

CKS : Certified Kubernetes Security Specialist

I) Understanding the Kubernetes Attack Surface

a. The Attack

Il existe plusieurs façons d'attaquer une application déployée sur un cluster Kubernetes, ci-dessous certaines commandes permettant d'attaquer une application de vote afin d'en modifier les résultats du vote.

Tout ce que le hacker dispose et a besoin à l'initial ce sont les noms de domaine des différentes applications :

- www.vote.com
- www.result.com

A partir de ces noms de domaines, il essayera d'avoir l'adresse IP du serveur en faisant un ping afin d'identifier l'adresse IP répondante. (> ping www.vote.com)

```
➜ ping www.vote.com
PING www.vote.com (104.21.63.124): 56 data bytes
64 bytes from 104.21.63.124: icmp_seq=0 ttl=52 time=80.821 ms
64 bytes from 104.21.63.124: icmp_seq=1 ttl=52 time=80.927 ms
64 bytes from 104.21.63.124: icmp_seq=2 ttl=52 time=80.695 ms
64 bytes from 104.21.63.124: icmp_seq=3 ttl=52 time=80.819 ms
```

```
➜ ping www.result.com
PING www.result.com (104.21.63.124): 56 data bytes
64 bytes from 104.21.63.124: icmp_seq=0 ttl=52 time=81.144 ms
64 bytes from 104.21.63.124: icmp_seq=1 ttl=52 time=80.919 ms
64 bytes from 104.21.63.124: icmp_seq=2 ttl=52 time=81.024 ms
64 bytes from 104.21.63.124: icmp_seq=3 ttl=52 time=80.878 ms
```

Une fois les ping fait, il remarquera en effet que les deux applications sont hébergées sur le même serveur d'adresse IP : 104.21.63.124

Après cela, il cherchera à avoir la liste de tous les ports ouverts sur ce serveur à l'aide d'un script et de la commande :

```
> zsh port-scan.sh 104.21.63.124
```

```
➜ zsh port-scan.sh 104.21.63.124
Scanning port 21 for      ftp ...          Fail
Scanning port 22 for      ssh ...          Fail
Scanning port 23 for      telnet...        Fail
Scanning port 25 for      smtp...          Fail
Scanning port 53 for      dns ...          Fail
Scanning port 80 for      http...          Fail
Scanning port 110 for     pop3...          Fail
Scanning port 111 for     rpcbind...        Fail
Scanning port 135 for     msrpc...          Fail
Scanning port 139 for     netbios-ssn...    Fail
Scanning port 143 for     imap...          Fail
Scanning port 443 for     https...         Fail
Scanning port 445 for     ms-ds...          Fail
Scanning port 993 for     imaps...         Fail
Scanning port 995 for     pop3s...         Fail
Scanning port 1723 for    pptp...          Fail
Scanning port 2375 for    docker...        Success
Scanning port 3306 for    mysql...          Fail
Scanning port 3389 for    ms-wbt...        Fail
Scanning port 5900 for    vnc ...          Fail
```

Dès lors il se rend compte en effet que le port 2375 de docker est ouvert et s'imagine à juste titre de ce fait que les applications doivent en effet tourner à l'aide des containers dockers.

Si une méthode d'authentification sur cet hôte à partir de docker n'a pas été mis en place, il peut à l'aide d'une commande docker lister l'ensemble des containers de l'hôte.

```
> docker -H www.vote.com ps
```

Et là il aura la liste des containers de l'hôte

```
> docker -H www.vote.com version
```

Par la suite, il peut lancer un container ayant les priviléges à partir de l'image Ubuntu et utiliser ce container pour se déplacer à travers les autres containers hébergeant les applications de vote

```
> docker -H www.vote.com version run --privileged -it ubuntu bash
```

Une fois sur ce container docker, il peut à présent manipuler l'hôte et agir sur les autres containers à sa guise.

Il existe plusieurs façons de hacker des applications sur un cluster si des mesures de sécurité appropriées n'ont pas été mises en place.

