Killer Shell - CKS CKA CKAD Simulator

Web Clip

CKS Simulator Kubernetes 1.20

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. Set these:

```
alias k=kubectl

export do="--dry-run=client -o yaml" # like short for dry output. use whatever you like
```

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 of the CKS Simulator.

Question 0 | Instructions

You should avoid using deprecated `kubectl` commands as these might not work in the exam.

There are three Kubernetes clusters and 7 nodes in total:

cluster1-master1
cluster1-worker1
cluster1-worker2
cluster2-master1
cluster2-worker1
cluster3-master1
cluster3-worker1

Rules

You're only allowed to have one other browser tab open with

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

Notes

You have a notepad (top right) where you can store plain text. This is useful to store questions you skipped and might try again at the end.

Difficulty

This simulator is more difficult than the real certification. We think this gives you a greater learning effect and also confidence to score in the real exam. Most of the simulator scenarios require good amount of work and can be considered "hard". In the real exam you will also face these "hard" scenarios, just less often.

SSH Access

As the k8s@terminal user you can connect via ssh to every node, like `ssh cluster1-master1`. Using `kubectl` as root user on a master node you can connect to the api-server of just that cluster.

File system

User k8s@terminal has root permissions using sudo should you face permission issues. Whenever you're asked to write or edit something in `/opt/course/...` it should be done so in your main terminal and not on any of the master or worker nodes.

K8s contexts

Using `kubectl` from k8s@terminal you can reach the api-servers of all available clusters through different pre-configured contexts. The command to switch to the correct Kubernetes context will be listed on top of every question when needed.

Ctrl/Cmd-F Search

Do not use the browser search via Ctrl-F or Cmd-F beause this will render the brower terminal unusuable. If this happened you can simply reload your browser page.

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/wQCMAAwHwYDVR0jBBgwFoAUPrspZIWR7YMN8yT5DF3s/LypxPQw
```

```
DQYJKoZIhvcNAQELBQADggEBAIDq0Zt77gXI1s+uW46zBw4mIWgAlBLl2QqCuwmV
kd86eH5bDOFCtWlb6vGdcKPdFccHh8Z6z2LjjLu6UoiGUdIJaALhbYNJiXXi/7cf
M7sqNOxpxQ5X5hyvOBYD1W7d/EzPHV/lcbXPUDYFHNqBYs842LWSTlPQioDpupXp
FFUQPxsenNXDa4TbmaRvnK2jka0yXcqdiXuIteZZovp/IgNkfmx2Ld4/Q+Xlnscf
CFtWbjRa/OW/3EW/ghQ7xtC7bgcOHJesoiTZPCZ+dfKuUfH6d1qxgj6JwtOHtyEf
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:

> Find a Pod running image `nginx` which creates unwanted package management processes inside its container.
> 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
Oct  9 21:46:55 ubuntu-bionic falco: Falco version 0.26.1 (driver
version 2aa88dcf6243982697811df4c1b484bcbe9488a2)
Oct  9 21:46:55 ubuntu-bionic falco: Falco initialized with
configuration file /etc/falco/falco.yaml
...
```

Yep, something going on in there. Let's investigate the first offending Pod:

```
→ root@cluster1-worker1:~# cat /var/log/syslog | grep falco | grep
nginx | grep process
Oct  9 23:14:49 ubuntu-bionic falco: 23:14:49.070029616: Error Package
management process launched in container (user=root user_loginuid=-1
```

```
command=apk container_id=b51765aea1cb container_name=k8s_nginx_webapp-
6b797d5b65-7mxz7_team-blue_a0221829-d3ee-4c72-bfac-496a16e1c3fc_0
image=nginx:1.19.2-alpine)
...
```

And for the second Pod:

```
→ root@cluster1-worker1:~# cat /var/log/syslog | grep falco | grep
httpd | grep passwd
Oct  9 23:26:19 ubuntu-bionic falco: 23:26:19.950348409: 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/passwdJPAmlk program=sed
gparent=<NA> ggparent=<NA> gggparent=<NA> container_id=d9ce1e213996
image=httpd)

