copy it manually. Or we do:

```
k config view --raw -ojsonpath="{.users[2].user.client-certificate-data}" | base64 -d > /opt/course/1/cert
```

Or even:

```
k config view --raw -ojsonpath="{.users[?(.name == 'restricted@infra-prod')].user.client-certificate-data}" | base64 -d
> /opt/course/1/cert
```

```
# /opt/course/1/cert
-----BEGIN CERTIFICATE-----
MIIDHzCCAgegAwIBAgIQN5Qe/Rj/PhaqckEI23LPnjANBgkqhkiG9w0BAQsFADAV
MRMwEQYDVQQDEwprdWJlcm5ldGVzMB4XDTIwMDkyNjIwNTUwNFoXDTIxMDkyNjIw
NTUwNFowKjETMBEGA1UEChMKcmVzdHJpY3RlZDETMBEGA1UEAxMKcmVzdHJpY3Rl
ZDCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAL/Jaf/QQdijyJTWIDij
qa5p4oAh+xDBX3jR9R0G5DkmPU/FgXjxej3rTwHJbuxg7qjTuqQbf9Fb2AHcVtwH
gUjC12ODUDE+nVtap+hCe8OLHZwH7BGFWWscgInZOZW2IATK/YdqyQL5OKpQpFkx
iAknVZmPa2DTZ8FoyRESboFSTZj6y+JVA7ot0pM09jnxswstal9GZLeqioqfFGY6
YBO/Dg4DDsbKhqfUwJVT6Ur3ELsktZIMTRS5By4Xz18798eBiFAHvgJGq1TTwuPM
EhBfwYwgYbalL8DSHeFrelLBKgciwUKjr1lolnnuc1vhkX1peV1J3xrf6o2KkyMc
lY0CAwEAAaNWMFQwDgYDVR0PAQH/BAQDAgWgMBMGA1UdJQQMMAoGCCsGAQUFBwMC
MAwGA1UdEwEB/wQCMAAwHwYDVR0jBBgwFoAUPrspZIWR7YMN8vT5DF3s/LvpxPQw
DQYJKoZIhvcNAQELBQADggEBAIDq0Zt77gXI1s+uW46zBw4mIWgAlBLl2QqCuwmV
kd86eH5bD0FCtWlb6vGdcKPdFccHh8Z6z2LjjLu6UoiGUdIJaALhbYNJiXXi/7cf
M7sqNOxpxQ5X5hyvOBYD1W7d/EzPHV/lcbXPUDYFHNqBYs842LWSTlPQioDpupXp
FFUQPxsenNXDa4TbmaRvnK2jka0yXcqdiXuIteZZovp/IgNkfmx2Ld4/Q+Xlnscf
CFtWbjRa/0W/3EW/ghQ7xtC7bgcOHJesoiTZPCZ+dfKuUfH6d1qxgj6Jwt0HtyEf
QTQSc66BdMLnw5DMObs4lXDo2YE6LvMrySdXm/S7img5YzU=
-----END CERTIFICATE-----
```

Question 2 | Runtime Security with Falco

Task weight: 4%

Use context: `kubectl config use-context workload-prod`

Falco is installed with default configuration on node `cluster1-worker1`. Connect using `ssh cluster1-worker1`. Use it to:

1. Find a *Pod* running image `nginx` which creates unwanted package management processes inside its container.
2. Find a *Pod* running image `httpd` which modifies `/etc/passwd`.

Save the Falco logs for case 1 under `/opt/course/2/falco.log` in format `time,container-id,container-name,user-name`. No other information should be in any line. Collect the logs for at least 30 seconds.

Afterwards remove the threads (both 1 and 2) by scaling the replicas of the *Deployments* that control the offending *Pods* down to 0.

Answer:

> Falco, the open-source cloud-native runtime security project, is the de facto Kubernetes threat detection engine.

> ***NOTE:*** Other tools you might have to be familar with are sysdig or tracee

Use Falco as service

First we can investigate Falco config a little:

```
➜ ssh cluster1-worker1

➜ root@cluster1-worker1:~# service falco status
● falco.service - LSB: Falco syscall activity monitoring agent
   Loaded: loaded (/etc/init.d/falco; generated)
   Active: active (running) since Sat 2020-10-10 06:36:15 UTC; 2h 1min ago
...

➜ root@cluster1-worker1:~# cd /etc/falco

➜ root@cluster1-worker1:/etc/falco# ls
falco.yaml  falco_rules.local.yaml  falco_rules.yaml  k8s_audit_rules.yaml  rules.available  rules.d
```

This is the default configuration, if we look into `falco.yaml` we can see:

```
# /etc/falco/falco.yaml

...
# Where security notifications should go.
# Multiple outputs can be enabled.

syslog_output:
  enabled: true
...
```

This means that Falco is writing into syslog, hence we can do:

```
➜ root@cluster1-worker1:~# cat /var/log/syslog | grep falco
Sep 15 08:44:04 ubuntu2004 falco: Falco version 0.29.1 (driver version 17f5df52a7d9ed6bb12d3b1768460def8439936d)
Sep 15 08:44:04 ubuntu2004 falco: Falco initialized with configuration file /etc/falco/falco.yaml
Sep 15 08:44:04 ubuntu2004 falco: Loading rules from file /etc/falco/falco_rules.yaml:
...
```

Yep, quite some action going on in there. Let's investigate the first offending *Pod*:

```
➜ root@cluster1-worker1:~# cat /var/log/syslog | grep falco | grep nginx | grep process
Sep 16 06:23:47 ubuntu2004 falco: 06:23:47.376241377: Error Package management process launched in container (user=root
user_loginuid=-1 command=apk container_id=7a5ea6a080d1 container_name=nginx image=docker.io/library/nginx:1.19.2-
alpine)
...

➜ root@cluster1-worker1:~# crictl ps -id 7a5ea6a080d1
CONTAINER ID        IMAGE               NAME        ...          POD ID
7a5ea6a080d1b       6f715d38cfe0e       nginx       ...          7a864406b9794

root@cluster1-worker1:~# crictl pods -id 7a864406b9794
POD ID              ...          NAME                             NAMESPACE        ...
7a864406b9794       ...          webapi-6cfddcd6f4-ftxg4          team-blue        ...
```

First *Pod* is `webapi-6cfddcd6f4-ftxg4` in *Namespace* `team-blue`.

```
➜ root@cluster1-worker1:~# cat /var/log/syslog | grep falco | grep httpd | grep passwd
Sep 16 06:23:48 ubuntu2004 falco: 06:23:48.830962378: Error File below /etc opened for writing (user=root
user_loginuid=-1 command=sed -i $d /etc/passwd parent=sh pcmdline=sh -c echo hacker >> /etc/passwd; sed -i '$d'
/etc/passwd; true file=/etc/passwdngFmAl program=sed gparent=<NA> ggparent=<NA> gggparent=<NA>
container_id=b1339d5cc2de image=docker.io/library/httpd)

➜ root@cluster1-worker1:~# crictl ps -id b1339d5cc2de
CONTAINER ID        IMAGE               NAME        ...          POD ID
b1339d5cc2dee       f6b40f9f8ad71       httpd       ...          595af943c3245

root@cluster1-worker1:~# crictl pods -id 595af943c3245
POD ID              ...          NAME                             NAMESPACE          ...
595af943c3245       ...          rating-service-68cbdf7b7-v2p6g   team-purple        ...
```

Second *Pod* is `rating-service-68cbdf7b7-v2p6g` in *Namespace* `team-purple`.

Eliminate offending *Pods*

The logs from before should allow us to find and "eliminate" the offending *Pods*:

```
➜ k get pod -A | grep webapi
team-blue              webapi-6cfddcd6f4-ftxg4                      1/1     Running

➜ k -n team-blue scale deploy webapi --replicas 0
deployment.apps/webapi scaled

➜ k get pod -A | grep rating-service
team-purple            rating-service-68cbdf7b7-v2p6g               1/1     Running

➜ k -n team-purple scale deploy rating-service --replicas 0
deployment.apps/rating-service scaled
```

Use Falco from command line

We can also use Falco directly from command line, but only if the service is disabled:

```
➜ root@cluster1-worker1:~# service falco stop

➜ root@cluster1-worker1:~# falco
Thu Sep 16 06:33:11 2021: Falco version 0.29.1 (driver version 17f5df52a7d9ed6bb12d3b1768460def8439936d)
Thu Sep 16 06:33:11 2021: Falco initialized with configuration file /etc/falco/falco.yaml
Thu Sep 16 06:33:11 2021: Loading rules from file /etc/falco/falco_rules.yaml:
Thu Sep 16 06:33:11 2021: Loading rules from file /etc/falco/falco_rules.local.yaml:
Thu Sep 16 06:33:11 2021: Loading rules from file /etc/falco/k8s_audit_rules.yaml:
Thu Sep 16 06:33:12 2021: Starting internal webserver, listening on port 8765
06:33:17.382603204: Error Package management process launched in container (user=root user_loginuid=-1 command=apk
container_id=7a5ea6a080d1 container_name=nginx image=docker.io/library/nginx:1.19.2-alpine)
...
```

We can see that rule files are loaded and logs printed afterwards.

Create logs in correct format

The task requires us to store logs for "unwanted package management processes" in format `time,container-id,container-name,user-name`. The output from `falco` shows entries for "Error Package management process launched" in a default format. Let's find the proper file that contains the rule and change it:

```
➜ root@cluster1-worker1:~# cd /etc/falco/

➜ root@cluster1-worker1:/etc/falco# grep -r "Package management process launched" .
./falco_rules.yaml:    Package management process launched in container (user=%user.name user_loginuid=%user.loginuid

➜ root@cluster1-worker1:/etc/falco# cp falco_rules.yaml falco_rules.yaml_ori

➜ root@cluster1-worker1:/etc/falco# vim falco_rules.yaml
```

Find the rule which looks like this:

```
# Container is supposed to be immutable. Package management should be done in building the image.
- rule: Launch Package Management Process in Container
  desc: Package management process ran inside container
  condition: >
    spawned_process
    and container
    and user.name != "_apt"
    and package_mgmt_procs
    and not package_mgmt_ancestor_procs
    and not user_known_package_manager_in_container
  output: >
    Package management process launched in container (user=%user.name user_loginuid=%user.loginuid
    command=%proc.cmdline container_id=%container.id container_name=%container.name
image=%container.image.repository:%container.image.tag)
  priority: ERROR
  tags: [process, mitre_persistence]
```

Should be changed into the required format:

```
# Container is supposed to be immutable. Package management should be done in building the image.
- rule: Launch Package Management Process in Container
  desc: Package management process ran inside container
  condition: >
    spawned_process
    and container
    and user.name != "_apt"
    and package_mgmt_procs
    and not package_mgmt_ancestor_procs
    and not user_known_package_manager_in_container
  output: >
    Package management process launched in container %evt.time,%container.id,%container.name,%user.name
  priority: ERROR
  tags: [process, mitre_persistence]
```

For all available fields we can check https://falco.org/docs/rules/supported-fields, which should be allowed to open during the exam.

Next we check the logs in our adjusted format:

```
➜ root@cluster1-worker1:/etc/falco# falco | grep "Package management"

06:38:28.077150666: Error Package management process launched in container 06:38:28.077150666,090aad374a0a,nginx,root
06:38:33.058263010: Error Package management process launched in container 06:38:33.058263010,090aad374a0a,nginx,root
06:38:38.068693625: Error Package management process launched in container 06:38:38.068693625,090aad374a0a,nginx,root
06:38:43.066159360: Error Package management process launched in container 06:38:43.066159360,090aad374a0a,nginx,root
06:38:48.059792139: Error Package management process launched in container 06:38:48.059792139,090aad374a0a,nginx,root
06:38:53.063328933: Error Package management process launched in container 06:38:53.063328933,090aad374a0a,nginx,root
```

This looks much better. Copy&paste the output into file `/opt/course/2/falco.log` **on your main terminal**. The content should be cleaned like this:

```
# /opt/course/2/falco.log
06:38:28.077150666,090aad374a0a,nginx,root
06:38:33.058263010,090aad374a0a,nginx,root
06:38:38.068693625,090aad374a0a,nginx,root
06:38:43.066159360,090aad374a0a,nginx,root
06:38:48.059792139,090aad374a0a,nginx,root
06:38:53.063328933,090aad374a0a,nginx,root
06:38:58.070912841,090aad374a0a,nginx,root
06:39:03.069592140,090aad374a0a,nginx,root
06:39:08.064805371,090aad374a0a,nginx,root
06:39:13.078109098,090aad374a0a,nginx,root
06:39:18.065077287,090aad374a0a,nginx,root
06:39:23.061012151,090aad374a0a,nginx,root
```

For a few entries it should be fast to just clean it up manually. If there are larger amounts of entries we could do:

```
cat /opt/course/2/falco.log.dirty | cut -d" " -f 9 > /opt/course/2/falco.log
```

The tool `cut` will split input into fields using space as the delimiter (`-d""`). We then only select the 9th field using `-f 9`.

Local falco rules

There is also a file `/etc/falco/falco_rules.local.yaml` in which we can override existing default rules. This is a much cleaner solution for production. Choose the faster way for you in the exam if nothing is specified in the task.

Question 3 | Apiserver Security

Task weight: 3%

Use context: `kubectl config use-context workload-prod`

You received a list from the DevSecOps team which performed a security investigation of the k8s cluster1 (`workload-prod`). The list states the following about the apiserver setup:

- Accessible through a NodePort *Service*

Change the apiserver setup so that:

- Only accessible through a ClusterIP *Service*

Answer:

In order to modify the parameters for the apiserver, we first ssh into the master node and check which parameters the apiserver process is running with:

```
➜ ssh cluster1-master1

➜ root@cluster1-master1:~# ps aux | grep kube-apiserver
root      13534  8.6 18.1 1099208 370684 ?      Ssl  19:55   8:40 kube-apiserver --advertise-address=192.168.100.11 --
allow-privileged=true --anonymous-auth=true --authorization-mode=Node,RBAC --client-ca-file=/etc/kubernetes/pki/ca.crt
--enable-admission-plugins=NodeRestriction --enable-bootstrap-token-auth=true --etcd-
cafile=/etc/kubernetes/pki/etcd/ca.crt --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt --etcd-
keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key --etcd-servers=https://127.0.0.1:2379 --insecure-port=0 --
kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet-client.crt --kubelet-client-
key=/etc/kubernetes/pki/apiserver-kubelet-client.key --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname -
-kubernetes-service-node-port=31000 --proxy-client-cert-file=/etc/kubernetes/pki/front-proxy-client.crt --proxy-client-
key-
...
```

We may notice the following argument:

```
--kubernetes-service-node-port=31000
```

We can also check the *Service* and see its of type NodePort:

```
➜ root@cluster1-master1:~# kubectl get svc
NAME         TYPE       CLUSTER-IP   EXTERNAL-IP   PORT(S)         AGE
kubernetes   NodePort   10.96.0.1    <none>        443:31000/TCP   5d2h
```

The apiserver runs as a static *Pod*, so we can edit the manifest. But before we do this we also create a copy in case we mess things up:

```
➜ root@cluster1-master1:~# cp /etc/kubernetes/manifests/kube-apiserver.yaml ~/3_kube-apiserver.yaml

➜ root@cluster1-master1:~# vim /etc/kubernetes/manifests/kube-apiserver.yaml
```

We should remove the unsecure settings:

```yaml
# /etc/kubernetes/manifests/kube-apiserver.yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubeadm.kubernetes.io/kube-apiserver.advertise-address.endpoint: 192.168.100.11:6443
  creationTimestamp: null
  labels:
    component: kube-apiserver
    tier: control-plane
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    - --advertise-address=192.168.100.11
    - --allow-privileged=true
    - --authorization-mode=Node,RBAC
```

```
    - --client-ca-file=/etc/kubernetes/pki/ca.crt
    - --enable-admission-plugins=NodeRestriction
    - --enable-bootstrap-token-auth=true
    - --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
    - --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt
    - --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key
    - --etcd-servers=https://127.0.0.1:2379
    - --kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet-client.crt
    - --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key
    - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
#     - --kubernetes-service-node-port=31000   # delete or set to 0
    - --proxy-client-cert-file=/etc/kubernetes/pki/front-proxy-client.crt
    - --proxy-client-key-file=/etc/kubernetes/pki/front-proxy-client.key
...
```

Once the changes are made, give the apiserver some time to start up again. Check the apiserver's *Pod* status and the process parameters:

```
➜ root@cluster1-master1:~# kubectl -n kube-system get pod | grep apiserver
kube-apiserver-cluster1-master1            1/1     Running        0          38s

➜ root@cluster1-master1:~# ps aux | grep kube-apiserver | grep node-port
```

The apiserver got restarted without the unsecure settings. However, the *Service* `kubernetes` will still be of type NodePort:

```
➜ root@cluster1-master1:~# kubectl get svc
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)         AGE
kubernetes   NodePort    10.96.0.1    <none>        443:31000/TCP   5d3h
```

We need to delete the *Service* for the changes to take effect:

```
➜ root@cluster1-master1:~# kubectl delete svc kubernetes
service "kubernetes" deleted
```

After a few seconds:

```
➜ root@cluster1-master1:~# kubectl get svc
NAME         TYPE         CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP    10.96.0.1    <none>        443/TCP   6s
```

This should satisfy the DevSecOps team.