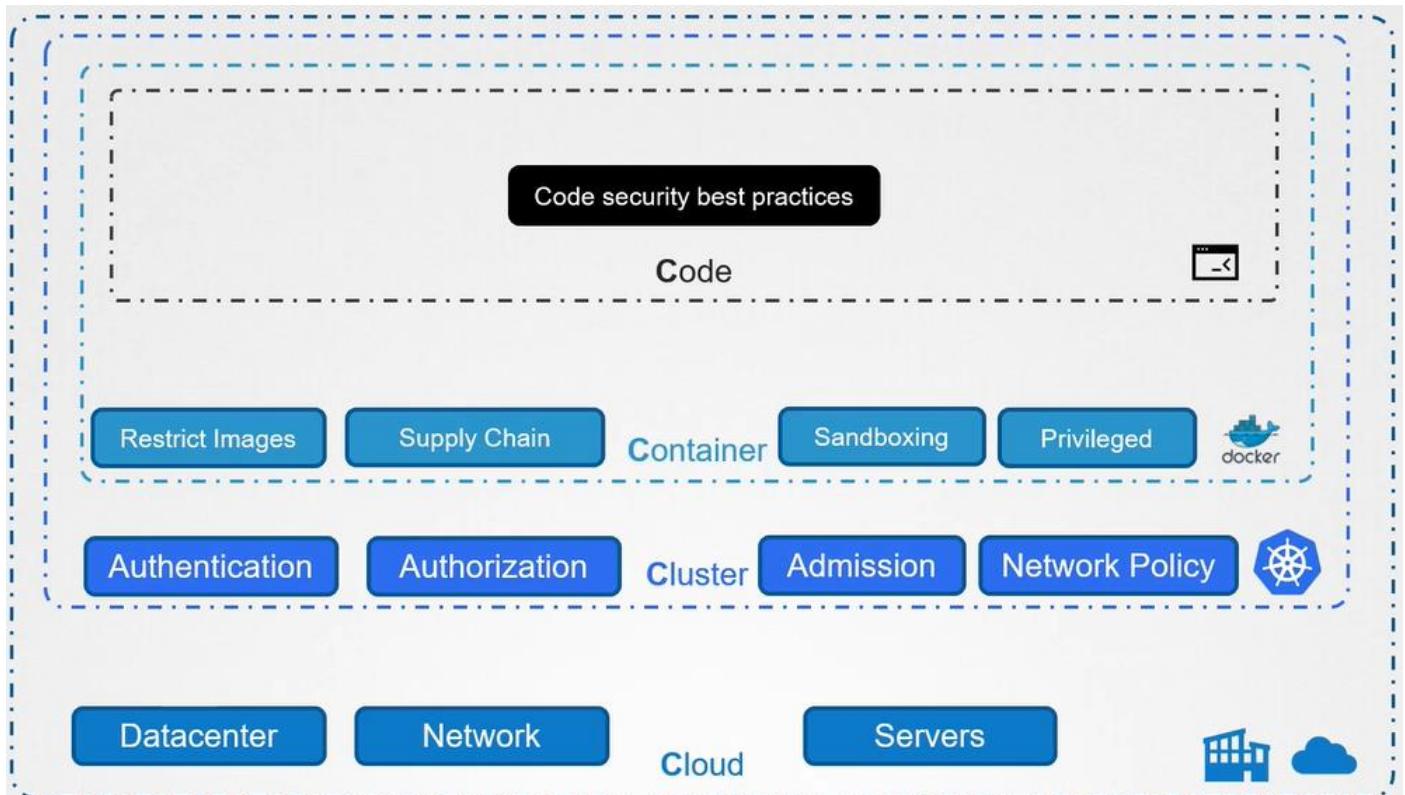
b. The 4C's of Cloud Native security

Les 4C pour la sécurité des applications dans le cloud sont :

- **Le cloud** : le cloud lui-même doit être sécurisé, car les applications sont hébergées par le cloud, il s'agit ici d'un cloud provider, ou d'un datacenter On-premises, sécuriser les serveurs, le réseau et mettre en place les pare-feu appropriés afin de prévenir les éventuelles intrusions. Il s'agit ici de garantir la sécurité de l'infrastructure globale hébergeant les serveurs.
- **Le Cluster** : Il s'agit ici de garantir la sécurité du cluster, car comme nous l'avons vu précédemment, le hacker peut facilement avoir accès au daemon docker qui est exposé publiquement sur le serveur ou à alors à travers le Dashboard Kubernetes qui lui aussi était exposé publiquement sans aucune sécurité. Cela peut être corrigé en mettant des mécanismes d'authentification et d'autorisations sur les cluster, en utilisant les concepts Admission et Network Policy pour sécuriser le cluster. Nous parlerons en détails de ces méthodes dans la partie « Cluster Setup and Hardening »
- **Container** : Nous avons vu plus haut que le hacker avait la possibilité de lancer tout autre container et de se connecter sur les containers de son choix de notre serveur. Cela doit être corrigé ; il faudrait aussi pouvoir

empêcher le Hacker de pouvoir installer des applications dans nos containers ou de lancer des scripts dans ces container-là. Ces concepts seront abordés dans la section « minimise Microservice Vulnerability »

- **Code** : Il s'agit ici du code applicatif, du code de notre application ; il est important de respecter les bonnes pratiques liées à la sécurité du code, par exemple coder en dur l'application avec les identifiants BDD, écrire les mots de passe en dur dans des variables d'environnements ou héberger des applications sans sécurité ou sans gestion des certificats TLS sont des mauvaises pratiques de codage.



II) Cluster Setup and Hardening

1) CIS BENCHMARK

CIS : **Center for Internet Security** est un organisme international à but non lucratif qui a pour mission de proposer ou standardiser un certain nombre de bonnes pratiques de développement, validation, déploiement et sécurisation des applications sur Internet dans le but de vous protéger le plus possible des cyber attaques.