→ root@cluster1-worker1:~# docker ps | grep d9ce1e213996
d9ce1e213996        35ab485181ce                            "httpd-
foreground"       3 minutes ago       Up 3 minutes
       k8s_httpd_rating-service-5c54c948c9-fnvn2_team-purple_02083909-
c6f5-4f87-bab3-fae9de2fc55a_0
```

Use Falco from command line

We can also use Falco directly from command line, even if the service is disabled, like this:

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

→ root@cluster1-worker1:~# falco
Sat Dec  5 19:58:29 2020: Falco version 0.26.1 (driver version
2aa88dcf6243982697811df4c1b484bcbe9488a2)
Sat Dec  5 19:58:29 2020: Falco initialized with configuration file
/etc/falco/falco.yaml
Sat Dec  5 19:58:29 2020: Loading rules from file
/etc/falco/falco_rules.yaml:
Sat Dec  5 19:58:29 2020: Loading rules from file
/etc/falco/falco_rules.local.yaml:
Sat Dec  5 19:58:30 2020: Loading rules from file
/etc/falco/k8s_audit_rules.yaml:
Sat Dec  5 19:58:30 2020: Starting internal webserver, listening on
port 8765

19:58:34.436913858: Error Package management process launched in
container (user=root user_loginuid=-1 command=apk
```

```
container_id=fd6a98d42973 container_name=k8s_nginx_webapi-5fcb69b746-
gtx8q_team-blue_d5e9178c-60fb-43e5-af89-3b8a579614ef_0
image=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 should look 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]
```

And change it 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"

20:23:14.395725592: Error Package management process launched in
container 20:23:14.395725592,fd6a98d42973,k8s_nginx_webapi-5fcb69b746-
gtx8q_team-blue_d5e9178c-60fb-43e5-af89-3b8a579614ef_0,root
20:23:19.566382518: Error Package management process launched in
container 20:23:19.566382518,fd6a98d42973,k8s_nginx_webapi-5fcb69b746-
gtx8q_team-blue_d5e9178c-60fb-43e5-af89-3b8a579614ef_0,root
20:23:24.502379334: Error Package management process launched in
container 20:23:24.502379334,fd6a98d42973,k8s_nginx_webapi-5fcb69b746-
gtx8q_team-blue_d5e9178c-60fb-43e5-af89-3b8a579614ef_0,root
20:23:29.802281942: Error Package management process launched in
container 20:23:29.802281942,fd6a98d42973,k8s_nginx_webapi-5fcb69b746-
gtx8q_team-blue_d5e9178c-60fb-43e5-af89-3b8a579614ef_0,root
20:23:34.560697766: Error Package management process launched in
container 20:23:34.560697766,fd6a98d42973,k8s_nginx_webapi-5fcb69b746-
gtx8q_team-blue_d5e9178c-60fb-43e5-af89-3b8a579614ef_0,root
20:23:39.840817808: Error Package management process launched in
```