Question 4 | Pod Security Policies

Task weight: 8%

Use context: `kubectl config use-context workload-prod`

There is *Deployment* `container-host-hacker` in *Namespace* `team-red` which mounts `/run/containerd` as a hostPath volume on the *Node* where its running. This means that the *Pod* can access various data about other containers running on the same *Node*.

You're asked to forbid this behavior by:

1. Enabling Admission Plugin `PodSecurityPolicy` in the apiserver
2. Creating a *PodSecurityPolicy* named `psp-mount` which allows hostPath volumes only for directory `/tmp`
3. Creating a *ClusterRole* named `psp-mount` which allows to use the new *PSP*
4. Creating a *RoleBinding* named `psp-mount` in *Namespace* `team-red` which binds the new *ClusterRole* to all *ServiceAccounts* in the *Namespace* `team-red`

Restart the *Pod* of *Deployment* `container-host-hacker` afterwards to verify new creation is prevented.

> ***NOTE:*** *PSPs* can affect the whole cluster. Should you encounter issues you can always disable the Admission Plugin again.

Answer:

Investigate

First of all, let's inspect what a *Pod* of *Deployment* `container-host-hacker` is capable of:

```
➜ k -n team-red get pod | grep hacker
container-host-hacker-69b6db5f5d-ldbdh    1/1     Running        0          8m

➜ k -n team-red describe pod container-host-hacker-69b6db5f5d-ldbdh
Name:          container-host-hacker-69b6db5f5d-ldbdh
Namespace:     team-red
Priority:      0
Node:          cluster1-worker2/192.168.100.13
...
```

```
    Mounts:
      /containerdata from containerdata (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-5ms9d (ro)
...
Volumes:
  containerdata:
    Type:          HostPath (bare host directory volume)
    Path:          /run/containerd
...
```

We see it mounts `/run/containerd` from the *Node* where it's running on, what does this mean?

```
➜ k -n team-red exec -it container-host-hacker-69b6db5f5d-ldbdh -- sh

➜ # find /containerdata
...

➜ # find /containerdata grep passwd
...
```

We can see that this *Pod* can access sensitive data from all containers running on the same *Node*. Something that should be prevented unless necessary.

Enable Admission Plugin for *PodSecurityPolicy*

We enable the Admission Plugin and create a config backup in case we misconfigure something:

```
➜ ssh cluster1-master1

➜ root@cluster1-master1:~# cp /etc/kubernetes/manifests/kube-apiserver.yaml ~/4_kube-apiserver.yaml

➜ root@cluster1-master1:~# vim /etc/kubernetes/manifests/kube-apiserver.yaml
```

```yaml
# /etc/kubernetes/manifests/kube-apiserver.yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubeadm.kubernetes.io/kube-apiserver.advertise-address.endpoint: 192.168.100.11:6443
  creationTimestamp: null
  labels:
    component: kube-apiserver
    tier: control-plane
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
    - command:
        - kube-apiserver
        - --advertise-address=192.168.100.11
        - --allow-privileged=true
        - --anonymous-auth=true
        - --authorization-mode=Node,RBAC
        - --client-ca-file=/etc/kubernetes/pki/ca.crt
        - --enable-admission-plugins=NodeRestriction,PodSecurityPolicy      # change
        - --enable-bootstrap-token-auth=true
        - --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
...
```

Existing *PodSecurityPolicy*

Enabling the *PSP* admission plugin without authorizing any policies would prevent any *Pods* from being created in the cluster. That's why there is already an existing *PSP* `default-allow-all` which allows everything and all *Namespaces* except `team-red` use it via a *RoleBinding*:

```
➜ k get psp
NAME                PRIV   CAPS   SELINUX    RUNASUSER   ...
default-allow-all   true   *      RunAsAny   RunAsAny    ...

➜ k get rolebinding -A | grep psp-access
default                psp-access ...  ClusterRole/psp-access
ngress-nginx           psp-access ...  ClusterRole/psp-access
kube-public            psp-access ...  ClusterRole/psp-access
kube-system            psp-access ...  ClusterRole/psp-access
kubernetes-dashboard   psp-access ...  ClusterRole/psp-access
team-blue              psp-access ...  ClusterRole/psp-access
team-green             psp-access ...  ClusterRole/psp-access
team-purple            psp-access ...  ClusterRole/psp-access
team-yellow            psp-access ...  ClusterRole/psp-access
```

Create new *PodSecurityPolicy*

Next we create the new *PSP* with the task requirements by copying an example from the k8s docs and altering it:

```
vim 4_psp.yaml
```

```
# 4_psp.yaml
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: psp-mount
spec:
  privileged: true
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  volumes:
  - '*'
  allowedHostPaths:              # task requirement
    - pathPrefix: "/tmp"         # task requirement
```

```
k -f 4_psp.yaml create
```

So far the *PSP* has no effect because we gave no RBAC permission for any *Pods-ServiceAccounts* to use it yet. So we do:

```
k -n team-red create clusterrole psp-mount --verb=use \
--resource=podsecuritypolicies --resource-name=psp-mount
```

Which will create a *ClusterRole* like:

```
# kubectl -n team-red create clusterrole psp-mount --verb=use --resource=podsecuritypolicies --resource-name=psp-mount
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  creationTimestamp: null
  name: psp-mount
rules:
- apiGroups:
  - policy
  resourceNames:
  - psp-mount
  resources:
  - podsecuritypolicies
  verbs:
  - use
```

And for the *RoleBinding*:

```
k -n team-red create rolebinding psp-mount --clusterrole=psp-mount --group system:serviceaccounts
```

Which will create:

```
# kubectl -n team-red create rolebinding psp-mount --clusterrole=psp-mount --group system:serviceaccounts
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  creationTimestamp: null
  name: psp-mount
  namespace: team-red
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: psp-mount
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:serviceaccounts
```

Test new PSP

We restart the *Deployment* and check the status:

```
➜ k -n team-red rollout restart deploy container-host-hacker
deployment.apps/container-host-hacker restarted

➜ k -n team-red describe deploy container-host-hacker
Name:                   container-host-hacker
Namespace:              team-red
...
Replicas:               1 desired | 0 updated | 1 total | 1 available | 1 unavailable
```

```
...
  Type             Status  Reason
  ----             ------  ------
  Available        True    MinimumReplicasAvailable
  Progressing      True    NewReplicaSetCreated
  ReplicaFailure   True    FailedCreate
```

We see FailedCreate and checking for *Events* shows more information about why:

```
➜ k -n team-red get events --sort-by='{.metadata.creationTimestamp}'
6s          Warning   FailedCreate        replicaset/container-host-hacker-7b4695b5f4   Error creating: pods
"container-host-hacker-7b4695b5f4-" is forbidden: PodSecurityPolicy: unable to admit pod:
[spec.volumes[0].hostPath.pathPrefix: Invalid value: "/run/containerd": is not allowed to be used]
```

Beautiful, the *PSP* seems to work. To verify further we can change the *Deployment*:

```
k -n team-red edit deploy container-host-hacker
```

```
# kubectl -n team-red edit deploy container-host-hacker
apiVersion: apps/v1
kind: Deployment
metadata:
...
spec:
...
  template:
    metadata:
...
    spec:
      containers:
      - command:
        - sh
        - -c
        - while true; do sleep 1d; done
        image: bash
...
        volumeMounts:
        - mountPath: /containerlogs
          name: containerlogs
...
      volumes:
      - hostPath:
          path: /tmp                          # change
          type: ""
```

And we should see it running:

```
➜ k -n team-red get pod -l app=container-host-hacker
NAME                                     READY   STATUS     RESTARTS    AGE
container-host-hacker-5674dbccc9-5lc6q   1/1     Running    0           20s
```

When a *Pod* has been allowed to be created by a *PSP*, then this is shown via an annotation:

```
➜ k -n team-red describe pod -l app=container-host-hacker
...
Annotations:  kubernetes.io/psp: psp-mount
...
```

PodSecurityPolicies can be really hard to come around at first, but once done they're a powerful part in the security tool box. Though they'll be [replaced with something else](https://kubernetes.io/blog/2021/04/06/podsecuritypolicy-deprecation-past-present-and-future) in future K8s releases.

Question 5 | CIS Benchmark

Task weight: 3%

Use context: `kubectl config use-context infra-prod`

You're ask to evaluate specific settings of cluster2 against the CIS Benchmark recommendations. Use the tool `kube-bench` which is already installed on the nodes.

Connect using `ssh cluster2-master1` and `ssh cluster2-worker1`.

On the master node ensure (correct if necessary) that the CIS recommendations are set for:

1. The `--profiling` argument of the kube-controller-manager
2. The ownership of directory `/var/lib/etcd`

On the worker node ensure (correct if necessary) that the CIS recommendations are set for:

3. The permissions of the kubelet configuration `/var/lib/kubelet/config.yaml`
4. The `--client-ca-file` argument of the kubelet

Answer:

Number 1

First we ssh into the master node run `kube-bench` against the master components:

```
➜ ssh cluster2-master1

➜ root@cluster2-master1:~# kube-bench master
...
== Summary ==
41 checks PASS
13 checks FAIL
11 checks WARN
0 checks INFO
```

We see some passes, fails and warnings. Let's check the required task (1) of the controller manager:

```
➜ root@cluster2-master1:~# kube-bench master | grep kube-controller -A 3
1.3.1 Edit the Controller Manager pod specification file /etc/kubernetes/manifests/kube-controller-manager.yaml
on the master node and set the --terminated-pod-gc-threshold to an appropriate threshold,
for example:
--terminated-pod-gc-threshold=10
--
1.3.2 Edit the Controller Manager pod specification file /etc/kubernetes/manifests/kube-controller-manager.yaml
on the master node and set the below parameter.
--profiling=false

1.3.6 Edit the Controller Manager pod specification file /etc/kubernetes/manifests/kube-controller-manager.yaml
on the master node and set the --feature-gates parameter to include RotateKubeletServerCertificate=true.
--feature-gates=RotateKubeletServerCertificate=true
```

There we see 1.3.2 which suggests to set `--profiling=false`, so we obey:

```
➜ root@cluster2-master1:~# vim /etc/kubernetes/manifests/kube-controller-manager.yaml
```

Edit the corresponding line:

```yaml
# /etc/kubernetes/manifests/kube-controller-manager.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    component: kube-controller-manager
    tier: control-plane
  name: kube-controller-manager
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-controller-manager
    - --allocate-node-cidrs=true
    - --authentication-kubeconfig=/etc/kubernetes/controller-manager.conf
    - --authorization-kubeconfig=/etc/kubernetes/controller-manager.conf
    - --bind-address=127.0.0.1
    - --client-ca-file=/etc/kubernetes/pki/ca.crt
    - --cluster-cidr=10.244.0.0/16
    - --cluster-name=kubernetes
    - --cluster-signing-cert-file=/etc/kubernetes/pki/ca.crt
    - --cluster-signing-key-file=/etc/kubernetes/pki/ca.key
    - --controllers=*,bootstrapsigner,tokencleaner
    - --kubeconfig=/etc/kubernetes/controller-manager.conf
    - --leader-elect=true
    - --node-cidr-mask-size=24
    - --port=0
    - --requestheader-client-ca-file=/etc/kubernetes/pki/front-proxy-ca.crt
    - --root-ca-file=/etc/kubernetes/pki/ca.crt
    - --service-account-private-key-file=/etc/kubernetes/pki/sa.key
    - --service-cluster-ip-range=10.96.0.0/12
    - --use-service-account-credentials=true
    - --profiling=false            # add
...
```

We wait for the *Pod* to restart, then run `kube-bench` again to check if the problem was solved:

```
➜ root@cluster2-master1:~# kube-bench master | grep kube-controller -A 3
1.3.1 Edit the Controller Manager pod specification file /etc/kubernetes/manifests/kube-controller-manager.yaml
on the master node and set the --terminated-pod-gc-threshold to an appropriate threshold,
for example:
--terminated-pod-gc-threshold=10
--
1.3.6 Edit the Controller Manager pod specification file /etc/kubernetes/manifests/kube-controller-manager.yaml
on the master node and set the --feature-gates parameter to include RotateKubeletServerCertificate=true.
--feature-gates=RotateKubeletServerCertificate=true
```

Problem solved and 1.3.2 is passing:

```
root@cluster2-master1:~# kube-bench master | grep 1.3.2
[PASS] 1.3.2 Ensure that the --profiling argument is set to false (Scored)
```

Number 2

Next task (2) is to check the ownership of directory `/var/lib/etcd`, so we first have a look:

```
➜ root@cluster2-master1:~# ls -lh /var/lib | grep etcd
drwx------  3 root      root      4.0K Sep 11 20:08 etcd
```

Looks like user root and group root. Also possible to check using:

```
➜ root@cluster2-master1:~# stat -c %U:%G /var/lib/etcd
root:root
```

But what has `kube-bench` to say about this?

```
➜ root@cluster2-master1:~# kube-bench master | grep "/var/lib/etcd" -B5

1.1.12 On the etcd server node, get the etcd data directory, passed as an argument --data-dir,
from the below command:
ps -ef | grep etcd
Run the below command (based on the etcd data directory found above).
For example, chown etcd:etcd /var/lib/etcd
```

To comply we run the following:

```
➜ root@cluster2-master1:~# chown etcd:etcd /var/lib/etcd

➜ root@cluster2-master1:~# ls -lh /var/lib | grep etcd
drwx------  3 etcd      etcd      4.0K Sep 11 20:08 etcd
```

This looks better. We run `kube-bench` again, and make sure test 1.1.12. is passing.

```
➜ root@cluster2-master1:~# kube-bench master | grep 1.1.12
[PASS] 1.1.12 Ensure that the etcd data directory ownership is set to etcd:etcd (Scored)
```

Done.

Number 3

To continue with number (3), we'll head to the worker node and ensure that the kubelet configuration file has the minimum necessary permissions as recommended:

```
➜ ssh cluster2-worker1

➜ root@cluster2-worker1:~# kube-bench node
...
== Summary ==
13 checks PASS
10 checks FAIL
2 checks WARN
0 checks INFO
```

Also here some passes, fails and warnings. We check the permission level of the kubelet config file:

```
➜ root@cluster2-worker1:~# stat -c %a /var/lib/kubelet/config.yaml
777
```

777 is highly permissive access level and not recommended by the `kube-bench` guidelines:

```
➜ root@cluster2-worker1:~# kube-bench node | grep /var/lib/kubelet/config.yaml -B2

2.2.10 Run the following command (using the config file location identified in the Audit step)
chmod 644 /var/lib/kubelet/config.yaml
```

We obey and set the recommended permissions:

```
➜ root@cluster2-worker1:~# chmod 644 /var/lib/kubelet/config.yaml

➜ root@cluster2-worker1:~# stat -c %a /var/lib/kubelet/config.yaml
644
```

And check if test 2.2.10 is passing:

```
➜ root@cluster2-worker1:~# kube-bench node | grep 2.2.10
[PASS] 2.2.10 Ensure that the kubelet configuration file has permissions set to 644 or more restrictive (Scored)
```

Number 4

Finally for number (4), let's check whether `--client-ca-file` argument for the kubelet is set properly according to `kube-bench` recommendations:

```
➜ root@cluster2-worker1:~# kube-bench node | grep client-ca-file
[PASS] 2.1.4 Ensure that the --client-ca-file argument is set as appropriate (Scored)
2.2.7 Run the following command to modify the file permissions of the --client-ca-file
2.2.8 Run the following command to modify the ownership of the --client-ca-file .
```

This looks passing with 2.1.4. The other ones are about the file that the parameter points to and can be ignored here.

To further investigate we run the following command to locate the kubelet config file, and open it:

```
➜ root@cluster2-worker1:~# ps -ef | grep kubelet
root      5157      1  2 20:28 ?        00:03:22 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf --config=/var/lib/kubelet/config.yaml --network-plugin=cni --pod-infra-container-image=k8s.gcr.io/pause:3.2
root     19940 11901  0 22:38 pts/0    00:00:00 grep --color=auto kubelet

➜ root@croot@cluster2-worker1:~# vim /var/lib/kubelet/config.yaml
```

```yaml
# /var/lib/kubelet/config.yaml
apiVersion: kubelet.config.k8s.io/v1beta1
authentication:
  anonymous:
    enabled: false
  webhook:
    cacheTTL: 0s
    enabled: true
  x509:
    clientCAFile: /etc/kubernetes/pki/ca.crt
...
```

The `clientCAFile` points to the location of the certificate, which is correct.