Le site de CIS propose plusieurs standards en matière de cybersécurité pour la plupart des fournisseurs d'infrastructures variant en fonction des systèmes d'exploitation tel que : Linux, Windows, MacOS ... ; des Cloud providers tels que Google, Azure, AWS et autres ; des solutions mobile (Apple IOS, Android) ; solutions réseaux telles que (Check Point, Cisco, Juniper, Palo Alto...) ou solutions serveurs telles que (Tomcat, Docker, Kubernetes et autres) Afin d'utiliser ces standards CIS, il faut au préalable s'enregistrer sur le site de CIS et par la suite télécharger les standards de sécurité correspondants à la solution de votre choix. Une fois téléchargé, dans les pages ou documentation de ces normes standards de sécurité, vous trouverez les instructions vous permettant d'appliquer les différentes recommandations de sécurité. Pour chacune des recommandations, il vous montrera pourquoi est ce que c'est une menace, comment vérifier si elle existe dans votre système et comment vous protéger contre cette menace ; bien entendu cela inclut les lignes de commandes nécessaires pour réaliser ces opérations fonction de la solution que vous avez choisie. CIS propose également des utilitaires qui permettent de checker les menaces et d'y remédier automatiquement. C'est le cas de CIS-CAT Lite qui compare l'état du serveur à celui recommandé dans les normes ou standards CIS et reporte les problèmes si jamais il en trouve. Une fois lancé, il génère un fichier html qui résume l'ensemble des recommandations nécessaires afin d'être conforme aux bonnes pratiques

2) Use CIS Benchmark (Ex. On Ubuntu)

Nous allons dans cet exemple lancer une commande permettant de checker le système à l'aide du CIS-CAT (utilitaire CIS). Pour lancer cette commande, il faut au préalable télécharger le répertoire (Assessor-CLI) contenant les fichiers de l'utilitaire CIS-CAT. Dans ce fichier, lancer le script Assessor-CLI.sh en passant en option le paramètres (--report-dir ou -rd <repertoire du rapport>) ; (-html pour spécifier que le rapport sera au format html) ; (-rp ou -report-prefix <fichier rapport>. Et (-nts ou -no-timestamp ne pas inclure le timestamp dans le rapport)

```
# sh ./Assessor-CLI.sh -i -rd /var/www/html/ -nts -rp index
```

Permet de lancer le script en mode interactive afin d'y entrer manuellement les valeurs des variables demandées lors de l'exécution :

- Benchmark/Data-Stream Collections: CIS Ubuntu Linux 18.04 LTS Benchmarkv2.1.0
- Profile: Level 1 - Server

3) CIS Benchmark for Kubernetes

Pour utiliser le CIS Benchmark pour Kubernetes, il faut aller sur le site de CIS (<https://www.cisecurity.org/>) télécharger le Benchmark Kubernetes, ouvrir la documentation et lancer les tests de menaces ou de vulnérabilité manuellement afin d'y remédier si nécessaire. Bien entendu le CIS Benchmark proposera l'ensemble des bonnes pratiques permettant de sécuriser votre cluster Kubernetes et les lignes de commandes à utiliser pour y arriver. Malheureusement l'utilitaire CIS-CAT lite que nous avons vu précédemment ne possède que des Benchmark liés aux OS (Windows 10, Ubuntu, MacOs, Google Chrome...) ; pour ce qui concerne les autres Benchmark, il faudrait utiliser l'utilitaire CIS-CAT Pro qui est payant et qui supporte beaucoup plus de solutions et propose de ce fait beaucoup plus de Benchmark. Il existe 02 version de CIS-CAT Pro (la CIS-CAT Pro Assessor V3 et la CIS-CAT Pro Assessor V4) et de ces deux versions, c'est à partir de la V4 que l'on retrouve les benchmark Kubernetes.



Nous verrons par la suite un utilitaire free qui possède les CIS Benchmark Kubernetes et permet de vérifier les menaces et vulnérabilités d'un cluster Kubernetes.

Download the CIS benchmark PDF's from the below link: <https://www.cisecurity.org/cis-benchmarks/#kubernetes>
Download the CIS CAT tool' from the below link: <https://www.cisecurity.org/cybersecurity-tools/cis-cat-pro/cis-benchmarks-supported-by-cis-cat-pro/>

4) Kube-bench

L'utilitaire Kube-bench est un outil gratuit (open source) de Aqua Security qui permet de réaliser de façon automatique les différents tests confirmant que Kubernetes a été déployé de façon correcte et en respectant les bonnes pratiques de sécurité.

Kube-bench peut être déployé de plusieurs façons :

- Comme un container Docker
- Comme un POD dans un cluster Kubernetes
- A partir des fichiers binaires de Kube-bench
- Ou alors en compilant son code source.

a. Installation de Kube-bench

```
#Téléchargement de l'archive : curl -L https://github.com/aquasecurity/kube-bench/releases/download/v0.4.0/kube-bench\_0.4.0\_linux\_amd64.tar.gz
```

```
#Décompression : tar -xvf kube-bench_0.4.0_linux_amd64.tar.gz
```

b. Lancement d'une commande kube-bench

```
.#Se placer dans le répertoire où se trouve le fichier kube-bench et lancer la commande : /kube-bench --config-dir `pwd`/cfg --config `pwd`/cfg/config.yaml
```

Une fois la commande lancée, on a en retour à l'écran le rapport de Kube-bench par module et comprenant également les commandes permettant de remédier aux éventuels problèmes.

5) Kubernetes Security Primitives

Dans cette partie, nous parlerons des primitives de sécurité ou bonnes pratiques minimales en matière de sécurité à appliquer lors du déploiement d'un cluster Kubernetes.