```
container 20:23:39.840817808,fd6a98d42973,k8s_nginx_webapi-5fcb69b746-
gtx8q_team-blue_d5e9178c-60fb-43e5-af89-3b8a579614ef_0,root
20:23:44.409411640: Error Package management process launched in
container 20:23:44.409411640,fd6a98d42973,k8s_nginx_webapi-5fcb69b746-
gtx8q_team-blue_d5e9178c-60fb-43e5-af89-3b8a579614ef_0,root
20:23:49.588266119: Error Package management process launched in
container 20:23:49.588266119,fd6a98d42973,k8s_nginx_webapi-5fcb69b746-
gtx8q_team-blue_d5e9178c-60fb-43e5-af89-3b8a579614ef_0,root
20:23:54.487145960: Error Package management process launched in
container 20:23:54.487145960,fd6a98d42973,k8s_nginx_webapi-5fcb69b746-
gtx8q_team-blue_d5e9178c-60fb-43e5-af89-3b8a579614ef_0,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
20:23:14.395725592,fd6a98d42973,k8s_nginx_webapi-5fcb69b746-gtx8q_team-
blue_d5e9178c-60fb-43e5-af89-3b8a579614ef_0,root
20:23:19.566382518,fd6a98d42973,k8s_nginx_webapi-5fcb69b746-gtx8q_team-
blue_d5e9178c-60fb-43e5-af89-3b8a579614ef_0,root
20:23:24.502379334,fd6a98d42973,k8s_nginx_webapi-5fcb69b746-gtx8q_team-
blue_d5e9178c-60fb-43e5-af89-3b8a579614ef_0,root
20:23:29.802281942,fd6a98d42973,k8s_nginx_webapi-5fcb69b746-gtx8q_team-
blue_d5e9178c-60fb-43e5-af89-3b8a579614ef_0,root
20:23:34.560697766,fd6a98d42973,k8s_nginx_webapi-5fcb69b746-gtx8q_team-
blue_d5e9178c-60fb-43e5-af89-3b8a579614ef_0,root
20:23:39.840817808,fd6a98d42973,k8s_nginx_webapi-5fcb69b746-gtx8q_team-
blue_d5e9178c-60fb-43e5-af89-3b8a579614ef_0,root
20:23:44.409411640,fd6a98d42973,k8s_nginx_webapi-5fcb69b746-gtx8q_team-
blue_d5e9178c-60fb-43e5-af89-3b8a579614ef_0,root
20:23:49.588266119,fd6a98d42973,k8s_nginx_webapi-5fcb69b746-gtx8q_team-
blue_d5e9178c-60fb-43e5-af89-3b8a579614ef_0,root
20:23:54.487145960,fd6a98d42973,k8s_nginx_webapi-5fcb69b746-gtx8q_team-
blue_d5e9178c-60fb-43e5-af89-3b8a579614ef_0,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 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.

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-6b797d5b65-7mxz7                      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-5c54c948c9-fnvn2              1/1
    Running

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

Job done.

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:

```
# /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 `docker-log-hacker` in Namespace `team-red` which mounts `/var/lib/docker` as a hostPath volume on the Node where its running. This means that the Pods can for example read all Docker container logs which are running on the same Node.

You're asked to forbid this behavior by:

> Enabling Admission Plugin `PodSecurityPolicy` in the apiserver
> Creating a PodSecurityPolicy named `psp-mount` which allows hostPath volumes only for directory `/tmp`
> Creating a ClusterRole named `psp-mount` which allows to use the new PSP
> 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 `docker-log-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 `docker-log-hacker` is capable of:

```
➜ k -n team-red get pod | grep hacker
docker-log-hacker-79cd6c58d5-2g4zv      1/1     Running   0
88s

➜ k -n team-red describe pod docker-log-hacker-79cd6c58d5-2g4zv
Name:         docker-log-hacker-79cd6c58d5-2g4zv
Namespace:    team-red
Priority:     0
Node:         cluster1-worker1/192.168.100.12
Start Time:   Mon, 28 Sep 2020 09:17:44 +0000
Labels:       app=docker-log-hacker
              pod-template-hash=79cd6c58d5
Annotations:  <none>
Status:       Running
IP:           10.44.0.19
IPs:
  IP:           10.44.0.19
Controlled By:  ReplicaSet/docker-log-hacker-79cd6c58d5
Containers:
  bash:
...
    Command:
      sh
```

```
      -c
      while true; do sleep 1d; done
...
    Mounts:
      /dockerlogs from dockerlogs (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-
9c2wf (ro)
...
Volumes:
  dockerlogs:
    Type:          HostPath (bare host directory volume)
    Path:          /var/lib/docker
...
```

We see it mounts `/var/lib/docker` from the Node where it's running on, what does this mean?

```
➜ k -n team-red exec -it docker-log-hacker-79cd6c58d5-2g4zv -- sh

➜ # cd /dockerlogs/containers/

➜ /dockerlogs/containers # ls
025c21a3e9f466550d15d06620318dc2a4dc5bd09562b3e30169fde56162f6ba
092cb84e4cb17f537aaf50a78f6e3d0737b90d78ff49c03aa88547daf66359dd
17f00b3e25313b05c1f642831f25e388852664699d8a5be26315cb14642016de
...