Question 6 | Verify Platform Binaries

Task weight: 2%

(can be solved in any kubectl context)

There are four Kubernetes server binaries located at `/opt/course/6/binaries`. You're provided with the following verified sha512 values for these:

kube-apiserver
`f417c0555bc0167355589dd1afe23be9bf909bf98312b1025f12015d1b58a1c62c9908c0067a7764fa35efdac7016a9efa8711a44425dd6692906a7c283f032c`

kube-controller-manager
`60100cc725e91fe1a949e1b2d0474237844b5862556e25c2c655a33boa8225855ec5ee22fa4927e6c46a60d43a7c4403a27268f96fbb726307d1608b44f38a60`

kube-proxy
`52f9d8ad045f8eee1d689619ef8ceef2d86d50c75a6a332653240d7ba5b2a114aca056d9e513984ade24358c9662714973c1960c62a5cb37dd375631c8a614c6`

kubelet
`4be40f2440619e990897cf956c32800dc96c2c983bf64519854a3309fa5aa21827991559f9c44595098e27e6f2ee4d64a3fdec6baba8a177881f20e3ec61e26c`

Delete those binaries that don't match with the sha512 values above.

Answer:

We check the directory:

```
➜ cd /opt/course/6/binaries

➜ ls
kube-apiserver  kube-controller-manager  kube-proxy  kubelet
```

To generate the sha512 sum of a binary we do:

```
➜ sha512sum kube-apiserver
f417c0555bc0167355589dd1afe23be9bf909bf98312b1025f12015d1b58a1c62c9908c0067a7764fa35efdac7016a9efa8711a44425dd6692906a7
c283f032c  kube-apiserver
```

Looking good, next:

```
➜ sha512sum kube-controller-manager
60100cc725e91fe1a949e1b2d0474237844b5862556e25c2c655a33b0a8225855ec5ee22fa4927e6c46a60d43a7c4403a27268f96fbb726307d1608
b44f38a60  kube-controller-manager
```

Okay, next:

```
➜ sha512sum kube-proxy
52f9d8ad045f8eee1d689619ef8ceef2d86d50c75a6a332653240d7ba5b2a114aca056d9e513984ade24358c9662714973c1960c62a5cb37dd37563
1c8a614c6  kube-proxy
```

Also good, and finally:

```
➜ sha512sum kubelet
7b720598e6a3483b45c537b57d759e3e82bc5c53b3274f681792f62e941019cde3d51a7f9b55158abf3810d506146bc0aa7cf97b36f27f341028a54
431b335be  kubelet
```

Catch! Binary `kubelet` has a different hash!

But did we actually compare everything properly before? Let's have a closer look at `kube-controller-manager` again:

```
➜ sha512sum kube-controller-manager > compare

➜ vim compare
```

Edit to only have the provided hash and the generated one in one line each:

```
# ./compare
60100cc725e91fe1a949e1b2d0474237844b5862556e25c2c655a33b0a8225855ec5ee22fa4927e6c46a60d43a7c4403a27268f96fbb726307d1608
b44f38a60
60100cc725e91fe1a949e1b2d0474237844b5862556e25c2c655a33boa8225855ec5ee22fa4927e6c46a60d43a7c4403a27268f96fbb726307d1608
b44f38a60
```

Looks right at a first glance, but if we do:

```
➜ cat compare | uniq
60100cc725e91fe1a949e1b2d0474237844b5862556e25c2c655a33b0a8225855ec5ee22fa4927e6c46a60d43a7c4403a27268f96fbb726307d1608
b44f38a60
60100cc725e91fe1a949e1b2d0474237844b5862556e25c2c655a33boa8225855ec5ee22fa4927e6c46a60d43a7c4403a27268f96fbb726307d1608
b44f38a60
```

This shows they are different, by just one character actually.

To complete the task we do:

```
rm kubelet kube-controller-manager
```

Question 7 | Open Policy Agent

Task weight: 6%

Use context: `kubectl config use-context infra-prod`

The Open Policy Agent and Gatekeeper have been installed to, among other things, enforce blacklisting of certain image registries. Alter the existing constraint and/or template to also blacklist images from `very-bad-registry.com`.

Test it by creating a single *Pod* using image `very-bad-registry.com/image` in *Namespace* `default`, it shouldn't work.

You can also verify your changes by looking at the existing *Deployment* `untrusted` in *Namespace* `default`, it uses an image from the new untrusted source. The OPA contraint should throw violation messages for this one.

Answer:

We look at existing OPA constraints, these are implemeted using CRDs by Gatekeeper:

```
➜ k get crd
NAME                                                 CREATED AT
blacklistimages.constraints.gatekeeper.sh            2020-09-14T19:29:31Z
configs.config.gatekeeper.sh                         2020-09-14T19:29:04Z
constraintpodstatuses.status.gatekeeper.sh           2020-09-14T19:29:05Z
constrainttemplatepodstatuses.status.gatekeeper.sh   2020-09-14T19:29:05Z
constrainttemplates.templates.gatekeeper.sh          2020-09-14T19:29:05Z
requiredlabels.constraints.gatekeeper.sh             2020-09-14T19:29:31Z
```

So we can do:

```
➜ k get constraint
NAME                                                          AGE
blacklistimages.constraints.gatekeeper.sh/pod-trusted-images   10m

NAME                                                                  AGE
requiredlabels.constraints.gatekeeper.sh/namespace-mandatory-labels   10m
```

and then look at the one that is probably about blacklisting images:

```
k edit blacklistimages pod-trusted-images
```

```
# kubectl edit blacklistimages pod-trusted-images
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: BlacklistImages
metadata:
...
spec:
  match:
    kinds:
    - apiGroups:
      - ""
      kinds:
      - Pod
```

It looks like this constraint simply applies the template to all *Pods*, no arguments passed. So we edit the template:

```
k edit constrainttemplates blacklistimages
```

```
# kubectl edit constrainttemplates blacklistimages
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
...
spec:
  crd:
    spec:
      names:
        kind: BlacklistImages
  targets:
  - rego: |
      package k8strustedimages

      images {
        image := input.review.object.spec.containers[_].image
        not startswith(image, "docker-fake.io/")
        not startswith(image, "google-gcr-fake.com/")
        not startswith(image, "very-bad-registry.com/") # ADD THIS LINE
      }

      violation[{"msg": msg}] {
        not images
        msg := "not trusted image!"
      }
    target: admission.k8s.gatekeeper.sh
```

We simply have to add another line. After editing we try to create a *Pod* of the bad image:

```
➜ k run opa-test --image=very-bad-registry.com/image
Error from server ([denied by pod-trusted-images] not trusted image!): admission webhook "validation.gatekeeper.sh"
denied the request: [denied by pod-trusted-images] not trusted image!
```

Nice! After some time we can also see that *Pods* of the existing *Deployment* "untrusted" will be listed as violators:

```
➜ k describe blacklistimages pod-trusted-images
...
  Total Violations:  2
  Violations:
    Enforcement Action:  deny
    Kind:                Namespace
    Message:             you must provide labels: {"security-level"}
    Name:                sidecar-injector
    Enforcement Action:  deny
    Kind:                Pod
    Message:             not trusted image!
    Name:                untrusted-68c4944d48-tfsnb
    Namespace:           default
Events:                  <none>
```

Great, OPA fights bad registries !

Question 8 | Secure Kubernetes Dashboard

Task weight: 3%

Use context: `kubectl config use-context workload-prod`

The Kubernetes Dashboard is installed in *Namespace* `kubernetes-dashboard` and is configured to:

1. Allow users to "skip login"
2. Allow insecure access (HTTP without authentication)
3. Allow basic authentication
4. Allow access from outside the cluster

You are asked to make it more secure by:

1. Deny users to "skip login"
2. Deny insecure access, enforce HTTPS (self signed certificates are ok for now)
3. Add the `--auto-generate-certificates` argument
4. Enforce authentication using a token (with possibility to use RBAC)
5. Allow only cluster internal access

Answer:

Head to https://github.com/kubernetes/dashboard/tree/master/docs to find documentation about the dashboard.

First we have a look in *Namespace* `kubernetes-dashboard`:

```
➜ k -n kubernetes-dashboard get pod,svc
NAME                                             READY   STATUS    RESTARTS   AGE
pod/dashboard-metrics-scraper-7b59f7d4df-fbpd9   1/1     Running   0          24m
pod/kubernetes-dashboard-6d8cd5dd84-w7wr2        1/1     Running   0          24m

NAME                                TYPE        ...   PORT(S)                        AGE
service/dashboard-metrics-scraper   ClusterIP   ...   8000/TCP                       24m
service/kubernetes-dashboard        NodePort    ...   9090:32520/TCP,443:31206/TCP   24m
```

We can see one running *Pod* and a NodePort *Service* exposing it. Let's try to connect to it via a NodePort, we can use IP of any *Node*:

(your port might be a different)

```
➜ k get node -o wide
NAME               STATUS   ROLES    AGE   VERSION   INTERNAL-IP     ...
cluster1-master1   Ready    master   37m   v1.22.1   192.168.100.11 ...
cluster1-worker1   Ready    <none>   36m   v1.22.1   192.168.100.12 ...
cluster1-worker2   Ready    <none>   34m   v1.22.1   192.168.100.13 ...

➜ curl http://192.168.100.11:32520
<!--
Copyright 2017 The Kubernetes Authors.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at
```

```
    http://www.apache.org/licenses/LICENSE-2.0
```

The dashboard is not secured because it allows unsecure HTTP access without authentication and is exposed externally. It's is loaded with a few parameter making it insecure, let's fix this.

First we create a backup in case we need to undo something:

```
k -n kubernetes-dashboard get deploy kubernetes-dashboard -oyaml > 8_deploy_kubernetes-dashboard.yaml
```

Then:

```
k -n kubernetes-dashboard edit deploy kubernetes-dashboard
```

The changes to make are :

```
  template:
    spec:
      containers:
      - args:
        - --namespace=kubernetes-dashboard
        - --authentication-mode=token        # change or delete, "token" is default
        - --auto-generate-certificates       # add
        #- --enable-skip-login=true          # delete or set to false
        #- --enable-insecure-login           # delete
        image: kubernetesui/dashboard:v2.0.3
        imagePullPolicy: Always
        name: kubernetes-dashboard
```

Next, we'll have to deal with the NodePort *Service*:

```
k -n kubernetes-dashboard get svc kubernetes-dashboard -o yaml > 8_svc_kubernetes-dashboard.yaml # backup

k -n kubernetes-dashboard edit svc kubernetes-dashboard
```

And make the following changes:

```
 spec:
  clusterIP: 10.107.176.19
  externalTrafficPolicy: Cluster
  ports:
  - name: http
    nodePort: 32513  # delete
    port: 9090
    protocol: TCP
    targetPort: 9090
  - name: https
    nodePort: 32441  # delete
    port: 443
    protocol: TCP
    targetPort: 8443
  selector:
    k8s-app: kubernetes-dashboard
  sessionAffinity: None
  type: ClusterIP          # change or delete
status:
  loadBalancer: {}
```

Let's confirm the changes, we can do that even without having a browser:

```
➜ k run tmp --image=nginx:1.19.2 --restart=Never --rm -it -- bash
If you don't see a command prompt, try pressing enter.
root@tmp:/# curl http://kubernetes-dashboard.kubernetes-dashboard:9090
curl: (7) Failed to connect to kubernetes-dashboard.kubernetes-dashboard port 9090: Connection refused

➜ root@tmp:/# curl https://kubernetes-dashboard.kubernetes-dashboard
curl: (60) SSL certificate problem: self signed certificate
More details here: https://curl.haxx.se/docs/sslcerts.html

curl failed to verify the legitimacy of the server and therefore could not
establish a secure connection to it. To learn more about this situation and
how to fix it, please visit the web page mentioned above.

➜ root@tmp:/# curl https://kubernetes-dashboard.kubernetes-dashboard -k
<!--
Copyright 2017 The Kubernetes Authors.
```

We see that insecure access is disabled and HTTPS works (using a self signed certificate for now). Let's also check the remote access:

(your port might be a different)

```
➜ curl http://192.168.100.11:32520
curl: (7) Failed to connect to 192.168.100.11 port 32520: Connection refused

➜ k -n kubernetes-dashboard get svc
NAME                        TYPE        CLUSTER-IP       ...   PORT(S)
dashboard-metrics-scraper   ClusterIP   10.111.171.247   ...   8000/TCP
kubernetes-dashboard        ClusterIP   10.100.118.128   ...   9090/TCP,443/TCP
```

Much better.

Question 9 | AppArmor Profile

Task weight: 3%

Use context: `kubectl config use-context workload-prod`

Some containers need to run more secure and restricted. There is an existing AppArmor profile located at `/opt/course/9/profile` for this.

1. Install the AppArmor profile on *Node* `cluster1-worker1`. Connect using `ssh cluster1-worker1`.
2. Add label `security=apparmor` to the *Node*
3. Create a *Deployment* named `apparmor` in *Namespace* `default` with:
   - One replica of image `nginx:1.19.2`
   - NodeSelector for `security=apparmor`
   - Single container named `c1` with the AppArmor profile enabled

   The *Pod* might not run properly with the profile enabled. Write the logs of the *Pod* into `/opt/course/9/logs` so another team can work on getting the application running.

Answer:

https://kubernetes.io/docs/tutorials/clusters/apparmor

Part 1

First we have a look at the provided profile:

```
vim /opt/course/9/profile
```

```
# /opt/course/9/profile

#include <tunables/global>

profile very-secure flags=(attach_disconnected) {
  #include <abstractions/base>

  file,

  # Deny all file writes.
  deny /** w,
}
```

Very simple profile named `very-secure` which denies all file writes. Next we copy it onto the *Node*:

```
➜ scp /opt/course/9/profile cluster1-worker1:~/
Warning: Permanently added the ECDSA host key for IP address '192.168.100.12' to the list of known hosts.
profile                                                                    100%  161   329.9KB/s   00:00

➜ ssh cluster1-worker1

➜ root@cluster1-worker1:~# ls
profile
```

And install it:

```
➜ root@cluster1-worker1:~# apparmor_parser -q ./profile
```

Verify it has been installed:

```
➜ root@cluster1-worker1:~# apparmor_status
apparmor module is loaded.
17 profiles are loaded.
17 profiles are in enforce mode.
```

```
   /sbin/dhclient
...
   man_filter
   man_groff
   very-secure
0 profiles are in complain mode.
56 processes have profiles defined.
56 processes are in enforce mode.
...
0 processes are in complain mode.
0 processes are unconfined but have a profile defined.
```

There we see among many others the `very-secure` one, which is the name of the profile specified in `/opt/course/9/profile`.

Part 2

We label the *Node*:

```
k label -h # show examples

k label node cluster1-worker1 security=apparmor
```

Part 3

Now we can go ahead and create the *Deployment* which uses the profile.

```
k create deploy apparmor --image=nginx:1.19.2 $do > 9_deploy.yaml

vim 9_deploy.yaml
```

```
# 9_deploy.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: apparmor
  name: apparmor
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: apparmor
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: apparmor
      annotations:                                                                # add
        container.apparmor.security.beta.kubernetes.io/c1: localhost/very-secure   # add
    spec:
      nodeSelector:                     # add
        security: apparmor              # add
      containers:
      - image: nginx:1.19.2
        name: c1                        # change
        resources: {}
```

```
k -f 9_deploy.yaml create
```

What the damage?

```
➜ k get pod -owide | grep apparmor
apparmor-85c65645dc-jbch8     0/1     CrashLoopBackOff  ...   cluster1-worker1

➜ k logs apparmor-85c65645dc-w852p
/docker-entrypoint.sh: 13: /docker-entrypoint.sh: cannot create /dev/null: Permission denied
/docker-entrypoint.sh: No files found in /docker-entrypoint.d/, skipping configuration
2021/09/15 11:51:57 [emerg] 1#1: mkdir() "/var/cache/nginx/client_temp" failed (13: Permission denied)
nginx: [emerg] mkdir() "/var/cache/nginx/client_temp" failed (13: Permission denied)
```

This looks alright, the *Pod* is running on `cluster1-worker1` because of the nodeSelector. The AppArmor profile simply denies all filesystem writes, but Nginx needs to write into some locations to run, hence the errors.

It looks like our profile is running but we can confirm this as well by inspecting the container:

```
➜ ssh cluster1-worker1

➜ root@cluster1-worker1:~# crictl pods | grep apparmor
be5c0aecee7c7       4 minutes ago       Ready               apparmor-85c65645dc-jbch8   ...

➜ root@cluster1-worker1:~# crictl ps -a | grep be5c0aecee7c7
e4d91cbdf72fb    ...  Exited       c1           6            be5c0aecee7c7

➜ root@cluster1-worker1:~# crictl inspect e4d91cbdf72fb | grep -i profile
          "apparmor_profile": "localhost/very-secure",
        "apparmorProfile": "very-secure",
```

First we find the *Pod* by it's name and get the pod-id. Next we use `crictl ps -a` to also show stopped containers. Then `crictl inspect` shows that the container is using our AppArmor profile. Notice to be fast between `ps` and `inspect` as K8s will restart the *Pod* periodically when in error state.

To complete the task we write the logs into the required location:

```
k logs apparmor-85c65645dc-jbch8 > /opt/course/9/logs
```

Fixing the errors is the job of another team, lucky us.

Question 10 | Container Runtime Sandbox gVisor

Task weight: 4%

Use context: `kubectl config use-context workload-prod`

Team purple wants to run some of their workloads more secure. Worker node `cluster1-worker2` has container engine containerd already installed and its configured to support the runsc/gvisor runtime.

Create a *RuntimeClass* named `gvisor` with handler `runsc`.