Nous allons commencer par la sécurité des différents hosts formant le cluster

a. Secure Hosts

Afin de sécuriser les différents hôtes, il faudrait :

- Tous les accès aux hôtes doivent être sécurisé, le compte root désactivé
- La méthode de connexion par mot de passe doit être désactivée
- N'autoriser l'authentification que par Clé SSH

Par la suite il faut sécuriser l'accès à Kubernetes (au cluster Kubernetes)

b. Secure Kubernetes

Comme nous l'avons vu précédemment, le composant Kube-apiserver est au centre de tout dans le cluster, car pour interagir avec les autres éléments du cluster à travers les commandes kubectl, on passe par le composant Kube-apiserver. Donc pour commencer, il faudrait déjà contrôler et sécuriser les accès aux Kube-apiserver lui-même

Sécuriser ou gérer les accès au Kube-apiserver signifie déterminer Qui peut accéder ? comment peut-on y accéder ? et par la suite après avoir accès que peut-on y faire ?

Who can access?

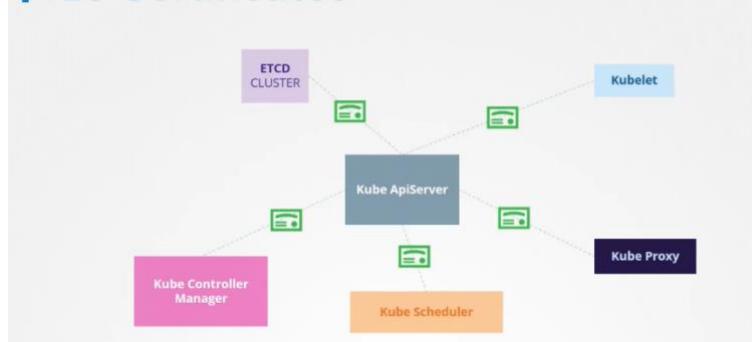
- Files – Username and Passwords
- Files – Username and Tokens
- Certificates
- External Authentication providers - LDAP
- Service Accounts

What can they do?

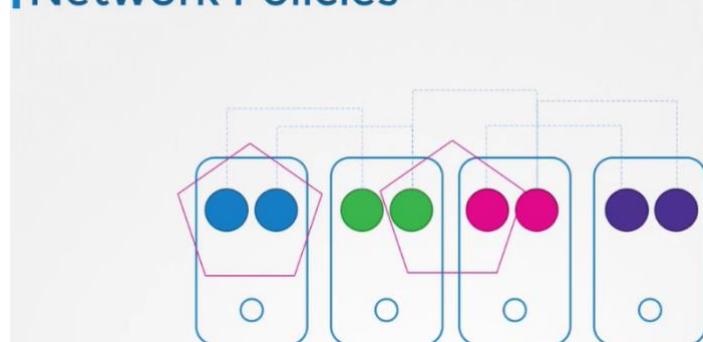
- RBAC Authorization
- ABAC Authorization
- Node Authorization
- Webhook Mode

L'accès aux différents composants du cluster Kubernetes se fait de façon sécurisée à l'aide des certificats TLS et l'on peut également limiter l'accès de certains Pods à d'autres en utilisant les network Policies. Ce sont là des différentes méthodes que l'on verra qui nous permettront de sécuriser davantage notre cluster ou nos applications sur un cluster Kubernetes.

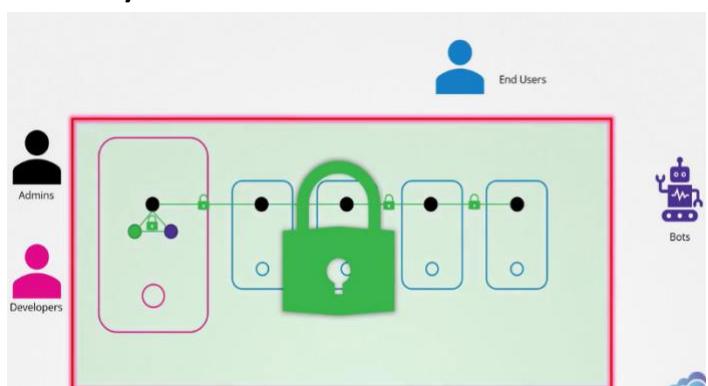
ITLS Certificates



Network Policies



6) Authentification à un cluster Kubernetes



Un cluster Kubernetes est un ensemble de machines (nœuds master et workers). Dans ce cluster on déploie des applications et les différents types de connexion que l'on peut avoir sur ce dernier sont :

- **Les admins** : pour administrer le cluster
- **Les développeurs** : se connectent pour déployer et tester les apps
- **Le End Users** : consomment les applications éployées sur le cluster
- **Les Bots** : qui renvoient à des API, Scripts ou applications tierces qui ont besoin de se connecter de façon programmatique au cluster afin d'interagir avec ses applications.

Il est important de bien sécuriser notre cluster en sécurisant

l'accès et les communications entre les différents composants internes du cluster, il faut également sécuriser les accès de management au cluster à travers des mécanismes d'authentification et d'autorisations.

NB : les Ends Users se connectant aux applications directement, leurs accès sont gérés par les applications et non par les mécanismes d'authentification et d'autorisation du cluster.