➜ /dockerlogs/containers # cd
47ac9c0a75aff011e865ebfb7b1695bddc891fccf59e6eafddb06032d44c6d5b/

➜
/dockerlogs/containers/47ac9c0a75aff011e865ebfb7b1695bddc891fccf59e6eaf
ddb06032d44c6d5b # head 47ac9c0a75aff011e865e
bfb7b1695bddc891fccf59e6eafddb06032d44c6d5b-json.log

{"log":"Mon Sep 28 08:55:14 UTC 2020\n","stream":"stdout","time":"2020-
09-28T08:55:14.431789056Z"}
{"log":"uid=0(root) gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(flop
py),20(dialout),26(tape),27(video)\n","stream":"stdout","time":"2020-
09-28T08:55:14.441553845Z"}
{"log":"\n","stream":"stdout","time":"2020-09-28T08:55:14.442354173Z"}
{"log":"Mon Sep 28 08:55:15 UTC 2020\n","stream":"stdout","time":"2020-
09-28T08:55:15.446719832Z"}
...
```

We can see that this Pod can access Docker logs 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
```

```
# /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

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:
```

```
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
```

```
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 docker-log-hacker
deployment.apps/docker-log-hacker restarted

➜ k -n team-red describe deploy docker-log-hacker
Name:                   docker-log-hacker
Namespace:              team-red
...
Replicas:               1 desired | 0 updated | 0 total | 0 available |
2 unavailable
...
Pod Template:
  Labels:       app=docker-log-hacker
  Annotations:  kubectl.kubernetes.io/restartedAt: 2020-09-28T11:08:18Z
  Containers:
   bash:
...
  Volumes:
   dockerlogs:
    Type:          HostPath (bare host directory volume)
    Path:          /var/lib/docker
```

```
    HostPathType:
Conditions:
  Type             Status  Reason
  ----             ------  ------
  Available        False   MinimumReplicasUnavailable
  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}'

docker-log-hacker-6bdfbf8546-" is forbidden: PodSecurityPolicy: unable
to admit pod: [spec.volumes[0].hostPath.pathPrefix: Invalid value:
"/var/lib/docker": 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 docker-log-hacker
```

```
# kubectl -n team-red edit deploy docker-log-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: /dockerlogs
          name: dockerlogs
...
      volumes:
      - hostPath:
          path: /tmp                          # change
          type: ""
```

And we should see it running:

```
➜ k -n team-red get pod -l app=docker-log-hacker
NAME                                 READY   STATUS    RESTARTS   AGE
docker-log-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=docker-log-hacker
...
Annotations:  kubernetes.io/psp: psp-mount
...
```

PodSecurityPolicies can be hard to come around at first, but once done they're a powerful part in the security tool box.

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:

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

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

The permissions of the kubelet configuration

`/var/lib/kubelet/config.yaml`
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
```

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

Edit the corresponding line:

```
# /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)
```

```
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
```

```
/var/lib/kubelet/config.yaml" 32

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
```

```
# /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
`f417c0555bc0167355589dd1afe23be9bf909bf98312b1025f12015d1b58a1c62c9908c00`
`67a7764fa35efdac7016a9efa8711a44425dd6692906a7c283f032c`

kube-controller-manager
`60100cc725e91fe1a949e1b2d0474237844b5862556e25c2c655a33boa8225855ec5ee22f`
`a4927e6c46a60d43a7c4403a27268f96fbb726307d1608b44f38a60`

kube-proxy
`52f9d8ad045f8eee1d689619ef8ceef2d86d50c75a6a332653240d7ba5b2a114aca056d9e`
`513984ade24358c9662714973c1960c62a5cb37dd375631c8a614c6`