Create a *Pod* that uses the *RuntimeClass*. The *Pod* should be in *Namespace* `team-purple`, named `gvisor-test` and of image `nginx:1.19.2`. Make sure the *Pod* runs on `cluster1-worker2`.

Write the `dmesg` output of the successfully started *Pod* into `/opt/course/10/gvisor-test-dmesg`.

Answer:

We check the nodes and we can see that all are using containerd:

```
➜ k get node -o wide
NAME                STATUS   ROLES                     ... CONTAINER-RUNTIME
cluster1-master1    Ready    control-plane,master      ... containerd://1.5.2
cluster1-worker1    Ready    <none>                    ... containerd://1.5.2
cluster1-worker2    Ready    <none>                    ... containerd://1.5.2
```

But just one has containerd configured to work with runsc/gvisor runtime which is `container1-worker2`.

(Optionally) we ssh into the worker node and check if containerd+runsc is configured:

```
➜ ssh cluster1-worker2

➜ root@cluster1-worker2:~# runsc --version
runsc version release-20201130.0
spec: 1.0.1-dev

➜ root@cluster1-worker2:~# cat /etc/containerd/config.toml | grep runsc
  [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runsc]
    runtime_type = "io.containerd.runsc.v1"
```

Now we best head to the k8s docs for *RuntimeClasses* https://kubernetes.io/docs/concepts/containers/runtime-class, steal an example and create the gvisor one:

```
vim 10_rtc.yaml
```

```yaml
# 10_rtc.yaml
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  name: gvisor
handler: runsc
```

```
k -f 10_rtc.yaml create
```

And the required *Pod*:

```
k -n team-purple run gvisor-test --image=nginx:1.19.2 $do > 10_pod.yaml

vim 10_pod.yaml
```

```yaml
# 10_pod.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: gvisor-test
  name: gvisor-test
  namespace: team-purple
spec:
  nodeName: cluster1-worker2 # add
  runtimeClassName: gvisor   # add
  containers:
  - image: nginx:1.19.2
    name: gvisor-test
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

```
k -f 10_pod.yaml create
```

After creating the pod we should check if its running and if it uses the gvisor sandbox:

```
➜ k -n team-purple get pod gvisor-test
NAME          READY   STATUS    RESTARTS   AGE
gvisor-test   1/1     Running   0          30s

➜ k -n team-purple exec gvisor-test -- dmesg
[    0.000000] Starting gVisor...
[    0.417740] Checking naughty and nice process list...
[    0.623721] Waiting for children...
[    0.902192] Gathering forks...
[    1.258087] Committing treasure map to memory...
[    1.653149] Generating random numbers by fair dice roll...
[    1.918386] Creating cloned children...
[    2.137450] Digging up root...
[    2.369841] Forking spaghetti code...
[    2.840216] Rewriting operating system in Javascript...
[    2.956226] Creating bureaucratic processes...
[    3.329981] Ready!
```

Looking good. And as required we finally write the `dmesg` output into the file:

```
k -n team-purple exec gvisor-test > /opt/course/10/gvisor-test-dmesg -- dmesg
```

Question 11 | Secrets in ETCD

Task weight: 7%

Use context: `kubectl config use-context workload-prod`

There is an existing *Secret* called `database-access` in *Namespace* `team-green`.

Read the complete *Secret* content directly from ETCD (using `etcdctl`) and store it into `/opt/course/11/etcd-secret-content`. Write the plain and decoded *Secret's* value of key "pass" into `/opt/course/11/database-password`.

Answer:

Let's try to get the *Secret* value directly from ETCD, which will work since it isn't encrypted.

First, we ssh into the master node where ETCD is running in this setup and check if `etcdctl` is installed and list its options:

```
➜ ssh cluster1-master1

➜ root@cluster1-master1:~# etcdctl
NAME:
```

```
   etcdctl - A simple command line client for etcd.

WARNING:
   Environment variable ETCDCTL_API is not set; defaults to etcdctl v2.
   Set environment variable ETCDCTL_API=3 to use v3 API or ETCDCTL_API=2 to use v2 API.

USAGE:
   etcdctl [global options] command [command options] [arguments...]
...
   --cert-file value   identify HTTPS client using this SSL certificate file
   --key-file value    identify HTTPS client using this SSL key file
   --ca-file value     verify certificates of HTTPS-enabled servers using this CA bundle
...
```

Among others we see arguments to identify ourselves. The apiserver connects to ETCD, so we can run the following command to get the path of the necessary .crt and .key files:

```
cat /etc/kubernetes/manifests/kube-apiserver.yaml | grep etcd
```

The output is as follows :

```
    - --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
    - --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt
    - --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key
    - --etcd-servers=https://127.0.0.1:2379 # optional since we're on same node
```

With this information we query ETCD for the secret value:

```
➜ root@cluster1-master1:~# ETCDCTL_API=3 etcdctl \
--cert /etc/kubernetes/pki/apiserver-etcd-client.crt \
--key /etc/kubernetes/pki/apiserver-etcd-client.key \
--cacert /etc/kubernetes/pki/etcd/ca.crt get /registry/secrets/team-green/database-access
```

ETCD in Kubernetes stores data under `/registry/{type}/{namespace}/{name}`. This is how we came to look for `/registry/secrets/team-green/database-access`. There is also an example on a page in the k8s documentation which you could save as a bookmark to access fast during the exam.

The tasks requires us to store the output on our terminal. For this we can simply copy&paste the content into a new file on our terminal:

```
# /opt/course/11/etcd-secret-content
/registry/secrets/team-green/database-access
k8s


v1Secret

database-access
team-green"*$3e0acd78-709d-4f07-bdac-d5193d0f2aa32bB
0kubectl.kubernetes.io/last-applied-configuration{"apiVersion":"v1","data":
{"pass":"Y29uZmlkZW50aWFs"},"kind":"Secret","metadata":{"annotations":{},"name":"database-access","namespace":"team-
green"}}
z
kubectl-client-side-applyUpdatevFieldsV1:
{"f:data":{".":{},"f:pass":{}},"f:metadata":{"f:annotations":{".":{},"f:kubectl.kubernetes.io/last-applied-
configuration":{}}},"f:type":{}}
pass
    confidentialOpaque"
```

We're also required to store the plain and "decrypted" database password. For this we can copy the base64-encoded value from the ETCD output and run on our terminal:

```
➜ echo Y29uZmlkZW50aWFs | base64 -d > /opt/course/11/database-password

➜ cat /opt/course/11/database-password
confidential
```

Question 12 | Hack Secrets

Task weight: 8%

Use context: `kubectl config use-context restricted@infra-prod`

You're asked to investigate a possible permission escape in *Namespace* `restricted`. The context authenticates as user `restricted` which has only limited permissions and shouldn't be able to read *Secret* values.

Try to find the password-key values of the *Secrets* `secret1`, `secret2` and `secret3` in *Namespace* `restricted`. Write the decoded plaintext values into files `/opt/course/12/secret1`, `/opt/course/12/secret2` and `/opt/course/12/secret3`.

Answer:

First we should explore the boundaries, we can try:

```
➜ k -n restricted get role,rolebinding,clusterrole,clusterrolebinding
Error from server (Forbidden): roles.rbac.authorization.k8s.io is forbidden: User "restricted" cannot list resource "roles" in API group "rbac.authorization.k8s.io" in the namespace "restricted"
Error from server (Forbidden): rolebindings.rbac.authorization.k8s.io is forbidden: User "restricted" cannot list resource "rolebindings" in API group "rbac.authorization.k8s.io" in the namespace "restricted"
Error from server (Forbidden): clusterroles.rbac.authorization.k8s.io is forbidden: User "restricted" cannot list resource "clusterroles" in API group "rbac.authorization.k8s.io" at the cluster scope
Error from server (Forbidden): clusterrolebindings.rbac.authorization.k8s.io is forbidden: User "restricted" cannot list resource "clusterrolebindings" in API group "rbac.authorization.k8s.io" at the cluster scope
```

But no permissions to view RBAC resources. So we try the obvious:

```
➜ k -n restricted get secret
Error from server (Forbidden): secrets is forbidden: User "restricted" cannot list resource "secrets" in API group "" in the namespace "restricted"

➜ k -n restricted get secret -o yaml
apiVersion: v1
items: []
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
Error from server (Forbidden): secrets is forbidden: User "restricted" cannot list resource "secrets" in API group "" in the namespace "restricted"
```

We're not allowed to get or list any *Secrets*. What can we see though?

```
➜ k -n restricted get all
NAME                      READY   STATUS    RESTARTS   AGE
pod1-fd5d64b9c-pcx6q      1/1     Running   0          37s
pod2-6494f7699b-4hks5     1/1     Running   0          37s
pod3-748b48594-24s76      1/1     Running   0          37s
Error from server (Forbidden): replicationcontrollers is forbidden: User "restricted" cannot list resource "replicationcontrollers" in API group "" in the namespace "restricted"
Error from server (Forbidden): services is forbidden: User "restricted" cannot list resource "services" in API group "" in the namespace "restricted"
...
```

There are some *Pods*, lets check these out regarding *Secret* access:

```
k -n restricted get pod -o yaml | grep -i secret
```

This output provides us with enough information to do:

```
➜ k -n restricted exec pod1-fd5d64b9c-pcx6q -- cat /etc/secret-volume/password
you-are

➜ echo you-are > /opt/course/12/secret1
```

And for the second *Secret*:

```
➜ k -n restricted exec pod2-6494f7699b-4hks5 -- env | grep PASS
PASSWORD=an-amazing

➜ echo an-amazing > /opt/course/12/secret2
```

None of the *Pods* seem to mount `secret3` though. Can we create or edit existing *Pods* to mount `secret3`?

```
➜ k -n restricted run test --image=nginx
Error from server (Forbidden): pods is forbidden: User "restricted" cannot create resource "pods" in API group "" in the namespace "restricted"

➜ k -n restricted delete pod pod1
Error from server (Forbidden): pods "pod1" is forbidden: User "restricted" cannot delete resource "pods" in API group "" in the namespace "restricted"
```

Doesn't look like it.

But the *Pods* seem to be able to access the *Secrets*, we can try to use a *Pod's ServiceAccount* to access the third *Secret*. We can actually see (like using `k -n restricted get pod -o yaml | grep automountServiceAccountToken`) that only *Pod* `pod3-*` has the *ServiceAccount* token mounted:

```
➜ k -n restricted exec -it pod3-748b48594-24s76 -- sh

/ # mount | grep serviceaccount
tmpfs on /run/secrets/kubernetes.io/serviceaccount type tmpfs (ro,relatime)

/ # ls /run/secrets/kubernetes.io/serviceaccount
ca.crt     namespace  token
```

> ***NOTE:*** You should have knowledge about *ServiceAccounts* and how they work with *Pods* like [described in the docs]

We can see all necessary information to contact the apiserver manually:

```
/ # curl https://kubernetes.default/api/v1/namespaces/restricted/secrets -H "Authorization: Bearer $(cat
/run/secrets/kubernetes.io/serviceaccount/token)" -k
...
    {
      "metadata": {
        "name": "secret3",
        "namespace": "restricted",
...
          }
        ]
      },
      "data": {
        "password": "cEVuRXRSYVRpT24tdEVzVGVSCg=="
      },
      "type": "Opaque"
    }
...
```

Let's encode it and write it into the requested location:

```
➜ echo cEVuRXRSYVRpT24tdEVzVGVSCg== | base64 -d
pEnEtRaTiOn-tEsTeR

➜ echo cEVuRXRSYVRpT24tdEVzVGVSCg== | base64 -d > /opt/course/12/secret3
```

This will give us:

```
# /opt/course/12/secret1
you-are
```

```
# /opt/course/12/secret2
an-amazing
```

```
# /opt/course/12/secret3
pEnEtRaTiOn-tEsTeR
```

We hacked all *Secrets*! It can be tricky to get RBAC right and secure.

> ***NOTE:*** One thing to consider is that giving the permission to "list" *Secrets*, will also allow the user to read the *Secret* values like using `kubectl get secrets -o yaml` even without the "get" permission set.

Question 13 | Restrict access to Metadata Server

Task weight: 7%

Use context: `kubectl config use-context infra-prod`

There is a metadata service available at `http://192.168.100.21:32000` on which *Nodes* can reach sensitive data, like cloud credentials for initialisation. By default, all *Pods* in the cluster also have access to this endpoint. The DevSecOps team has asked you to restrict access to this metadata server.

In *Namespace* `metadata-access`:

- Create a *NetworkPolicy* named `metadata-deny` which prevents egress to `192.168.100.21` for all *Pods* but still allows access to everything else
- Create a *NetworkPolicy* named `metadata-allow` which allows *Pods* having label `role: metadata-accessor` to access endpoint `192.168.100.21`

There are existing *Pods* in the target *Namespace* with which you can test your policies, but don't change their labels.

Answer:

There was a famous hack at Spotify which was based on revealed information via metadata for nodes.

Check the *Pods* in the *Namespace* `metadata-access` and their labels:

```
➜ k -n metadata-access get pods --show-labels
NAME                    ...   LABELS
pod1-7d67b4ff9-xrcd7    ...   app=pod1,pod-template-hash=7d67b4ff9
pod2-7b6fc66944-2hc7n   ...   app=pod2,pod-template-hash=7b6fc66944
pod3-7dc879bd59-hkgrr   ...   app=pod3,role=metadata-accessor,pod-template-hash=7dc879bd59
```

There are three *Pods* in the *Namespace* and one of them has the label `role=metadata-accessor`.

Check access to the metadata server from the *Pods*:

```
➜ k exec -it -n metadata-access pod1-7d67b4ff9-xrcd7 -- curl http://192.168.100.21:32000
metadata server

➜ k exec -it -n metadata-access pod2-7b6fc66944-2hc7n -- curl http://192.168.100.21:32000
metadata server

➜ k exec -it -n metadata-access pod3-7dc879bd59-hkgrr -- curl http://192.168.100.21:32000
metadata server
```

All three are able to access the metadata server.

To restrict the access, we create a *NetworkPolicy* to deny access to the specific IP.

```
vim 13_metadata-deny.yaml
```

```yaml
# 13_metadata-deny.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: metadata-deny
  namespace: metadata-access
spec:
  podSelector: {}
  policyTypes:
  - Egress
  egress:
  - to:
    - ipBlock:
        cidr: 0.0.0.0/0
        except:
        - 192.168.100.21/32
```

```
k -f 13_metadata-deny.yaml apply
```

> ***NOTE:*** You should know about general default-deny K8s NetworkPolcies.

Verify that access to the metadata server has been blocked, but other endpoints are still accessible:

```
➜ k exec -it -n metadata-access pod1-7d67b4ff9-xrcd7 -- curl http://192.168.100.21:32000
curl: (28) Failed to connect to 192.168.100.21 port 32000: Operation timed out
command terminated with exit code 28

➜ kubectl exec -it -n metadata-access pod1-7d67b4ff9-xrcd7 -- curl -I https://kubernetes.io
HTTP/2 200
cache-control: public, max-age=0, must-revalidate
content-type: text/html; charset=UTF-8
date: Mon, 14 Sep 2020 15:39:39 GMT
etag: "b46e429397e5f1fecf48c10a533f5cd8-ssl"
strict-transport-security: max-age=31536000
age: 13
content-length: 22252
server: Netlify
x-nf-request-id: 1d94a1d1-6bac-4a98-b065-346f661f1db1-393998290
```

Similarly, verify for the other two *Pods*.

Now create another *NetworkPolicy* that allows access to the metadata server from *Pods* with label `role=metadata-accessor`.