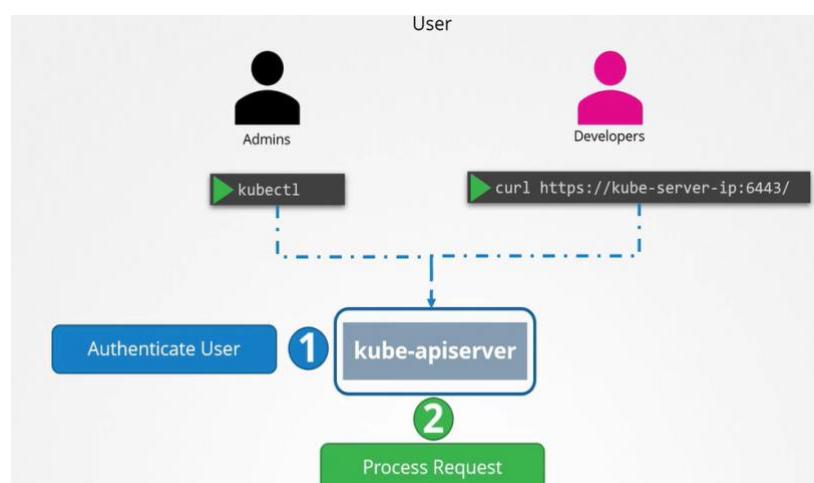
De ce fait, il nous reste deux types d'utilisateurs à gérer dans le cluster Kubernetes (les hommes ou users : admins et développeurs et les robots (services, API ou applications)



a) Les Users dans Kubernetes

Les Users ne sont pas vraiment gérés comme des ressources Kubernetes, donc il n'est pas possible de créer un user avec la commande `kubectl create user frazer`. Kubernetes ne permet pas de manager de façon native les users, cependant les users peuvent être fournis à Kubernetes à travers un fichier externe contenant les informations sur ces derniers, les certificats ou à travers une solution d'identification tierce comme LDAP.

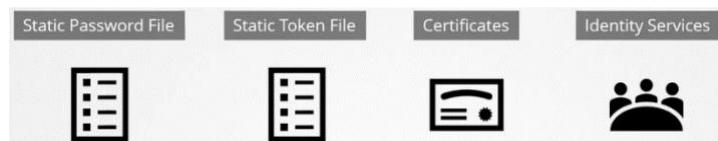
Il est important de rappeler que la gestion des accès des users est entièrement gérée par Kube-Apiserver, que ce soit une requête via `kubectl` ou via un `curl` sur l'adresse du cluster. Toutes ces requêtes passent par Kube-Apiserver qui dans un premier temps authentifie le user et par la suite exécute la requête.



b) Les modes d'authentification des Users dans Kubernetes

On distingue plusieurs méthodes d'authentification qui peuvent être configurées sur Kubernetes :

- Liste de users et password dans un fichier statique
- Liste de users et token dans un fichier statique
- Authentification à l'aide de certificat
- Authentification via un service d'identification tierce (LDAP...)



i) Static Password File

Il est possible de créer un fichier csv contenant les informations d'identifications, liste de users et password et le passer à Kubernetes comme la source d'informations concernant les users. Le fichier csv doit avoir 03 colonnes (password, username et userID). Et pour passer ce fichier comme source de users à Kubernetes, il faut modifier le manifest de création ou le service de `kube-apiserver` et ajouter `--basic-auth-file=user-details.csv` puis redémarrer `kube-apiserver` pour qu'il prenne en compte les modifications. Une fois redémarré, pour s'authentifier à `kube-apiserver` en utilisant un user, lancer par exemple la commande `curl` en spécifiant en paramètre le user `curl -v -k https://master-node-ip:6443/api/v1/pods -u "user1:password123"` et mdp à utiliser.

NB : il est possible d'avoir une quatrième et dernière colonne dans le fichier de définition des users permettant de spécifier le groupe auquel appartient cet user `password123,user5,u0005,group2`

```
user-details.csv
password123,user1,u0001
password123,user2,u0002
password123,user3,u0003
password123,user4,u0004
password123,user5,u0005
```

```
--basic-auth-file=user-details.csv
```

```
user-token-details.csv
KpjCVbI7rCFAHYPkByTlZRb7gu1cUc4B,user10,u0010,group1
rJjnchMvtxHc6MLWQddhtvNyyhgTdxSG,user11,u0011,group1
mjpOFIEiFOkL9toikaRNtt59ePtczZSq,user12,u0012,group2
PG41IXhs7QjqwWkmBkvgGT9g1OyUqZij,user13,u0013,group2

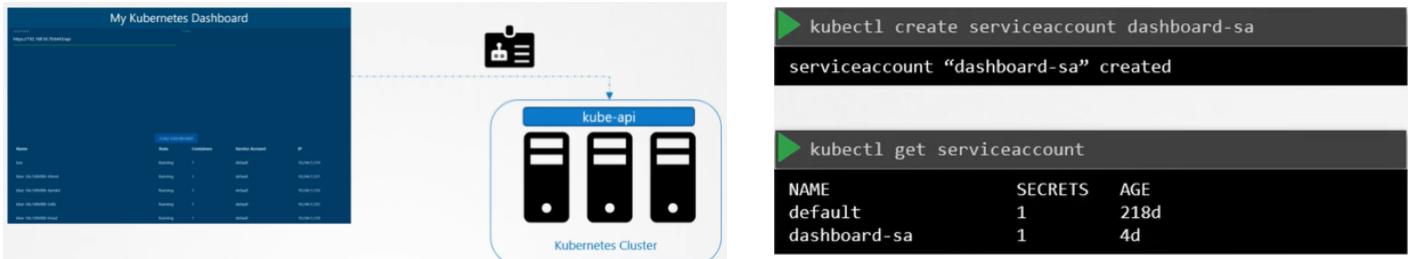
--token-auth-file=user-details.csv
```

```
curl -v -k https://master-node-ip:6443/api/v1/pods --header "Authorization: Bearer KpjCVbI7rCFAHYPkBzRb7gu1cUc4B"
```