`513984ade24358c9662714973c1960c62a5cb37dd375631c8a614c6`

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
f417c0555bc0167355589dd1afe23be9bf909bf98312b1025f12015d1b58a1c62c9908c0067a7764fa35efdac7016a9efa8711a44425dd6692906a7c283f032c  kube-apiserver
```

Looking good, next:

```
➜ sha512sum kube-controller-manager
60100cc725e91fe1a949e1b2d0474237844b5862556e25c2c655a33b0a8225855ec5ee22fa4927e6c46a60d43a7c4403a27268f96fbb726307d1608b44f38a60  kube-controller-manager
```

Okay, next:

```
➜ sha512sum kube-proxy
52f9d8ad045f8eee1d689619ef8ceef2d86d50c75a6a332653240d7ba5b2a114aca056d9e513984ade24358c9662714973c1960c62a5cb37dd375631c8a614c6  kube-proxy
```

Also good, and finally:

```
➜ sha512sum kubelet
7b720598e6a3483b45c537b57d759e3e82bc5c53b3274f681792f62e941019cde3d51a7f9b55158abf3810d506146bc0aa7cf97b36f27f341028a54431b335be  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
60100cc725e91fe1a949e1b2d0474237844b5862556e25c2c655a33b0a8225855ec5ee2
2fa4927e6c46a60d43a7c4403a27268f96fbb726307d1608b44f38a60
60100cc725e91fe1a949e1b2d0474237844b5862556e25c2c655a33boa8225855ec5ee2
2fa4927e6c46a60d43a7c4403a27268f96fbb726307d1608b44f38a60
```

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

```
➜ cat compare | uniq
60100cc725e91fe1a949e1b2d0474237844b5862556e25c2c655a33b0a8225855ec5ee2
2fa4927e6c46a60d43a7c4403a27268f96fbb726307d1608b44f38a60
60100cc725e91fe1a949e1b2d0474237844b5862556e25c2c655a33boa8225855ec5ee2
2fa4927e6c46a60d43a7c4403a27268f96fbb726307d1608b44f38a60
```

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:  1
  Violations:
    Enforcement Action:  deny
    Kind:                Pod
    Message:             not trusted image!
    Name:                untrusted-68c4944d48-2hgt9
    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:

- Allow users to "skip login"
- Allow insecure access (HTTP without authentication)
- Allow basic authentication
- Allow access from outside the cluster

You are asked to make it more secure by:

You are asked to make it more secure by:

Deny users to "skip login"
Deny insecure access, enforce HTTPS (self signed certificates are ok for now)
Add the `--auto-generate-certificates` argument
Enforce authentication using a token (with possibility to use RBAC)
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   v1.19.1   192.168.100.11 ...
cluster1-worker1   Ready    <none>   36m   v1.19.1   192.168.100.12 ...
cluster1-worker2   Ready    <none>   34m   v1.19.1   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.

Install the AppArmor profile on Node `cluster1-worker1`. Connect using `ssh cluster1-worker1`.

Add label `security=apparmor` to the Node

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-w852p            0/1     CrashLoopBackOff   1
   118s   10.44.0.18   cluster1-worker1   <none>           <none>

➜ k logs apparmor-85c65645dc-w852p
/docker-entrypoint.sh: No files found in /docker-entrypoint.d/,
skipping configuration
/docker-entrypoint.sh: 13: /docker-entrypoint.sh: cannot create
/dev/null: Permission denied
2020/09/26 18:14:11 [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 Docker container:

```
➜ ssh cluster1-worker1

➜ root@cluster1-worker1:~# docker ps -a | grep apparmor
41f014a9e7a8        7e4d58f0e5f3                            "/docker-
entrypoint.…"   About a minute ago   Exited (1) About a minute ago
                  k8s_c1_apparmor-85c65645dc-w852p_default_3e209d37-
74ee-43c5-8a5e-b61a683f068c_6
c79fe47d5a78        k8s.gcr.io/pause:3.2                    "/pause"
            7 minutes ago        Up 7 minutes
             k8s_POD_apparmor-85c65645dc-w852p_default_3e209d37-74ee-
43c5-8a5e-b61a683f068c_0