```
vim 13_metadata-allow.yaml
```

```yaml
# 13_metadata-allow.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: metadata-allow
  namespace: metadata-access
spec:
  podSelector:
    matchLabels:
      role: metadata-accessor
  policyTypes:
  - Egress
  egress:
  - to:
    - ipBlock:
        cidr: 192.168.100.21/32
```

```
k -f 13_metadata-allow.yaml apply
```

Verify that required *Pod* has access to metadata endpoint and others do not:

```
➜ k -n metadata-access exec pod3-7dc879bd59-hkgrr -- curl http://192.168.100.21:32000
metadata server

➜ k -n metadata-access exec pod2-7b6fc66944-9ngzr  -- curl http://192.168.100.21:32000
^Ccommand terminated with exit code 130
```

It only works for the *Pod* having the label. With this we implemented the required security restrictions.

If a *Pod* doesn't have a matching *NetworkPolicy* then all traffic is allowed from and to it. Once a *Pod* has a matching *NP* then the contained rules are additive. This means that for *Pods* having label `metadata-accessor` the rules will be combined to:

```yaml
# merged policies into one for pods with label metadata-accessor
spec:
  podSelector: {}
  policyTypes:
  - Egress
  egress:
  - to: # first rule
    - ipBlock: # condition 1
        cidr: 0.0.0.0/0
        except:
        - 192.168.100.21/32
  - to: # second rule
    - ipBlock: # condition 1
        cidr: 192.168.100.21/32
```

We can see that the merged *NP* contains two separate rules with one condition each. We could read it as:

```
Allow outgoing traffic if:
(destination is 0.0.0.0/0 but not 192.168.100.21/32) OR (destination is 192.168.100.21/32)
```

Hence it allows *Pods* with label `metadata-accessor` to access everything.

Question 14 | Syscall Activity

Task weight: 4%

Use context: `kubectl config use-context workload-prod`

There are *Pods* in *Namespace* `team-yellow`. A security investigation noticed that some processes running in these *Pods* are using the Syscall `kill`, which is forbidden by a Team Yellow internal policy.

Find the offending *Pod(s)* and remove these by reducing the replicas of the parent *Deployment* to 0.

Answer:

Syscalls are used by processes running in Userspace to communicate with the Linux Kernel. There are many available syscalls: https://man7.org/linux/man-pages/man2/syscalls.2.html. It makes sense to restrict these for container processes and Docker/Containerd already restrict some by default, like the `reboot` Syscall. Restricting even more is possible for example using Seccomp or AppArmor.

But for this task we should simply find out which binary process executes a specific Syscall. Processes in containers are simply run on the same Linux operating system, but isolated. That's why we first check on which nodes the *Pods* are running:

```
➜ k -n team-yellow get pod -owide
NAME                              ...   NODE              NOMINATED NODE   ...
collector1-7585cc58cb-n5rtd   1/1    ...   cluster1-worker1   <none>           ...
collector1-7585cc58cb-vdlp9   1/1    ...   cluster1-worker1   <none>           ...
collector2-8556679d96-z7g7c   1/1    ...   cluster1-worker1   <none>           ...
collector3-8b58fdc88-pjg24    1/1    ...   cluster1-worker1   <none>           ...
collector3-8b58fdc88-s9ltc    1/1    ...   cluster1-worker1   <none>           ...
```

All on `cluster1-worker1`, hence we ssh into it and find the processes for the first *Deployment* `collector1`.

```
➜ ssh cluster1-worker1

➜ root@cluster1-worker1:~# crictl pods --name collector1
POD ID              CREATED             STATE       NAME                          ...
21aacb8f4ca8d       17 minutes ago      Ready       collector1-7585cc58cb-vdlp9   ...
186631e40104d       17 minutes ago      Ready       collector1-7585cc58cb-n5rtd   ...

➜ root@cluster1-worker1:~# crictl ps --pod 21aacb8f4ca8d
CONTAINER ID        IMAGE               CREATED          ...       POD ID
9ea02422f8660       5d867958e04e1       12 minutes ago   ...       21aacb8f4ca8d

➜ root@cluster1-worker1:~# crictl inspect 9ea02422f8660 | grep args -A1
        "args": [
          "./collector1-process"
```

1. Using `crictl pods` we first searched for the *Pods* of *Deployment* `collector1`, which has two replicas
2. We then took one pod-id to find it's containers using `crictl ps`
3. And finally we used `crictl inspect` to find the process name, which is `collector1-process`

We can find the process PIDs (two because there are two *Pods*):

```
➜ root@cluster1-worker1:~# ps aux | grep collector1-process
root       35039  0.0  0.1 702208  1044 ?        Ssl  13:37   0:00 ./collector1-process
root       35059  0.0  0.1 702208  1044 ?        Ssl  13:37   0:00 ./collector1-process
```

Using the PIDs we can call `strace` to find Sycalls:

```
➜ root@cluster1-worker1:~# strace -p 35039
strace: Process 35039 attached
futex(0x4d7e68, FUTEX_WAIT_PRIVATE, 0, NULL) = 0
kill(666, SIGTERM)                      = -1 ESRCH (No such process)
epoll_pwait(3, [], 128, 999, NULL, 1)   = 0
kill(666, SIGTERM)                      = -1 ESRCH (No such process)
epoll_pwait(3, [], 128, 999, NULL, 1)   = 0
kill(666, SIGTERM)                      = -1 ESRCH (No such process)
epoll_pwait(3, ^Cstrace: Process 35039 detached
 <detached ...>
...
```

First try and already a catch! We see it uses the forbidden Syscall by calling `kill(666, SIGTERM)`.

Next let's check the *Deployment* `collector2` processes:

```
➜ root@cluster1-worker1:~# ps aux | grep collector2-process
root       35375  0.0  0.0 702216   604 ?        Ssl  13:37   0:00 ./collector2-process

➜ root@cluster1-worker1:~# strace -p 35375
strace: Process 35375 attached
futex(0x4d9e68, FUTEX_WAIT_PRIVATE, 0, NULL) = 0
futex(0x4d9e68, FUTEX_WAIT_PRIVATE, 0, NULL) = 0
futex(0x4d9e68, FUTEX_WAIT_PRIVATE, 0, NULL) = 0
futex(0x4d9e68, FUTEX_WAIT_PRIVATE, 0, NULL) = 0
...
```

Looks alright. What about `collector3`:

```
➜ root@cluster1-worker1:~# ps aux | grep collector3-process
root       35155  0.0  0.1 702472  1040 ?        Ssl  13:37   0:00 ./collector3-process
root       35241  0.0  0.1 702472  1044 ?        Ssl  13:37   0:00 ./collector3-process

➜ root@cluster1-worker1:~# strace -p 35155
strace: Process 35155 attached
futex(0x4d9e68, FUTEX_WAIT_PRIVATE, 0, NULL) = 0
futex(0x4d9e68, FUTEX_WAIT_PRIVATE, 0, NULL) = 0
futex(0x4d9e68, FUTEX_WAIT_PRIVATE, 0, NULL) = 0
epoll_pwait(3, [], 128, 999, NULL, 1)   = 0
epoll_pwait(3, [], 128, 999, NULL, 1)   = 0
...
```

Also nothing about the forbidden Syscall. So we finalise the task:

```
k -n team-yellow scale deploy collector1 --replicas 0
```

And the world is a bit safer again.

Question 15 | Configure TLS on Ingress

Task weight: 4%

Use context: `kubectl config use-context workload-prod`

In *Namespace* `team-pink` there is an existing Nginx *Ingress* resources named `secure` which accepts two paths `/app` and `/api` which point to different ClusterIP *Services*.

From your main terminal you can connect to it using for example:

- HTTP: `curl -v http://secure-ingress.test:31080/app`
- HTTPS: `curl -kv https://secure-ingress.test:31443/app`

Right now it uses a default generated TLS certificate by the Nginx Ingress Controller.

You're asked to instead use the key and certificate provided at `/opt/course/15/tls.key` and `/opt/course/15/tls.crt`. As it's a self-signed certificate you need to use `curl -k` when connecting to it.

Answer:

Investigate

We can get the IP address of the *Ingress* and we see it's the same one to which `secure-ingress.test` is pointing to:

```
➜ k -n team-pink get ing secure
NAME     CLASS    HOSTS                 ADDRESS          PORTS   AGE
secure   <none>   secure-ingress.test   192.168.100.12   80      7m11s

➜ ping secure-ingress.test
PING cluster1-worker1 (192.168.100.12) 56(84) bytes of data.
64 bytes from cluster1-worker1 (192.168.100.12): icmp_seq=1 ttl=64 time=0.316 ms
```

Now, let's try to access the paths `/app` and `/api` via HTTP:

```
➜ curl http://secure-ingress.test:31080/app
This is the backend APP!

➜ curl http://secure-ingress.test:31080/api
This is the API Server!
```

What about HTTPS?

```
➜ curl https://secure-ingress.test:31443/api
curl: (60) SSL certificate problem: unable to get local issuer certificate
More details here: https://curl.haxx.se/docs/sslcerts.html

curl failed to verify the legitimacy of the server and therefore could not
establish a secure connection to it. To learn more about this situation and
how to fix it, please visit the web page mentioned above.

➜ curl -k https://secure-ingress.test:31443/api
This is the API Server!
```

HTTPS seems to be already working if we accept self-signed certificated using `-k`. But what kind of certificate is used by the server?

```
➜ curl -kv https://secure-ingress.test:31443/api
...
* Server certificate:
*  subject: O=Acme Co; CN=Kubernetes Ingress Controller Fake Certificate
*  start date: Sep 28 12:28:35 2020 GMT
*  expire date: Sep 28 12:28:35 2021 GMT
*  issuer: O=Acme Co; CN=Kubernetes Ingress Controller Fake Certificate
*  SSL certificate verify result: unable to get local issuer certificate (20), continuing anyway.
...
```

It seems to be "Kubernetes Ingress Controller Fake Certificate".

Implement own TLS certificate

First, let us generate a *Secret* using the provided key and certificate:

```
➜ cd /opt/course/15

➜ :/opt/course/15$ ls
tls.crt  tls.key

➜ :/opt/course/15$ k -n team-pink create secret tls tls-secret --key tls.key --cert tls.crt
secret/tls-secret created
```

Now, we configure the *Ingress* to make use of this *Secret*:

```
➜ k -n team-pink get ing secure -oyaml > 15_ing_bak.yaml

➜ k -n team-pink edit ing secure
```

```
# kubectl -n team-pink edit ing secure
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
...
  generation: 1
  name: secure
  namespace: team-pink
...
spec:
  tls:                              # add
    - hosts:                        # add
      - secure-ingress.test         # add
      secretName: tls-secret        # add
  rules:
  - host: secure-ingress.test
    http:
      paths:
      - backend:
          service:
            name: secure-app
            port: 80
        path: /app
        pathType: ImplementationSpecific
      - backend:
          service:
            name: secure-api
            port: 80
        path: /api
        pathType: ImplementationSpecific
...
```

After adding the changes we check the *Ingress* resource again:

```
➜ k -n team-pink get ing
NAME     CLASS    HOSTS                 ADDRESS          PORTS     AGE
secure   <none>   secure-ingress.test   192.168.100.12   80, 443   25m
```

It now actually lists port 443 for HTTPS. To verify:

```
➜ curl -k https://secure-ingress.test:31443/api
This is the API Server!

➜ curl -kv https://secure-ingress.test:31443/api
...
* Server certificate:
*  subject: CN=secure-ingress.test; O=secure-ingress.test
*  start date: Sep 25 18:22:10 2020 GMT
*  expire date: Sep 20 18:22:10 2040 GMT
*  issuer: CN=secure-ingress.test; O=secure-ingress.test
*  SSL certificate verify result: self signed certificate (18), continuing anyway.
...
```

We can see that the provided certificate is now being used by the *Ingress* for TLS termination.

Question 16 | Docker Image Attack Surface

Task weight: 7%

Use context: `kubectl config use-context workload-prod`

There is a *Deployment* `image-verify` in *Namespace* `team-blue` which runs image `registry.killer.sh:5000/image-verify:v1`. DevSecOps has asked you to improve this image by:

1. Changing the base image to `alpine:3.12`
2. Not installing `curl`
3. Updating `nginx` to use the version constraint `>=1.18.0`
4. Running the main process as user `myuser`

Do not add any new lines to the Dockerfile, just edit existing ones. The file is located at `/opt/course/16/image/Dockerfile`.

Tag your version as `v2`. You can build, tag and push using:

```
cd /opt/course/16/image
podman build -t registry.killer.sh:5000/image-verify:v2 .
podman run registry.killer.sh:5000/image-verify:v2 # to test your changes
podman push registry.killer.sh:5000/image-verify:v2
```

Make the *Deployment* use your updated image tag `v2`.

Answer:

We should have a look at the Docker Image at first:

```
cd /opt/course/16/image

cp Dockerfile Dockerfile.bak

vim Dockerfile
```

```
# /opt/course/16/image/Dockerfile
FROM alpine:3.4
RUN apk update && apk add vim curl nginx=1.10.3-r0
RUN addgroup -S myuser && adduser -S myuser -G myuser
COPY ./run.sh run.sh
RUN ["chmod", "+x", "./run.sh"]
USER root
ENTRYPOINT ["/bin/sh", "./run.sh"]
```

Very simple Dockerfile which seems to execute a script `run.sh`:

```
# /opt/course/16/image/run.sh
while true; do date; id; echo; sleep 1; done
```

So it only outputs current date and credential information in a loop. We can see that output in the existing *Deployment* `image-verify`:

```
➜ k -n team-blue logs -f -l id=image-verify
Fri Sep 25 20:59:12 UTC 2020
uid=0(root) gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy),20(dialout),26(tape),27(video)
```

We see its running as root.

Next we update the `Dockerfile` according to the requirements:

```
# /opt/course/16/image/Dockerfile

# change
FROM alpine:3.12

# change
RUN apk update && apk add vim nginx>=1.18.0

RUN addgroup -S myuser && adduser -S myuser -G myuser
COPY ./run.sh run.sh
RUN ["chmod", "+x", "./run.sh"]

# change
USER myuser

ENTRYPOINT ["/bin/sh", "./run.sh"]
```

Then we build the new image:

```
➜ :/opt/course/16/image$ podman build -t registry.killer.sh:5000/image-verify:v2 .
...
STEP 7/7: ENTRYPOINT ["/bin/sh", "./run.sh"]
COMMIT registry.killer.sh:5000/image-verify:v2
--> ceb8989101b
Successfully tagged registry.killer.sh:5000/image-verify:v2
ceb8989101bccd9f6b9c3b4c6c75f6c3561f19a5b784edd1f1a36fa0fb34a9df
```

We can then test our changes by running the container locally:

```
➜ :/opt/course/16/image$ podman run registry.killer.sh:5000/image-verify:v2
Thu Sep 16 06:01:47 UTC 2021
uid=101(myuser) gid=102(myuser) groups=102(myuser)

Thu Sep 16 06:01:48 UTC 2021
uid=101(myuser) gid=102(myuser) groups=102(myuser)

Thu Sep 16 06:01:49 UTC 2021
uid=101(myuser) gid=102(myuser) groups=102(myuser)
```

Looking good, so we push:

```
➜ :/opt/course/16/image$ podman push registry.killer.sh:5000/image-verify:v2
Getting image source signatures
Copying blob cd0853834d88 done
Copying blob 5298d0709c3e skipped: already exists
Copying blob e6688e911f15 done
Copying blob dbc406096645 skipped: already exists
Copying blob 98895ed393d9 done
Copying config ceb8989101 done
Writing manifest to image destination
Storing signatures
```

And we update the *Deployment* to use the new image:

```
k -n team-blue edit deploy image-verify
```

```
# kubectl -n team-blue edit deploy image-verify
apiVersion: apps/v1
kind: Deployment
metadata:
...
spec:
...
  template:
...
    spec:
      containers:
      - image: registry.killer.sh:5000/image-verify:v2 # change
```

And afterwards we can verify our changes by looking at the *Pod* logs:

```
➜ k -n team-blue logs -f -l id=image-verify
Fri Sep 25 21:06:55 UTC 2020
uid=101(myuser) gid=102(myuser) groups=102(myuser)
```

Also to verify our changes even further:

```
➜ k -n team-blue exec image-verify-55fbcd4c9b-x2flc -- curl
OCI runtime exec failed: exec failed: container_linux.go:349: starting container process caused "exec: \"curl\":
executable file not found in $PATH": unknown
command terminated with exit code 126

➜ k -n team-blue exec image-verify-55fbcd4c9b-x2flc -- nginx -v
nginx version: nginx/1.18.0
```

Another task solved.

Question 17 | Audit Log Policy

Task weight: 7%

Use context: `kubectl config use-context infra-prod`

Audit Logging has been enabled in the cluster with an Audit *Policy* located at `/etc/kubernetes/audit/policy.yaml` on `cluster2-master1`.

Change the configuration so that only one backup of the logs is stored.

Alter the *Policy* in a way that it only stores logs:

1. From *Secret* resources, level Metadata
2. From "system:nodes" userGroups, level RequestResponse

After you altered the *Policy* make sure to empty the log file so it only contains entries according to your changes, like using `truncate -s 0 /etc/kubernetes/audit/logs/audit.log`.

> ***NOTE:*** You can use `jq` to render json more readable. `cat data.json | jq`

Answer:

First we check the apiserver configuration and change as requested:

```
➜ ssh cluster2-master1

➜ root@cluster2-master1:~# cp /etc/kubernetes/manifests/kube-apiserver.yaml ~/17_kube-apiserver.yaml # backup

➜ root@cluster2-master1:~# vim /etc/kubernetes/manifests/kube-apiserver.yaml
```

```yaml
# /etc/kubernetes/manifests/kube-apiserver.yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubeadm.kubernetes.io/kube-apiserver.advertise-address.endpoint: 192.168.100.21:6443
  creationTimestamp: null
  labels:
    component: kube-apiserver
    tier: control-plane
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    - --audit-policy-file=/etc/kubernetes/audit/policy.yaml
    - --audit-log-path=/etc/kubernetes/audit/logs/audit.log
    - --audit-log-maxsize=5
    - --audit-log-maxbackup=1                                    # CHANGE
    - --advertise-address=192.168.100.21
    - --allow-privileged=true
...
```

> ***NOTE:*** You should know how to enable Audit Logging completely yourself [as described in the docs](). Feel free to try this in another cluster in this environment.

Now we look at the existing *Policy*:

```
➜ root@cluster2-master1:~# vim /etc/kubernetes/audit/policy.yaml
```

```yaml
# /etc/kubernetes/audit/policy.yaml
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
- level: Metadata
```

We can see that this simple *Policy* logs everything on Metadata level. So we change it to the requirements:

```yaml
# /etc/kubernetes/audit/policy.yaml
apiVersion: audit.k8s.io/v1
kind: Policy
rules:

# log Secret resources audits, level Metadata
- level: Metadata
  resources:
  - group: ""
    resources: ["secrets"]

# log node related audits, level RequestResponse
- level: RequestResponse
  userGroups: ["system:nodes"]

# for everything else don't log anything
- level: None
```

After saving the changes we have to restart the apiserver:

```
➜ root@cluster2-master1:~# cd /etc/kubernetes/manifests/

➜ root@cluster2-master1:/etc/kubernetes/manifests# mv kube-apiserver.yaml ..