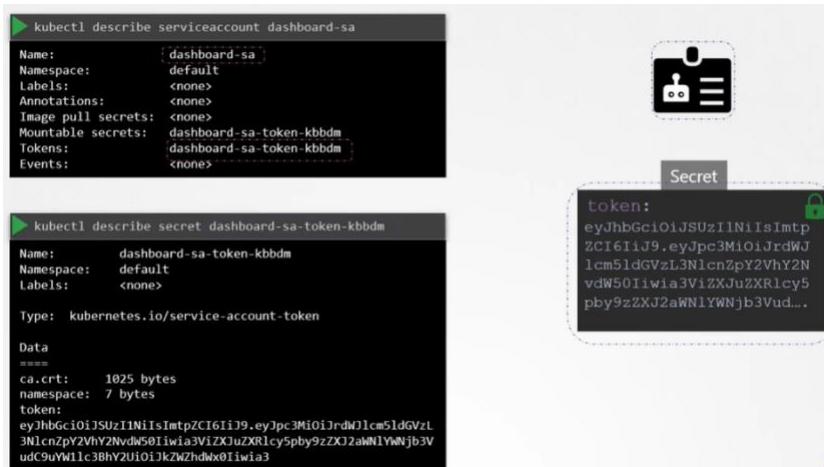
NB: les mécanismes d'authentification par static file password ou static file token ne sont pas recommandés car insécurisés. Penser aussi à monter les fichiers dans de volumes (POD) et à configurer les RBAC pour les différents users.

7) Service Accounts

Les services Accounts sont utilisés par des machines, des processus, services ou applications afin d'accéder au cluster Kubernetes. Il s'agit ici des comptes que l'on crée dans kubernetes afin de permettre à certaines applications externes de les utiliser pour interagir avec le cluster kubernetes. Exemple : Prometheus, graffana, jenkins...



Lorsque le serviceaccount est créé, il crée automatiquement un token qui sera en effet utilisé par l'application externe afin d'agir avec le cluster Kubernetes. Le token est sauvegardé dans kubernetes sous forme de secret



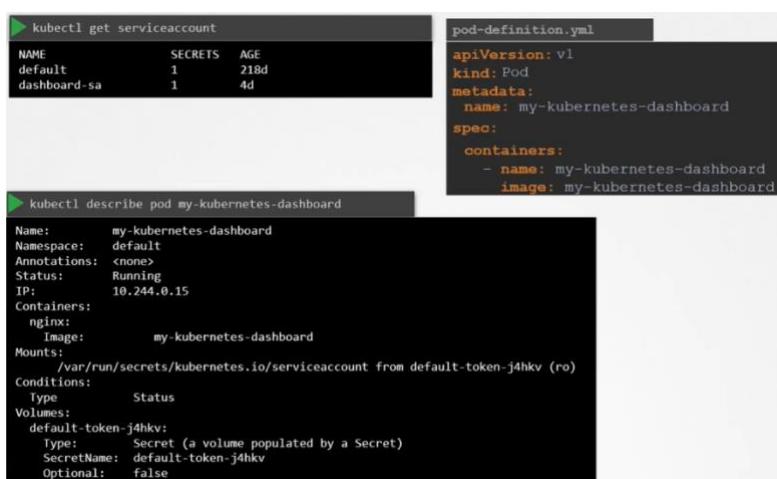
ce secret étant une ressource kubernetes à part entière, on peut accéder à ses paramètres à l'aide de la commande `kubectl describe secret <nom-secret>`

Ce token pourra ensuite être utilisé dans une application tierce afin d'interagir avec le cluster, par exemple :

```
▶ curl https://192.168.56.70:6443/api -insecure
--header "Authorization: Bearer eyJhbG..."
```

NB : après avoir créé le serviceaccount, il faudrait créer également les RBAC nécessaires

Il existe un serviceaccount par défaut dans tous les namespaces du cluster kubernetes, chaque namespace possède donc ce serviceaccount par défaut qui est « default ». ce serviceaccount est donc lié à un secret token qui est systématiquement monté par kubernetes comme volume dans chaque pod créé dans ce namespace.



le secret est monté dans le répertoire par défaut `/var/run/secrets/kubernetes.io/serviceaccount` et si on vérifie le contenu de ce répertoire une fois le pod créé, on se rend compte qu'il contient 03 fichiers :