➜ root@cluster1-worker1:~# docker inspect 41f014a9e7a8 | grep -i
profile
        "AppArmorProfile": "very-secure",
```

We need to use `docker ps -a` to also show stopped containers. Then `docker 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-w852p > /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.

The `cluster1-worker2` kubelet uses containerd instead of docker. Write the two arguments the kubelet has been configured with to use containerd into `/opt/course/10/arguments`.

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 can see that worker2 runs using containterd:

```
➜ k get node -o wide
NAME               STATUS   ROLES    AGE   VERSION ... CONTAINER-
RUNTIME
cluster1-master1   Ready    master   9h    v1.19.1 ... docker://19.3.6
cluster1-worker1   Ready    <none>   9h    v1.19.1 ... docker://19.3.6
cluster1-worker2   Ready    <none>   9h    v1.19.1 ...
containerd://1.3.3
```

First we ssh into the worker node and optionally check if containerd and runsc are installed and configured as it as described in the task:

```
➜ ssh cluster1-worker2

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

➜ root@cluster1-worker2:~# service containerd status
● containerd.service - containerd container runtime
   Loaded: loaded (/lib/systemd/system/containerd.service; enabled;
vendor preset: enabled)
   Active: active (running) since Thu 2020-09-03 15:58:22 UTC; 2min 36s
ago
```

```
    ...

→ root@cluster1-worker2:~# cat /etc/containerd/config.toml
disabled_plugins = ["restart"]
[plugins.linux]
  shim_debug = true
[plugins.cri.containerd.runtimes.runsc]
  runtime_type = "io.containerd.runsc.v1"
```

Looking good. Next we check the arguments of the kubelet.

```
# defines how the kubelet is started
vim /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
```

We see that it references file `/etc/default/kubelet`.

```
# /etc/default/kubelet
KUBELET_EXTRA_ARGS="--container-runtime remote --container-runtime-
endpoint unix:///run/containerd/containerd.sock"
```

And we write these into the required location on our main terminal:

```
# /opt/course/10/arguments
--container-runtime remote
--container-runtime-endpoint unix:///run/containerd/containerd.sock
```

For the next requirement it's 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
```

```
# 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
```

```
# 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-d5193d0f2aa32b8
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
```

```
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 >
```

```
→ echo cEVuRXRSYVRpT24tdEV2VGVzCg== | 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.

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`

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
```

```
# 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
```

```
# 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:

```
# 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 does restrict already 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
READINESS GATES
collector1-59ddbd6c7f-ffjjv    ...  cluster1-worker1   <none>
<none>
collector1-59ddbd6c7f-p9qqz    ...  cluster1-worker1   <none>
<none>
collector2-7b6868b5dc-h6zxx    ...  cluster1-worker1   <none>
<none>
collector3-77b7c5bf47-5hgcb    ...  cluster1-worker1   <none>
<none>
collector3-77b7c5bf47-rswrl    ...  cluster1-worker1   <none>
<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:~# docker ps | grep collector1
3e07aee08a48        registry.killer.sh:5000/collector1
"./collector1-process"   14 seconds ago       Up 13 seconds
              k8s_c_collector1-59ddbd6c7f-vvwgt_team-yellow_099822ad-
2dfc-4963-bfc7-d806e00c0daf_0
...
```

Deployment `collector1` has two replicas, and we can see that the processes execute `./collector1-process`. We can find the process PIDs:

```
➜ root@cluster1-worker1:~# ps aux | grep collector1-process
root     10991  0.0  0.0   2412   760 ?        Ssl  22:41   0:00
./collector1-process
root     11150  0.0  0.0   2412   756 ?        Ssl  22:41   0:00
./collector1-process
```

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