➜ root@cluster2-master1:/etc/kubernetes/manifests# watch crictl ps # wait for apiserver gone

➜ root@cluster2-master1:/etc/kubernetes/manifests# truncate -s 0 /etc/kubernetes/audit/logs/audit.log

➜ root@cluster2-master1:/etc/kubernetes/manifests# mv ../kube-apiserver.yaml .
```

Once the apiserver is running again we can check the new logs and scroll through some entries:

```
cat audit.log | tail | jq
```


```
{
  "kind": "Event",
  "apiVersion": "audit.k8s.io/v1",
  "level": "Metadata",
  "auditID": "e598dc9e-fc8b-4213-aee3-0719499ab1bd",
  "stage": "RequestReceived",
  "requestURI": "...",
  "verb": "watch",
  "user": {
    "username": "system:serviceaccount:gatekeeper-system:gatekeeper-admin",
    "uid": "79870838-75a8-479b-ad42-4b7b75bd17a3",
    "groups": [
      "system:serviceaccounts",
      "system:serviceaccounts:gatekeeper-system",
      "system:authenticated"
    ]
  },
  "sourceIPs": [
    "192.168.102.21"
  ],
  "userAgent": "manager/v0.0.0 (linux/amd64) kubernetes/$Format",
  "objectRef": {
    "resource": "secrets",
    "apiVersion": "v1"
  },
  "requestReceivedTimestamp": "2020-09-27T20:01:36.238911Z",
  "stageTimestamp": "2020-09-27T20:01:36.238911Z",
  "annotations": {
    "authentication.k8s.io/legacy-token": "..."
  }
}
```


Above we logged a watch action by OPA Gatekeeper for *Secrets*, level Metadata.


```
{
  "kind": "Event",
  "apiVersion": "audit.k8s.io/v1",
  "level": "RequestResponse",
  "auditID": "c90e53ed-b0cf-4cc4-889a-f1204dd39267",
  "stage": "ResponseComplete",
  "requestURI": "...",
  "verb": "list",
  "user": {
    "username": "system:node:cluster2-master1",
    "groups": [
      "system:nodes",
      "system:authenticated"
    ]
  },
  "sourceIPs": [
    "192.168.100.21"
  ],
  "userAgent": "kubelet/v1.19.1 (linux/amd64) kubernetes/206bcad",
  "objectRef": {
    "resource": "configmaps",
    "namespace": "kube-system",
    "name": "kube-proxy",
    "apiVersion": "v1"
  },
  "responseStatus": {
    "metadata": {},
    "code": 200
  },
  "responseObject": {
    "kind": "ConfigMapList",
    "apiVersion": "v1",
    "metadata": {
      "selfLink": "/api/v1/namespaces/kube-system/configmaps",
```

```
        "resourceVersion": "83409"
      },
      "items": [
        {
          "metadata": {
            "name": "kube-proxy",
            "namespace": "kube-system",
            "selfLink": "/api/v1/namespaces/kube-system/configmaps/kube-proxy",
            "uid": "0f1c3950-430a-4543-83e4-3f9c87a478b8",
            "resourceVersion": "232",
            "creationTimestamp": "2020-09-26T20:59:50Z",
            "labels": {
              "app": "kube-proxy"
            },
            "annotations": {
              "kubeadm.kubernetes.io/component-config.hash": "..."
            },
            "managedFields": [
              {
...
              }
            ]
          },
...
        }
      ]
    },
    "requestReceivedTimestamp": "2020-09-27T20:01:36.223781Z",
    "stageTimestamp": "2020-09-27T20:01:36.225470Z",
    "annotations": {
      "authorization.k8s.io/decision": "allow",
      "authorization.k8s.io/reason": ""
    }
}
```


And in the one above we logged a list action by system:nodes for a *ConfigMaps*, level RequestResponse.

Because all JSON entries are written in a single line in the file we could also run some simple verifications on our *Policy*:

```
# shows Secret entries
cat audit.log | grep '"resource":"secrets"' | wc -l

# confirms Secret entries are only of level Metadata
cat audit.log | grep '"resource":"secrets"' | grep -v '"level":"Metadata"' | wc -l

# shows RequestResponse level entries
cat audit.log | grep -v '"level":"RequestResponse"' | wc -l

# shows RequestResponse level entries are only for system:nodes
cat audit.log | grep '"level":"RequestResponse"' | grep -v "system:nodes" | wc -l
```

Looks like our job is done.

Question 18 | Investigate Break-in via Audit Log

Task weight: 4%

Use context: `kubectl config use-context infra-prod`

Namespace `security` contains five *Secrets* of type Opaque which can be considered highly confidential. The latest Incident-Prevention-Investigation revealed that *ServiceAccount* `p.auster` had too broad access to the cluster for some time. This *SA* should've never had access to any *Secrets* in that *Namespace*.

Find out which *Secrets* in *Namespace* `security` this *SA* did access by looking at the Audit Logs under `/opt/course/18/audit.log`.

Change the password to any new string of only those *Secrets* that were accessed by this *SA*.

> ***NOTE:*** You can use `jq` to render json more readable. `cat data.json | jq`

Answer:

First we look at the *Secrets* this is about:

```
➜ k -n security get secret | grep Opaque
kubeadmin-token       Opaque                                1      37m
mysql-admin           Opaque                                1      37m
postgres001           Opaque                                1      37m
postgres002           Opaque                                1      37m
vault-token           Opaque                                1      37m
```

Next we investigate the Audit Log file:

```
➜ cd /opt/course/18

➜ :/opt/course/18$ ls -lh
total 7.1M
-rw-r--r-- 1 k8s k8s 7.5M Sep 24 21:31 audit.log

➜ :/opt/course/18$ cat audit.log | wc -l
4451
```

Audit Logs can be huge and it's common to limit the amount by creating an Audit *Policy* and to transfer the data in systems like Elasticsearch. In this case we have a simple JSON export, but it already contains 4451 lines.

We should try to filter the file down to relevant information:

```
➜ :/opt/course/18$ cat audit.log | grep "p.auster" | wc -l
28
```

Not too bad, only 28 logs for *ServiceAccount* `p.auster`.

```
➜ :/opt/course/18$ cat audit.log | grep "p.auster" | grep Secret | wc -l
2
```

And only 2 logs related to *Secrets*...

```
➜ :/opt/course/18$ cat audit.log | grep "p.auster" | grep Secret | grep list | wc -l
0

➜ :/opt/course/18$ cat audit.log | grep "p.auster" | grep Secret | grep get | wc -l
2
```

No list actions, which is good, but 2 get actions, so we check these out:

```
cat audit.log | grep "p.auster" | grep Secret | grep get | jq
```

```
{
  "kind": "Event",
  "apiVersion": "audit.k8s.io/v1",
  "level": "RequestResponse",
  "auditID": "74fd9e03-abea-4df1-b3d0-9cfeff9ad97a",
  "stage": "ResponseComplete",
  "requestURI": "/api/v1/namespaces/security/secrets/vault-token",
  "verb": "get",
  "user": {
    "username": "system:serviceaccount:security:p.auster",
    "uid": "29ecb107-c0e8-4f2d-816a-b16f4391999c",
    "groups": [
      "system:serviceaccounts",
      "system:serviceaccounts:security",
      "system:authenticated"
    ]
  },
...
  "userAgent": "curl/7.64.0",
  "objectRef": {
    "resource": "secrets",
    "namespace": "security",
    "name": "vault-token",
    "apiVersion": "v1"
  },
 ...
}
{
  "kind": "Event",
  "apiVersion": "audit.k8s.io/v1",
  "level": "RequestResponse",
  "auditID": "aed6caf9-5af0-4872-8f09-ad55974bb5e0",
  "stage": "ResponseComplete",
  "requestURI": "/api/v1/namespaces/security/secrets/mysql-admin",
  "verb": "get",
  "user": {
    "username": "system:serviceaccount:security:p.auster",
    "uid": "29ecb107-c0e8-4f2d-816a-b16f4391999c",
```

```
      "groups": [
        "system:serviceaccounts",
        "system:serviceaccounts:security",
        "system:authenticated"
      ]
    },
  ...
    "userAgent": "curl/7.64.0",
    "objectRef": {
      "resource": "secrets",
      "namespace": "security",
      "name": "mysql-admin",
      "apiVersion": "v1"
    },
  ...
}
```

There we see that *Secrets* `vault-token` and `mysql-admin` were accessed by `p.auster`. Hence we change the passwords for those.

```
➜ echo new-vault-pass | base64
bmV3LXZhdWx0LXBhc3MK

➜ k -n security edit secret vault-token

➜ echo new-mysql-pass | base64
bmV3LW15c3FsLXBhc3MK

➜ k -n security edit secret mysql-admin
```

Audit Logs ftw.

By running `cat audit.log | grep "p.auster" | grep Secret | grep password` we can see that passwords are stored in the Audit Logs, because they store the complete content of *Secrets*. It's never a good idea to reveal passwords in logs. In this case it would probably be sufficient to only store Metadata level information of *Secrets* which can be controlled via a Audit *Policy*.

Question 19 | Immutable Root FileSystem

Task weight: 2%

Use context: `kubectl config use-context workload-prod`

The *Deployment* `immutable-deployment` in *Namespace* `team-purple` should run immutable, it's created from file `/opt/course/19/immutable-deployment.yaml`. Even after a successful break-in, it shouldn't be possible for an attacker to modify the filesystem of the running container.

Modify the *Deployment* in a way that no processes inside the container can modify the local filesystem, only `/tmp` directory should be writeable. Don't modify the Docker image.

Save the updated YAML under `/opt/course/19/immutable-deployment-new.yaml` and update the running *Deployment*.

Answer:

Processes in containers can write to the local filesystem by default. This increases the attack surface when a non-malicious process gets hijacked. Preventing applications to write to disk or only allowing to certain directories can mitigate the risk. If there is for example a bug in Nginx which allows an attacker to override any file inside the container, then this only works if the Nginx process itself can write to the filesystem in the first place.

Making the root filesystem readonly can be done in the Docker image itself or in a *Pod* declaration.

Let us first check the *Deployment* `immutable-deployment` in *Namespace* `team-purple`:

```
➜ k -n team-purple edit deploy -o yaml
```

```
# kubectl -n team-purple edit deploy -o yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: team-purple
  name: immutable-deployment
  labels:
    app: immutable-deployment
  ...
spec:
  replicas: 1
  selector:
    matchLabels:
      app: immutable-deployment
  template:
```

```
    metadata:
      labels:
        app: immutable-deployment
    spec:
      containers:
      - image: busybox:1.32.0
        command: ['sh', '-c', 'tail -f /dev/null']
        imagePullPolicy: IfNotPresent
        name: busybox
      restartPolicy: Always
...
```

The container has write access to the Root File System, as there are no restrictions defined for the *Pods* or containers by an existing SecurityContext. And based on the task we're not allowed to alter the Docker image.

So we modify the YAML manifest to include the required changes:

```
cp /opt/course/19/immutable-deployment.yaml /opt/course/19/immutable-deployment-new.yaml

vim /opt/course/19/immutable-deployment-new.yaml
```

```
# /opt/course/19/immutable-deployment-new.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: team-purple
  name: immutable-deployment
  labels:
    app: immutable-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: immutable-deployment
  template:
    metadata:
      labels:
        app: immutable-deployment
    spec:
      containers:
      - image: busybox:1.32.0
        command: ['sh', '-c', 'tail -f /dev/null']
        imagePullPolicy: IfNotPresent
        name: busybox
        securityContext:                  # add
          readOnlyRootFilesystem: true    # add
        volumeMounts:                     # add
        - mountPath: /tmp                 # add
          name: temp-vol                  # add
      volumes:                            # add
      - name: temp-vol                    # add
        emptyDir: {}                      # add
      restartPolicy: Always
```

SecurityContexts can be set on *Pod* or container level, here the latter was asked. Enforcing `readOnlyRootFilesystem: true` will render the root filesystem readonly. We can then allow some directories to be writable by using an emptyDir volume.

Once the changes are made, let us update the *Deployment*:

```
➜ k delete -f /opt/course/19/immutable-deployment-new.yaml
deployment.apps "immutable-deployment" deleted

➜ k create -f /opt/course/19/immutable-deployment-new.yaml
deployment.apps/immutable-deployment created
```

We can verify if the required changes are propagated:

```
➜ k -n team-purple exec immutable-deployment-5b7ff8d464-j2nrj -- touch /abc.txt
touch: /abc.txt: Read-only file system
command terminated with exit code 1

➜ k -n team-purple exec immutable-deployment-5b7ff8d464-j2nrj -- touch /var/abc.txt
touch: /var/abc.txt: Read-only file system
command terminated with exit code 1

➜ k -n team-purple exec immutable-deployment-5b7ff8d464-j2nrj -- touch /etc/abc.txt
touch: /etc/abc.txt: Read-only file system
command terminated with exit code 1

➜ k -n team-purple exec immutable-deployment-5b7ff8d464-j2nrj -- touch /tmp/abc.txt

➜ k -n team-purple exec immutable-deployment-5b7ff8d464-j2nrj -- ls /tmp
abc.txt
```

The *Deployment* has been updated so that the container's file system is read-only, and the updated YAML has been placed under the required location. Sweet!

Question 20 | Update Kubernetes

Task weight: 8%

Use context: `kubectl config use-context workload-stage`

The cluster is running Kubernetes `1.21.4`. Update it to `1.22.1` available via `apt` package manager.

Use `ssh cluster3-master1` and `ssh cluster3-worker1` to connect to the instances.

Answer:

Let's have a look at the current versions:

```
➜ k get node
NAME               STATUS   ROLES                  AGE   VERSION
cluster3-master1   Ready    control-plane,master   83m   v1.21.4
cluster3-worker1   Ready    <none>                 78m   v1.21.4
```

Control Plane Master Components

First we should update the control plane components running on the master node, so we drain it:

```
➜ k drain cluster3-master1 --ignore-daemonsets
```

Next we ssh into it and check versions:

```
➜ ssh cluster3-master1

➜ root@cluster3-master1:~# kubeadm version
kubeadm version: &version.Info{Major:"1", Minor:"22", GitVersion:"v1.22.1",
GitCommit:"632ed300f2c34f6d6d15ca4cef3d3c7073412212", GitTreeState:"clean", BuildDate:"2021-08-19T15:44:22Z",
GoVersion:"go1.16.7", Compiler:"gc", Platform:"linux/amd64"}

➜ root@cluster3-master1:~# kubelet --version
Kubernetes v1.21.4
```

We see `kubeadm` is already installed in the required version. Else we would need to install it:

```
apt-get install kubeadm=1.22.1-00
```

Check what kubeadm has available as an upgrade plan:

```
➜ root@cluster3-master1:~# kubeadm upgrade plan
[upgrade/config] Making sure the configuration is correct:
[upgrade/config] Reading configuration from the cluster...
[upgrade/config] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
[preflight] Running pre-flight checks.
[upgrade] Running cluster health checks
[upgrade] Fetching available versions to upgrade to
[upgrade/versions] Cluster version: v1.21.4
[upgrade/versions] kubeadm version: v1.22.1
[upgrade/versions] Target version: v1.22.1
[upgrade/versions] Latest version in the v1.21 series: v1.21.4
...
```

And we apply to the required version:

```
➜ root@cluster3-master1:~# kubeadm upgrade apply v1.22.1
[upgrade/config] Making sure the configuration is correct:
[upgrade/config] Reading configuration from the cluster...
[upgrade/config] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
[preflight] Running pre-flight checks.
[upgrade] Running cluster health checks
[upgrade/version] You have chosen to change the cluster version to "v1.22.1"
[upgrade/versions] Cluster version: v1.21.4
[upgrade/versions] kubeadm version: v1.22.1
[upgrade/confirm] Are you sure you want to proceed with the upgrade? [y/N]: y
...
[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.22.1". Enjoy!
[upgrade/kubelet] Now that your control plane is upgraded, please proceed with upgrading your kubelets if you haven't
already done so.
```

After this finished we verify we're up to date by showing upgrade plans again:

```
➜ root@cluster3-master1:~# kubeadm upgrade plan
[upgrade/config] Making sure the configuration is correct:
[upgrade/config] Reading configuration from the cluster...
[upgrade/config] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
[preflight] Running pre-flight checks.
[upgrade] Running cluster health checks
[upgrade] Fetching available versions to upgrade to
[upgrade/versions] Cluster version: v1.22.1
[upgrade/versions] kubeadm version: v1.22.1
[upgrade/versions] Target version: v1.22.1
[upgrade/versions] Latest version in the v1.22 series: v1.22.1
```

Control Plane kubelet and kubectl

```
➜ root@cluster3-master1:~# apt-get update
...
Reading package lists... Done

➜ root@cluster3-master1:~# apt-get install kubelet=1.22.1-00 kubectl=1.22.1-00
...
Unpacking kubelet (1.22.1-00) over (1.21.4-00) ...
Setting up kubectl (1.22.1-00) ...
Setting up containers-common (100:1-21) ...
Setting up kubelet (1.22.1-00) ...
Setting up podman (100:3.3.1-1) ...