- token : qui représente le token du default serviceaccount
 - ca.crt : certificat de l'autorité de certificat Api-server
 - namespace : Nom du namespace de ce serviceaccount
- NB :** le serviceaccount par défaut fourni automatiquement à tous les Pods du cluster dispose en effet juste des accès basiques et limitées au cluster, pour permettre à ce pod ou à l'application qui y est hébergée d'interagir avec un certain niveau d'accès défini avec le cluster, vous devez définir le serviceaccount à utiliser lors de la définition de votre pod.

```
kubectl get serviceaccount
NAME      SECRETS   AGE
default   1          21d
dashboard-sa 1          4d
```

```
pod-definition.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-kubernetes-dashboard
spec:
  containers:
    - name: my-kubernetes-dashboard
      image: my-kubernetes-dashboard
  serviceAccountName: dashboard-sa
```

```
kubectl describe pod my-kubernetes-dashboard
```

```
Name:           my-kubernetes-dashboard
Namespace:      default
Annotations:    <none>
Status:         Running
IP:            10.244.0.15
Containers:
  nginx:
    Image:        my-kubernetes-dashboard
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from dashboard-sa-token-kbbdm (ro)
Conditions:
  Type  Status
  ----  -----
  dashboard-sa-token-kbbdm:  Type:  Secret (a volume populated by a Secret)
  SecretName: dashboard-sa-token-kbbdm
  Optional:  false
```

Il faut en effet rajouter l'attribut `serviceAccountName` dans les `spec` du POD (cf. image manifest ci-après)

Et après cela si on décrit à nouveau le POD créé, on se rend compte en effet qu'il utilise bien le nouveau serviceaccount et fait bien le montage de ce secret dans le répertoire par défaut `/var/run/secrets/kubernetes.io/serviceaccount`.

NB : kubernetes monte le serviceaccount par défaut de chaque namespace dans chaque pod si un aucun serviceaccount spécifique n'a été défini dans le manifest de création du pod.

Si jamais l'on ne souhaite pas monter de serviceaccount par défaut, il faudrait définir le paramètre `automountServiceAccountToken: false`

8) Les certificats TLS

Les certificats TLS sont utilisés pour garantir la confiance entre deux parties durant une transaction. Par exemple, si un utilisateur essaie d'accéder à un serveur web, le certificat TLS s'assure que la communication entre le serveur et l'utilisateur est encrypté et également que le serveur en question est bel et bien le bon serveur demandé par le User. Le cryptage des communications est très important et on distingue 02 type de cryptage :

- **Cryptage symétrique** : la clé de cryptage utilisée pour crypter les données est la même pour le décryptage
- **Cryptage asymétrique** : la clé de décryptage est différente de celle de cryptage (Private and Public key)

Les certificats en plus d'encrypter les communications, permettent également d'identifier les différentes serveurs afin de limiter les différentes attaques en se faisant passer pour vos serveurs.

Chaque certificat est encodé de façon numérique et contient les informations suivantes :

- A qui le certificat est attribué
- La clé publique dudit serveur
- L'emplacement géographique dudit serveur
- La date de validité du certificat et plusieurs autres informations
- Les informations sur qui a signé et généré ce certificat (autorité de certificat)



CERTIFICATE AUTHORITY (CA)

Symantec
GlobalSign
digicert

Certificate Signing Request (CSR)

Validate Information

Sign and Send Certificate

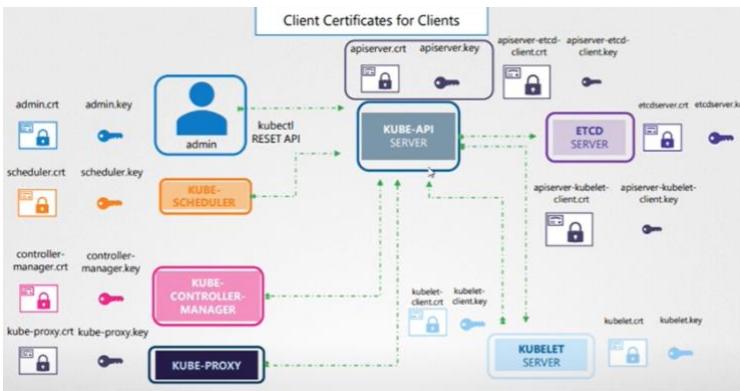
```
openssl req -new -key my-bank.key -out my-bank.csr
-subj "/C=US/ST=CA/O=MyOrg, Inc./CN=my-bank.com"
my-bank.key my-bank.csr
```

Les autorités de certification ont pour rôle de générer et de signer des certificats afin d'empêcher qui conque de signer un certificat en se faisant passer pour vous. Pour avoir un certificat, le user fait une demande de certificat CSR à l'autorité, ce dernier vérifie cette demande et si Ok, il l'approuve et la retourne signé et validé à l'utilisateur. La demande de CSR est faite par une commande OpenSSL et à l'aide de la clé privée générée précédemment (cf. image ci-après)

NB : généralement la clé privé a le mot clé key à l'intérieur du nom de la clé contrairement à la clé publique qui ne l'a pas forcément.