```
➜ root@cluster1-worker1:~# strace -p 10991
strace: Process 10991 attached
restart_syscall(<... resuming interrupted futex ...>) = -1 ETIMEDOUT
(Connection timed out)
futex(0x4ad5d0, FUTEX_WAKE, 1)          = 1
kill(666, SIGTERM)                      = -1 ESRCH (No such process)
futex(0xc420030948, FUTEX_WAKE, 1)      = 1
futex(0x4afe80, FUTEX_WAIT, 0, {tv_sec=0, tv_nsec=999998945}) = -1
ETIMEDOUT (Connection timed out)
...
```

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:~# docker ps | grep collector2
60531737cb83        registry.killer.sh:5000/collector2
"./collector2-process"   4 minutes ago       Up 4 minutes
             k8s_c_collector2-7b6868b5dc-
...

➜ root@cluster1-worker1:~# ps aux | grep collector2-process
root     10438  0.0  0.0   2420   764 ?        Ssl  20:19   0:00
./collector2-process
root     26442  0.0  0.0  14856  1000 pts/0    S+   21:13   0:00 grep -
-color=auto collector2-process

➜ root@cluster1-worker1:~# strace -p 10438
strace: Process 11080 attached
restart_syscall(<... resuming interrupted futex ...>) = -1 ETIMEDOUT
(Connection timed out)
futex(0x4af3d0, FUTEX_WAKE, 1)          = 1
futex(0x4af2f0, FUTEX_WAKE, 1)          = 1
futex(0xc420030548, FUTEX_WAKE, 1)      = 1
futex(0x4b1c80, FUTEX_WAIT, 0, {tv_sec=0, tv_nsec=999999578}) = -1
ETIMEDOUT (Connection timed out)
...
```

Looks alright. What about `collector3`:

```
➜ root@cluster1-worker1:~# docker ps | grep collector3
4db52d9aa69e        registry.killer.sh:5000/collector3
"./collector3-process"   6 minutes ago       Up 6 minutes
```

```
./collector3-process    0 minutes ago       Up 0 minutes
            k8s_c_collector3-77b7c5bf47-
...

➜ root@cluster1-worker1:~# ps aux | grep collector3-process
root     10915  0.0  0.0   2428   756 ?        Ssl  22:41   0:00
./collector3-process
root     11135  0.0  0.0   2428   760 ?        Ssl  22:41   0:00
./collector3-process
root     13374  0.0  0.1  14856  1104 pts/0    S+   22:49   0:00 grep -
-color=auto collector3-process

➜ root@cluster1-worker1:~# strace -p 10915
strace: Process 10915 attached
restart_syscall(<... resuming interrupted futex ...>) = -1 ETIMEDOUT
(Connection timed out)
futex(0x4b13d0, FUTEX_WAKE, 1)          = 1
futex(0x4b12f0, FUTEX_WAKE, 1)          = 1
futex(0xc420030548, FUTEX_WAKE, 1)      = 1
futex(0x4b3c80, FUTEX_WAIT, 0, {tv_sec=0, tv_nsec=999999504}) = -1
ETIMEDOUT (Connection timed out)
```

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
```

```yaml
# 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:

- Changing the base image to `alpine:3.12`
- Not installing `curl`
- Updating `nginx` to use the version constraint `>=1.18.0`
- 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
sudo docker build -t registry.killer.sh:5000/image-verify:v2 .
sudo docker run registry.killer.sh:5000/image-verify:v2 # to test your
changes
sudo docker 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(flop
py),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$ sudo docker build -t
registry.killer.sh:5000/image-verify:v2 .
Sending build context to Docker daemon  3.072kB
...
Successfully built a5df16d42c5b
Successfully tagged registry.killer.sh:5000/image-verify:v2
```

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

```
➜ :/opt/course/16/image$ sudo docker run
registry.killer.sh:5000/image-verify:v2
Fri Sep 25 21:02:09 UTC 2020
uid=101(myuser) gid=102(myuser) groups=102(myuser)
```

Looking good, so we push:

```
➜ :/opt/course/16/image$ sudo docker push
registry.killer.sh:5000/image-verify:v2
The push refers to repository [registry.killer.sh:5000/image-verify]
bf21d6611c7c: Layer already exists
82eb465441ab: Layer already exists
f88b13f57e3a: Pushed
32099b2fa646: Pushed
50644c29ef5a: Pushed
v2: digest:
sha256:867c1fded95faeec9e73404e822f6ed001b83163bd1e86f8945e8c00a758fdae
size: 1362
```

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:

From Secret resources, level Metadata
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`.

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
```

```
# /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
```

```
# /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:

```
# /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# 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:


```
{
  "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.

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 | vim -
```


```
{
  "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",
        "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
```

```
→ 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` directoy 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.

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:

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 `1.19.6`. Update it to `1.20.1` 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    master   13m   v1.19.6
cluster3-worker1   Ready    <none>   11m   v1.19.6
```

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:"19",
GitVersion:"v1.19.4",
GitCommit:"d360454c9bcd1634cf4cc52d1867af5491dc9c5f",
GitTreeState:"clean", BuildDate:"2020-11-11T13:15:05Z",
GoVersion:"go1.15.2", Compiler:"gc", Platform:"linux/amd64"}

➜ root@cluster3-master1:~# kubelet --version
Kubernetes v1.18.6
```

Install the wanted `kubeadm` version:

```
root@cluster3-master1:~# apt-get install kubeadm=1.20.1-00
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

```
kubeadm is already the newest version (1.20.1-00).
0 upgraded, 0 newly installed, 0 to remove and 2 not upgraded.
```

We can see that `kubeadm` is already installed in the wanted version, otherwise we would have to install it.

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...
...
```

And we apply it using:

```
➜ root@cluster3-master1:~# kubeadm upgrade apply v1.20.1
[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
"v1.20.1"
[upgrade/versions] Cluster version: v1.19.6
[upgrade/versions] kubeadm version: v1.20.1
...
[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.20.1".
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: v1.20.1
[upgrade/versions] kubeadm version: v1.20.1
[upgrade/versions] Latest stable version: v1.20.1
```

```
[upgrade/versions] Latest stable version: v1.20.1
[upgrade/versions] Latest version in the v1.20 series: v1.20.1
[upgrade/versions] Latest version in the v1.20 series: v1.20.1
```

Control Plane kubelet and kubectl

```
→ root@cluster3-master1:~# apt-get update

→ root@cluster3-master1:~# apt-get install kubelet=1.20.1-00
kubectl=1.20.1-00
...
Preparing to unpack .../kubectl_1.20.1-00_amd64.deb ...
Unpacking kubectl (1.20.1-00) over (1.19.6-00) ...
Preparing to unpack .../kubelet_1.20.1-00_amd64.deb ...
Unpacking kubelet (1.20.1-00) over (1.19.6-00) ...
Setting up kubelet (1.20.1-00) ...
Setting up kubectl (1.20.1-00) ...
→ 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
21h   v1.20.1
cluster3-worker1   Ready                      <none>
21h   v1.19.6
```

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   21h   v1.20.1
cluster3-worker1   Ready    <none>                 21h   v1.19.6
```

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
...

➜ root@cluster3-worker1:~# apt-get install kubeadm=1.20.1-00
Reading package lists... Done
Building dependency tree
Reading state information... Done
kubeadm is already the newest version (1.20.1-00).
0 upgraded, 0 newly installed, 0 to remove and 2 not upgraded.

➜ 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=1.20.1-00
kubectl=1.20.1-00
...
Preparing to unpack .../kubectl_1.20.1-00_amd64.deb ...
Unpacking kubectl (1.20.1-00) over (1.19.6-00) ...
Preparing to unpack .../kubelet_1.20.1-00_amd64.deb ...
Unpacking kubelet (1.20.1-00) over (1.19.6-00) ...
Setting up kubelet (1.20.1-00) ...
Setting up kubectl (1.20.1-00) ...

➜ 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
21h   v1.20.1
cluster3-worker1   Ready,SchedulingDisabled   <none>
21h   v1.20.1
```

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   21h   v1.20.1
cluster3-worker1   Ready    <none>                 21h   v1.20.1
```

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-`

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
```

```
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@sSwOrd                # 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
```