➜ root@cluster3-master1:~# systemctl daemon-reload && systemctl restart kubelet

➜ root@cluster3-master1:~# kubectl get node
NAME               STATUS                     ROLES                  AGE   VERSION
cluster3-master1   Ready,SchedulingDisabled   control-plane,master   89m   v1.22.1
cluster3-worker1   Ready                      <none>                 85m   v1.21.4
```

Done, and uncordon:

```
➜ k uncordon cluster3-master1
node/cluster3-master1 uncordoned
```

Data Plane

```
➜ k get node
NAME               STATUS   ROLES                  AGE   VERSION
cluster3-master1   Ready    control-plane,master   90m   v1.22.1
cluster3-worker1   Ready    <none>                 85m   v1.21.4
```

Our data plane consist of one single worker node, so let's update it. First thing is we should drain it:

```
k drain cluster3-worker1 --ignore-daemonsets
```

Next we ssh into it and upgrade kubeadm to the wanted version, or check if already done:

```
➜ ssh cluster3-worker1

➜ root@cluster3-worker1:~# apt-get update
...
Reading package lists... Done

➜ root@cluster3-worker1:~# apt-get install kubeadm=1.22.1-00
Reading package lists... Done
Building dependency tree
Reading state information... Done
...

➜ root@cluster3-worker1:~# kubeadm upgrade node
[upgrade] Reading configuration from the cluster...
[upgrade] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
[preflight] Running pre-flight checks
[preflight] Skipping prepull. Not a control plane node.
[upgrade] Skipping phase. Not a control plane node.
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[upgrade] The configuration for this node was successfully updated!
[upgrade] Now you should go ahead and upgrade the kubelet package using your package manager.
```

Now we follow that kubeadm told us in the last line and upgrade kubelet (and kubectl):

```
➜ root@cluster3-worker1:~# apt-get install kubelet=1.22.1-00 kubectl=1.22.1-00
...
(Reading database ... 111894 files and directories currently installed.)
Preparing to unpack .../kubectl_1.22.1-00_amd64.deb ...
Unpacking kubectl (1.22.1-00) over (1.21.4-00) ...
Preparing to unpack .../kubelet_1.22.1-00_amd64.deb ...
Unpacking kubelet (1.22.1-00) over (1.21.4-00) ...
Setting up kubectl (1.22.1-00) ...
Setting up kubelet (1.22.1-00) ...

➜ root@cluster3-worker1:~# systemctl daemon-reload && systemctl restart kubelet
```

Looking good, what does the node status say?

```
➜ k get node
NAME               STATUS                     ROLES                  AGE   VERSION
cluster3-master1   Ready                      control-plane,master   97m   v1.22.1
cluster3-worker1   Ready,SchedulingDisabled   <none>                 92m   v1.22.1
```

Beautiful, let's make it schedulable again:

```
➜ k uncordon cluster3-worker1
node/cluster3-worker1 uncordoned

➜ k get node
NAME               STATUS   ROLES                  AGE   VERSION
cluster3-master1   Ready    control-plane,master   97m   v1.22.1
cluster3-worker1   Ready    <none>                 92m   v1.22.1
```

We're up to date.

Question 21 | Image Vulnerability Scanning

Task weight: 2%

(can be solved in any kubectl context)

The Vulnerability Scanner `trivy` is installed on your main terminal. Use it to scan the following images for known CVEs:

- `nginx:1.16.1-alpine`
- `k8s.gcr.io/kube-apiserver:v1.18.0`
- `k8s.gcr.io/kube-controller-manager:v1.18.0`
- `docker.io/weaveworks/weave-kube:2.7.0`

Write all images that don't contain the vulnerabilities `CVE-2020-10878` or `CVE-2020-1967` into `/opt/course/21/good-images`.

Answer:

The tool `trivy` is very simple to use, it compares images against public databases.

```
➜ trivy nginx:1.16.1-alpine
2020-10-09T20:59:39.198Z        INFO    Need to update DB
2020-10-09T20:59:39.198Z        INFO    Downloading DB...
18.81 MiB / 18.81 MiB [-------------------------------------
2020-10-09T20:59:45.499Z        INFO    Detecting Alpine vulnerabilities...

nginx:1.16.1-alpine (alpine 3.10.4)
===================================
Total: 7 (UNKNOWN: 0, LOW: 0, MEDIUM: 7, HIGH: 0, CRITICAL: 0)

+---------------+------------------+----------+-------------------
|    LIBRARY    | VULNERABILITY ID | SEVERITY | INSTALLED VERSION
+---------------+------------------+----------+-------------------
| libcrypto1.1  | CVE-2020-1967    | MEDIUM   | 1.1.1d-r2
...
```

To solve the task we can run:

```
➜ trivy nginx:1.16.1-alpine | grep -E 'CVE-2020-10878|CVE-2020-1967'
| libcrypto1.1   | CVE-2020-1967    | MEDIUM
| libssl1.1      | CVE-2020-1967    |

➜ trivy k8s.gcr.io/kube-apiserver:v1.18.0 | grep -E 'CVE-2020-10878|CVE-2020-1967'
| perl-base      | CVE-2020-10878      | HIGH

➜ trivy k8s.gcr.io/kube-controller-manager:v1.18.0 | grep -E 'CVE-2020-10878|CVE-2020-1967'
| perl-base      | CVE-2020-10878      | HIGH

➜ trivy docker.io/weaveworks/weave-kube:2.7.0 | grep -E 'CVE-2020-10878|CVE-2020-1967'
➜
```

The only image without the any of the two CVEs is `docker.io/weaveworks/weave-kube:2.7.0`, hence our answer will be:

```
# /opt/course/21/good-images
docker.io/weaveworks/weave-kube:2.7.0
```

Question 22 | Manual Static Security Analysis

Task weight: 3%

(can be solved in any kubectl context)

The Release Engineering Team has shared some YAML manifests and Dockerfiles with you to review. The files are located under `/opt/course/22/files`.

As a container security expert, you are asked to perform a manual static analysis and find out possible security issues with respect to unwanted credential exposure. Running processes as root is of no concern in this task.

Write the filenames which have issues into `/opt/course/22/security-issues`.

> ***NOTE:*** In the Dockerfile and YAML manifests, assume that the referred files, folders, secrets and volume mounts are present. Disregard syntax or logic errors.

Answer:

We check location `/opt/course/22/files` and list the files.

```
➜ ls -la /opt/course/22/files
total 48
drwxr-xr-x 2 k8s k8s 4096 Sep 16 19:08 .
drwxr-xr-x 3 k8s k8s 4096 Sep 16 19:08 ..
-rw-r--r-- 1 k8s k8s  692 Sep 16 19:08 Dockerfile-go
-rw-r--r-- 1 k8s k8s  897 Sep 16 19:08 Dockerfile-mysql
-rw-r--r-- 1 k8s k8s  743 Sep 16 19:08 Dockerfile-py
-rw-r--r-- 1 k8s k8s  341 Sep 16 19:08 deployment-nginx.yaml
-rw-r--r-- 1 k8s k8s  705 Sep 16 19:08 deployment-redis.yaml
-rw-r--r-- 1 k8s k8s  392 Sep 16 19:08 pod-nginx.yaml
-rw-r--r-- 1 k8s k8s  228 Sep 16 19:08 pv-manual.yaml
-rw-r--r-- 1 k8s k8s  188 Sep 16 19:08 pvc-manual.yaml
-rw-r--r-- 1 k8s k8s  211 Sep 16 19:08 sc-local.yaml
-rw-r--r-- 1 k8s k8s  902 Sep 16 19:08 statefulset-nginx.yaml
```

We have 3 Dockerfiles and 7 Kubernetes Resource YAML manifests. Next we should go over each to find security issues with the way credentials have been used.

> ***NOTE:*** You should be comfortable with Docker Best Practices and the Kubernetes Configuration Best Practices.

While navigating through the files we might notice:

Number 1

File `Dockerfile-mysql` might look innocent on first look. It copies a file `secret-token` over, uses it and deletes it afterwards. But because of the way Docker works, every `RUN`, `COPY` and `ADD` command creates a new layer and every layer is persistet in the image.

This means even if the file `secret-token` get's deleted in layer Z, it's still included with the image in layer X and Y. In this case it would be better to use for example variables passed to Docker.

```
# /opt/course/22/files/Dockerfile-mysql
```

```
FROM ubuntu

# Add MySQL configuration
COPY my.cnf /etc/mysql/conf.d/my.cnf
COPY mysqld_charset.cnf /etc/mysql/conf.d/mysqld_charset.cnf

RUN apt-get update && \
    apt-get -yq install mysql-server-5.6 &&

# Add MySQL scripts
COPY import_sql.sh /import_sql.sh
COPY run.sh /run.sh

# Configure credentials
COPY secret-token .                                       # LAYER X
RUN /etc/register.sh ./secret-token                       # LAYER Y
RUN rm ./secret-token # delete secret token again         # LATER Z

EXPOSE 3306
CMD ["/run.sh"]
```

So we do:

```
echo Dockerfile-mysql >> /opt/course/22/security-issues
```

Number 2

The file `deployment-redis.yaml` is fetching credentials from a *Secret* named `mysecret` and writes these into environment variables. So far so good, but in the command of the *container* it's echoing these which can be directly read by any user having access to the logs.

```
# /opt/course/22/files/deployment-redis.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: mycontainer
        image: redis
        command: ["/bin/sh"]
        args:
        - "-c"
        - "echo $SECRET_USERNAME && echo $SECRET_PASSWORD && docker-entrypoint.sh" # NOT GOOD
        env:
        - name: SECRET_USERNAME
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: username
        - name: SECRET_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: password
```

Credentials in logs is never a good idea, hence we do:

```
echo deployment-redis.yaml >> /opt/course/22/security-issues
```

Number 3

In file `statefulset-nginx.yaml`, the password is directly exposed in the environment variable definition of the *container*.

```
# /opt/course/22/files/statefulset-nginx.yaml
...
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 2
  selector:
    matchLabels:
      app: nginx
```

```
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: k8s.gcr.io/nginx-slim:0.8
        env:
        - name: Username
          value: Administrator
        - name: Password
          value: MyDiReCtP@sSw0rd               # NOT GOOD
        ports:
        - containerPort: 80
          name: web
..
```

This should better be injected via a *Secret*. So we do:

```
echo statefulset-nginx.yaml >> /opt/course/22/security-issues
```

```
➜ cat /opt/course/22/security-issues
Dockerfile-mysql
deployment-redis.yaml
statefulset-nginx.yaml
```

CKS Simulator Preview Kubernetes 1.22

https://killer.sh

This is a preview of the full CKS Simulator course content.

The full course contains 22 questions and scenarios which cover all the CKS areas. The course also provides a browser terminal which is a very close replica of the original one. This is great to get used and comfortable before the real exam. After the test session (120 minutes), or if you stop it early, you'll get access to all questions and their detailed solutions. You'll have 36 hours cluster access in total which means even after the session, once you have the solutions, you can still play around.

The following preview will give you an idea of what the full course will provide. These preview questions are not part of the 22 in the full course but in addition to it. But the preview questions are part of the same CKS simulation environment which we setup for you, so with access to the full course you can solve these too.

The answers provided here assume that you did run the initial terminal setup suggestions as provided in the tips section, but especially:

```
alias k=kubectl

export do="-o yaml --dry-run=client"
```

These questions can be solved in the test environment provided through the CKS Simulator

Preview Question 1

Use context: `kubectl config use-context infra-prod`

You have admin access to cluster2. There is also context `gianna@infra-prod` which authenticates as user `gianna` with the same cluster.

There are existing cluster-level RBAC resources in place to, among other things, ensure that user `gianna` can never read *Secret* contents cluster-wide. Confirm this is correct or restrict the existing RBAC resources to ensure this.

I addition, create more RBAC resources to allow user `gianna` to create *Pods* and *Deployments* in *Namespaces* `security`, `restricted` and `internal`. It's likely the user will receive these exact permissions as well for other *Namespaces* in the future.

Answer:

Part 1 - check existing RBAC rules

We should probably first have a look at the existing RBAC resources for user `gianna`. We don't know the resource names but we know these are cluster-level so we can search for a *ClusterRoleBinding*:

```
k get clusterrolebinding -oyaml | grep gianna -A10 -B20
```

From this we see the binding is also called `gianna`:

```
k edit clusterrolebinding gianna
```

```
# kubectl edit clusterrolebinding gianna
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  creationTimestamp: "2020-09-26T13:57:58Z"
  name: gianna
  resourceVersion: "3049"
  selfLink: /apis/rbac.authorization.k8s.io/v1/clusterrolebindings/gianna
  uid: 72b64a3b-5958-4cf8-8078-e5be2c55b25d
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: gianna
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: gianna
```

It links user `gianna` to same named *ClusterRole*:

```
k edit clusterrole gianna
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  creationTimestamp: "2020-09-26T13:57:55Z"
  name: gianna
  resourceVersion: "3038"
  selfLink: /apis/rbac.authorization.k8s.io/v1/clusterroles/gianna
  uid: b713c1cf-87e5-4313-808e-1a51f392adc0
rules:
- apiGroups:
  - ""
  resources:
  - secrets
  - configmaps
  - pods
  - namespaces
  verbs:
  - list
```

According to the task the user should never be able to read *Secrets* content. They verb `list` might indicate on first look that this is correct. We can also check using K8s User Impersonation:

```
➜ k auth can-i list secrets --as gianna
yes

➜ k auth can-i get secrets --as gianna
no
```

But let's have a closer look:

```
➜ k config use-context gianna@infra-prod
Switched to context "gianna@infra-prod".

➜ k -n security get secrets
NAME                  TYPE                                  DATA   AGE
default-token-gn455   kubernetes.io/service-account-token   3      20m
kubeadmin-token       Opaque                                1      20m
mysql-admin           Opaque                                1      20m
postgres001           Opaque                                1      20m
postgres002           Opaque                                1      20m
vault-token           Opaque                                1      20m

➜ k -n security get secret kubeadmin-token
Error from server (Forbidden): secrets "kubeadmin-token" is forbidden: User "gianna" cannot get resource "secrets" in
API group "" in the namespace "security"
```

Still all expected, but being able to list resources also allows to specify the format:

```
➜ k -n security get secrets -oyaml | grep password
    password: ekhHYW5lQUVTaVVxCg==
        {"apiVersion":"v1","data":{"password":"ekhHYW5lQUVTaVVxCg=="},"kind":"Secret","metadata":{"annotations":
{},"name":"kubeadmin-token","namespace":"security"},"type":"Opaque"}
          f:password: {}
    password: bWdFVlBSdEpEWHBFCg==
...
```

The user `gianna` is actually able to read *Secret* content. To prevent this we should remove the ability to list these:

```
k config use-context infra-prod # back to admin context

k edit clusterrole gianna
```

```
# kubectl edit clusterrole gianna
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  creationTimestamp: "2020-09-26T13:57:55Z"
  name: gianna
  resourceVersion: "4496"
  selfLink: /apis/rbac.authorization.k8s.io/v1/clusterroles/gianna
  uid: b713c1cf-87e5-4313-808e-1a51f392adc0
rules:
- apiGroups:
  - ""
  resources:
#  - secrets       # remove
  - configmaps
  - pods
  - namespaces
  verbs:
  - list
```

Part 2 - create additional RBAC rules

Let's talk a little about RBAC resources:

A *ClusterRole*|*Role* defines a set of permissions and **where it is available**, in the whole cluster or just a single *Namespace*.

A *ClusterRoleBinding*|*RoleBinding* connects a set of permissions with an account and defines **where it is applied**, in the whole cluster or just a single *Namespace*.

Because of this there are 4 different RBAC combinations and 3 valid ones:

1. *Role* + *RoleBinding* (available in single *Namespace*, applied in single *Namespace*)
2. *ClusterRole* + *ClusterRoleBinding* (available cluster-wide, applied cluster-wide)
3. *ClusterRole* + *RoleBinding* (available cluster-wide, applied in single *Namespace*)
4. *Role* + *ClusterRoleBinding* (**NOT POSSIBLE:** available in single *Namespace*, applied cluster-wide)

The user `gianna` should be able to create *Pods* and *Deployments* in three *Namespaces*. We can use number 1 or 3 from the list above. But because the task says: "The user might receive these exact permissions as well for other *Namespaces* in the future", we choose number 3 as it requires to only create one *ClusterRole* instead of three *Roles*.

```
k create clusterrole gianna-additional --verb=create --resource=pods --resource=deployments
```

This will create a *ClusterRole* like:

```
# kubectl create clusterrole gianna-additional --verb=create --resource=pods --resource=deployments
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  creationTimestamp: null
  name: gianna-additional
rules:
- apiGroups:
  - ""
  resources:
  - pods
  verbs:
  - create
- apiGroups:
  - apps
  resources:
  - deployments
  verbs:
  - create
```

Next the three bindings:

```
k -n security create rolebinding gianna-additional \
--clusterrole=gianna-additional --user=gianna

k -n restricted create rolebinding gianna-additional \
--clusterrole=gianna-additional --user=gianna

k -n internal create rolebinding gianna-additional \
--clusterrole=gianna-additional --user=gianna
```

Which will create *RoleBindings* like:

```
# k -n security create rolebinding gianna-additional --clusterrole=gianna-additional --user=gianna
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  creationTimestamp: null
  name: gianna-additional
  namespace: security
```

```yaml
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: gianna-additional
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: gianna
```

And we test:

```
➜ k -n default auth can-i create pods --as gianna
no

➜ k -n security auth can-i create pods --as gianna
yes

➜ k -n restricted auth can-i create pods --as gianna
yes

➜ k -n internal auth can-i create pods --as gianna
yes
```

Feel free to verify this as well by actually creating *Pods* and *Deployments* as user `gianna` through context `gianna@infra-prod`.

Preview Question 2

Use context: `kubectl config use-context infra-prod`

There is an existing Open Policy Agent + Gatekeeper policy to enforce that all *Namespaces* need to have label `security-level` set. Extend the policy constraint and template so that all *Namespaces* also need to set label `management-team`. Any new *Namespace* creation without these two labels should be prevented.

Write the names of all existing *Namespaces* which violate the updated policy into `/opt/course/p2/fix-namespaces`.

Answer:

We look at existing OPA constraints, these are implemeted using CRDs by Gatekeeper:

```
➜ k get crd
NAME                                                 CREATED AT
blacklistimages.constraints.gatekeeper.sh            2020-09-14T19:29:31Z
configs.config.gatekeeper.sh                         2020-09-14T19:29:04Z
constraintpodstatuses.status.gatekeeper.sh           2020-09-14T19:29:05Z
constrainttemplatepodstatuses.status.gatekeeper.sh   2020-09-14T19:29:05Z
constrainttemplates.templates.gatekeeper.sh          2020-09-14T19:29:05Z
requiredlabels.constraints.gatekeeper.sh             2020-09-14T19:29:31Z
```

So we can do:

```
➜ k get constraint
NAME                                                          AGE
blacklistimages.constraints.gatekeeper.sh/pod-trusted-images   10m

NAME                                                                  AGE
requiredlabels.constraints.gatekeeper.sh/namespace-mandatory-labels   10m
```

And check violations for the `namespace-mandatory-label` one, which we can do in the resource status:

```
➜ k describe requiredlabels namespace-mandatory-labels
Name:         namespace-mandatory-labels
Namespace:
Labels:       <none>
Annotations:  <none>
API Version:  constraints.gatekeeper.sh/v1beta1
Kind:         RequiredLabels
...
Status:
...
  Total Violations:  1
  Violations:
    Enforcement Action:  deny
    Kind:                Namespace
    Message:             you must provide labels: {"security-level"}
    Name:                sidecar-injector
Events:
```

We see one violation for *Namespace* "sidecar-injector". Let's get an overview over all *Namespaces*:

```
➜ k get ns --show-labels
NAME                 STATUS   AGE   LABELS
default              Active   21m   management-team=green,security-level=high
gatekeeper-system    Active   14m   admission.gatekeeper.sh/ignore=no-self-managing,control-plane=controller-
manager,gatekeeper.sh/system=yes,management-team=green,security-level=high
jeffs-playground     Active   14m   security-level=high
kube-node-lease      Active   21m   management-team=green,security-level=high
kube-public          Active   21m   management-team=red,security-level=low
kube-system          Active   21m   management-team=green,security-level=high
restricted           Active   14m   management-team=blue,security-level=medium
security             Active   14m   management-team=blue,security-level=medium
sidecar-injector     Active   14m   <none>
```

When we try to create a *Namespace* without the required label we get an OPA error:

```
➜ k create ns test
Error from server ([denied by namespace-mandatory-labels] you must provide labels: {"security-level"}): error when
creating "ns.yaml": admission webhook "validation.gatekeeper.sh" denied the request: [denied by namespace-mandatory-
labels] you must provide labels: {"security-level"}
```

Next we edit the constraint to add another required label:

```
k edit requiredlabels namespace-mandatory-labels
```

```
# kubectl edit requiredlabels namespace-mandatory-labels
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: RequiredLabels
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"constraints.gatekeeper.sh/v1beta1","kind":"RequiredLabels","metadata":{"annotations":
{},"name":"namespace-mandatory-labels"},"spec":{"match":{"kinds":[{"apiGroups":[""],"kinds":
["Namespace"]}]},"parameters":{"labels":["security-level"]}}}
  creationTimestamp: "2020-09-14T19:29:53Z"
  generation: 1
  name: namespace-mandatory-labels
  resourceVersion: "3081"
  selfLink: /apis/constraints.gatekeeper.sh/v1beta1/requiredlabels/namespace-mandatory-labels
  uid: 2a51a291-e07f-4bab-b33c-9b8c90e5125b
spec:
  match:
    kinds:
    - apiGroups:
      - ""
      kinds:
      - Namespace
  parameters:
    labels:
    - security-level
    - management-team # add
```

As we can see the constraint is using `kind: RequiredLabels` as template, which is a CRD created by Gatekeeper. Let's apply the change and see what happens (give OPA a minute to apply the changes internally):

```
➜ k describe requiredlabels namespace-mandatory-labels
...
  Violations:
    Enforcement Action:  deny
    Kind:                Namespace
    Message:             you must provide labels: {"management-team"}
    Name:                jeffs-playground
```

After the changes we can see that now another *Namespace* `jeffs-playground` is in trouble. Because that one only specifies one required label. But what about the earlier violation of *Namespace* `sidecar-injector`?

```
➜ k get ns --show-labels
NAME                 STATUS   AGE   LABELS
default              Active   21m   management-team=green,security-level=high
gatekeeper-system    Active   17m   admission.gatekeeper.sh/ignore=no-self-managing,control-plane=controller-
manager,gatekeeper.sh/system=yes,management-team=green,security-level=high
jeffs-playground     Active   17m   security-level=high
kube-node-lease      Active   21m   management-team=green,security-level=high
kube-public          Active   21m   management-team=red,security-level=low
kube-system          Active   21m   management-team=green,security-level=high
restricted           Active   17m   management-team=blue,security-level=medium
security             Active   17m   management-team=blue,security-level=medium
sidecar-injector     Active   17m   <none>
```

Namespace `sidecar-injector` should also be in trouble, but it isn't any longer. This doesn't seem right, it means we could still create *Namespaces* without any labels just like using `k create ns test`.

So we check the template:

```
➜ k get constrainttemplates
NAME              AGE
blacklistimages   20m
requiredlabels    20m

➜ k edit constrainttemplates requiredlabels
```

```
# kubectl edit constrainttemplates requiredlabels
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
...
spec:
  crd:
    spec:
      names:
        kind: RequiredLabels
      validation:
        openAPIV3Schema:
          properties:
            labels:
              items: string
              type: array
  targets:
  - rego: |
      package k8srequiredlabels

      violation[{"msg": msg, "details": {"missing_labels": missing}}] {
        provided := {label | input.review.object.metadata.labels[label]}
        required := {label | label := input.parameters.labels[_]}
        missing := required - provided
        # count(missing) == 1 # WRONG
        count(missing) > 0
        msg := sprintf("you must provide labels: %v", [missing])
      }
    target: admission.k8s.gatekeeper.sh
```

In the rego script we need to change `count(missing) == 1` to `count(missing) > 0` . If we don't do this then the policy only complains if there is one missing label, but there can be multiple missing ones.

After waiting a bit we check the constraint again:

```
➜ k describe requiredlabels namespace-mandatory-labels
...
  Total Violations:  2
  Violations:
    Enforcement Action:  deny
    Kind:                Namespace
    Message:             you must provide labels: {"management-team"}
    Name:                jeffs-playground
    Enforcement Action:  deny
    Kind:                Namespace
    Message:             you must provide labels: {"security-level", "management-team"}
    Name:                sidecar-injector
Events:                  <none>
```

This looks better. Finally we write the *Namespace* names with violations into the required location:

```
# /opt/course/p2/fix-namespaces
sidecar-injector
jeffs-playground
```

Preview Question 3

Use context: `kubectl config use-context workload-stage`

A security scan result shows that there is an unknown miner process running on one of the *Nodes* in cluster3. The report states that the process is listening on port 6666. Kill the process and delete the binary.

Answer:

We have a look at existing *Nodes*:

```
➜ k get node
NAME               STATUS   ROLES                  AGE    VERSION
cluster3-master1   Ready    control-plane,master   109m   v1.21.4
cluster3-worker1   Ready    <none>                 105m   v1.21.4
```

First we check the master:

```
➜ ssh cluster3-master1

➜ root@cluster3-master1:~# netstat -plnt | grep 6666

➜ root@cluster3-master1:~#
```

Doesn't look like any process listening on this port. So we check the worker:

```
➜ ssh cluster3-worker1

➜ root@cluster3-worker1:~# netstat -plnt | grep 6666
tcp6       0      0 :::6666                 :::*        LISTEN      9591/system-atm
```

There we go! We could also use `lsof`:

```
➜ root@cluster3-worker1:~# lsof -i :6666
COMMAND    PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
system-at 9591 root    3u  IPv6  47760      0t0  TCP *:6666 (LISTEN)
```

Before we kill the process we can check the magic `/proc` directory for the full process path:

```
➜ root@cluster3-worker1:~# ls -lh /proc/9591/exe
lrwxrwxrwx 1 root root 0 Sep 26 16:10 /proc/9591/exe -> /bin/system-atm
```

So we finish it:

```
➜ root@cluster3-worker1:~# kill -9 9591

➜ root@cluster3-worker1:~# rm /bin/system-atm
```

Done.

CKS Tips Kubernetes 1.22

In this section we'll provide some tips on how to handle the CKS exam and browser terminal.

Knowledge

Pre-Knowledge

You should have your CKA knowledge up to date and be fast with kubectl, so we suggest to do:

- The [CKAD series with scenarios]() on Medium
- The [CKA series with scenarios]() on Medium

Knowledge

- Study all topics as proposed in the curriculum till you feel comfortable with all.
- Check our [CKS Exam Series]()
- Read the free Sysdig [Kubernetes Security Guide]()
- Also a nice read (though based on outdated k8s version) is the [Kubernetes Security book]() by Liz Rice
- Check out the [Cloud Native Security Whitepaper]()
- Great repository with many tips and sources: [walidshari]()

Approach

- Do 1 or 2 test session with this CKS Simulator. Understand the solutions and maybe try out other ways to achieve the same thing.
- Setup your aliases, be fast and breath `kubectl`

Content

- Be comfortable with changing the kube-apiserver in a kubeadm setup
- Be able to work with [AdmissionControllers]()
- Know how to create and use the [ImagePolicyWebhook]()
- Know how to use opensource tools [Falco](), [Sysdig](), [Tracee](), [Trivy]()

CKS Exam Info

Read the Curriculum

https://github.com/cncf/curriculum

Read the Handbook

https://docs.linuxfoundation.org/tc-docs/certification/lf-candidate-handbook

Read the important tips

https://docs.linuxfoundation.org/tc-docs/certification/important-instructions-cks

Read the FAQ

https://docs.linuxfoundation.org/tc-docs/certification/faq-cka-ckad-cks

Kubernetes documentation

Get familiar with the Kubernetes documentation and be able to use the search. You can have one browser tab open with one of the allowed links (check the official docs for updated list):

- https://kubernetes.io/docs
- https://github.com/kubernetes
- https://kubernetes.io/blog
- https://aquasecurity.github.io/trivy
- https://docs.sysdig.com
- https://falco.org/docs
- https://gitlab.com/apparmor/apparmor/-/wikis/Documentation

> ***NOTE:*** You can have the other tab open as a separate window, this is why a big screen is handy

Deprecated commands

Make sure to not depend on deprecated commands as they might stop working at any time. When you execute a deprecated `kubectl` command a message will be shown, so you know which ones to avoid.

The Test Environment / Browser Terminal

You'll be provided with a browser terminal which uses Ubuntu 20. The standard shells included with a minimal install of Ubuntu 20 will be available, including bash.

Laggin

There could be some lagging, definitely make sure you are using a good internet connection because your webcam and screen are uploading all the time.

Kubectl autocompletion and commands

Autocompletion is configured by default, as well as the `k` alias source and others:

`kubectl` with `k` alias and Bash autocompletion

`yq` and `jq` for YAML/JSON processing

`tmux` for terminal multiplexing

`curl` and `wget` for testing web services

`man` and man pages for further documentation

Copy & Paste

There could be issues copying text (like pod names) from the left task information into the terminal. Some suggested to "hard" hit or long hold `Cmd/Ctrl+C` a few times to take action. Apart from that copy and paste should just work like in normal terminals.

Percentages and Score

There are 15-20 questions in the exam and 100% of total percentage to reach. Each questions shows the % it gives if you solve it. Your results will be automatically checked according to the handbook. If you don't agree with the results you can request a review by contacting the Linux Foundation support.

Notepad & Skipping Questions

You have access to a simple notepad in the browser which can be used for storing any kind of plain text. It makes sense to use this for saving skipped question numbers and their percentages. This way it's possible to move some questions to the end. It might make sense to skip 2% or 3% questions and go directly to higher ones.

Contexts

You'll receive access to various different clusters and resources in each. They provide you the exact command you need to run to connect to another cluster/context. But you should be comfortable working in different namespaces with `kubectl`.

Your Desktop

You are allowed to have multiple monitors connected and have to share every monitor with the proctor. Having one large screen definitely helps as you're only allowed **one** application open (Chrome Browser) with two tabs, one terminal and one k8s docs.

> ***NOTE:*** You can have the other tab open as a separate window, this is why a big screen is handy

The questions will be on the left (default maybe ~30% space), the terminal on the right. You can adjust the size of the split though to your needs in the real exam.

If you use a laptop you could work with lid closed, external mouse+keyboard+monitor attached. Make sure you also have a webcam+microphone working.

You could also have both monitors, laptop screen and external, active. You might be asked that your webcam points straight into your face. So using an external screen and your laptop webcam could not be accepted. Just keep that in mind.

You have to be able to move your webcam around in the beginning to show your whole room and desktop. Have a clean desk with only the necessary on it. You can have a glass/cup with water without anything printed on.

In the end you should feel very comfortable with your setup.

CKS clusters

In the CKS exam you'll get access to as many clusters as you have questions, each will be solved in its own cluster. This is great because you cannot interfere with other tasks by breaking one. Every cluster will have one master and one worker node.

Browser Terminal Setup

It should be considered to spend ~1 minute in the beginning to setup your terminal. In the real exam the vast majority of questions will be done from the main terminal. For few you might need to ssh into another machine. Just be aware that configurations to your shell will not be transferred in this case.

Minimal Setup

Alias

The alias `k` for `kubectl` will be configured together with autocompletion. In case not you can configure it using this link.

Vim

Create the file `~/.vimrc` with the following content:

```
set tabstop=2
set expandtab
set shiftwidth=2
```

The `expandtab` make sure to use spaces for tabs. Memorize these and just type them down. You can't have any written notes with commands on your desktop etc.

Optional Setup

Fast dry-run output

```
export do="--dry-run=client -o yaml"
```

This way you can just run `k run pod1 --image=nginx $do`. Short for "dry output", but use whatever name you like.

Fast pod delete

```
export now="--force --grace-period 0"
```

This way you can run `k delete pod1 $now` and don't have to wait for ~30 seconds termination time.

Persist bash settings

You can store aliases and other setup in `~/.bashrc` if you're planning on using different shells or `tmux`.

Be fast

Use the `history` command to reuse already entered commands or use even faster history search through **Ctrl r** .

If a command takes some time to execute, like sometimes `kubectl delete pod x`. You can put a task in the background using **Ctrl z** and pull it back into foreground running command `fg`.

You can delete *pods* fast with:

```
k delete pod x --grace-period 0 --force

k delete pod x $now # if export from above is configured
```

Vim

Be great with vim.

Toggle vim line numbers

When in `vim` you can press **Esc** and type `:set number` or `:set nonumber` followed by **Enter** to toggle line numbers. This can be useful when finding syntax errors based on line - but can be bad when wanting to mark&copy by mouse. You can also just jump to a line number with **Esc** `:22` + **Enter**.

Copy&paste

Get used to copy/paste/cut with vim:

```
Mark lines: Esc+V (then arrow keys)
Copy marked lines: y
Cut marked lines: d
Past lines: p or P
```

Indent multiple lines

In case not defined in `.vimrc`, to indent multiple lines press **Esc** and type `:set shiftwidth=2`.

First mark multiple lines using `Shift v` and the up/down keys. Then to indent the marked lines press `>` or `<`. You can then press `.` to repeat the action.

Split terminal screen

By default `tmux` is installed and can be used to split your one terminal into multiple. **But** just do this if you know your shit, because scrolling is different and copy&pasting might be weird.

https://www.hamvocke.com/blog/a-quick-and-easy-guide-to-tmux