Certificate (Public Key)	Private Key
*.crt *.pem	*.key *.key.pem

9) TLS in Kubernetes



Sur Kubernetes, il est nécessaire d'avoir un certificat + key pour pouvoir communiquer avec ses différentes ressources.

Il existe 02 types de certificats sur Kubernetes :

- Les certificats + key de type serveur que possèdent les ressources de types serveurs (Kube-ApiServer, ETCD et Kubelet) sur lesquelles les clients peuvent avoir besoin de s'y connecter pour demander un service
- Les certificats + key de type client que les clients devront présenter aux serveurs afin qu'ils leur donnent l'accès.

Sur kubernetes installé à l'aide de kubeadm, les certificats serveurs et admin principal sont générés automatiquement à l'installation et se trouvent dans le dossier /etc/kubernetes/pki/

Les certificats génériques existants sont :

- KUBE-APISERVER

- Certificat + key de type serveur : /etc/kubernetes/pki/apiserver.crt et *.key
- Certificat + key de type client pour se connecter à ETCD : --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt et *.key
- Certificat + key de type client pour se connecter à Kubelet : --kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet-client.crt et *.key
- Certificat + key de l'autorité de certificat kubernetes-ca : /etc/kubernetes/pki/ca.crt et *.key
- Certificat + key de l'autorité de certificat etcd-ca : --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt et *.key

- ETCD

- Certificat + key ETCD server : --cert-file=/etc/kubernetes/pki/etcd/server.crt et --key-file=*.key
- Certificat de l'autorité de certificats etcd-ca : --peer-trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt et *.key

- KUBELET

- Certificat + key kubelet server : /etc/kubernetes/pki/kubelet.crt et kubelet.key
- Certificat de l'autorité de certificat kubernetes-ca : /etc/kubernetes/pki/ca.crt et *.key

- LES ADMINS

- Certificat + key de type client lui permettant de se connecter à API-server : <nom-admin>.crt et *.key
- Certificat de l'autorité de certificat kubernetes-ca : /etc/kubernetes/pki/ca.crt et *.key

NB : comme vous avez pu les remarquer, il existe 02 autorité de certificats sur kubernetes :

- L'autorité de certificat Kubernetes-ca pour api-server et kubelet
- L'autorité de certificat etcd-ca pour générer les certificats nécessaires pour ETCD

Tous ces certificats doivent être bien définis dans les manifests yaml permettant de créer les ressources static pods de l'api-server, kubelet et etcd se trouvant dans /etc/kubernetes/manifests

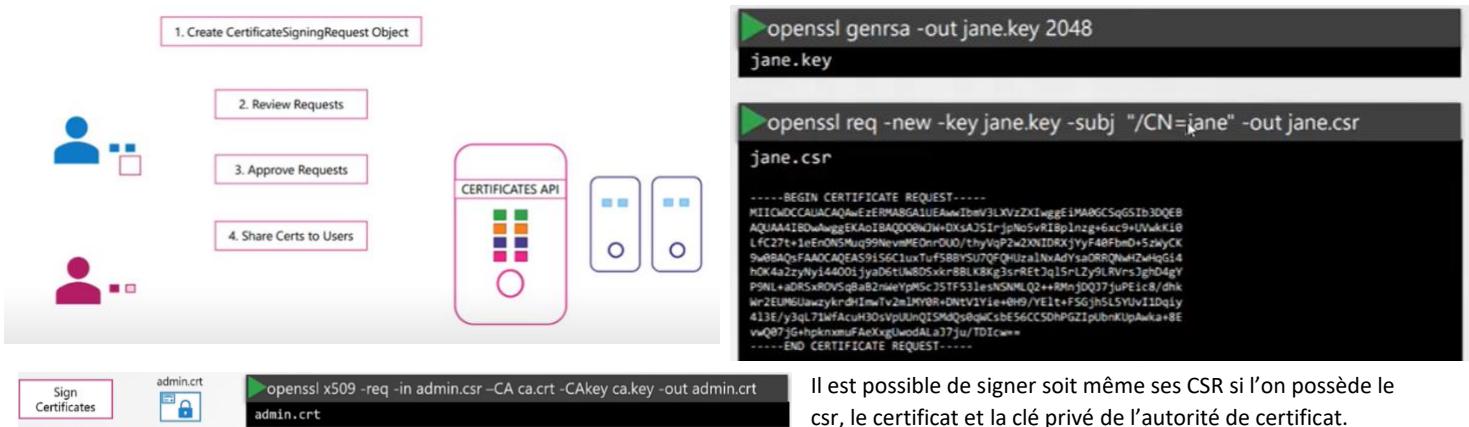
NB : la commande pour voir le contenu détaillé d'un certificat est : `openssl x509 -in <file-path>.crt -text`

Les certificats TLS permettent de créer des nouveaux utilisateurs au cluster kubernetes. Le processus de création d'utilisateurs sur kubernetes se fait en plusieurs étapes :

1. Création de la demande de certificats (CSR ou CertificateSigningRequest)
2. Après la création de la CSR qui est en effet un objet kubernetes, un admin existant devra vérifier ce CSR
3. Après vérification l'admin existant devra approuver ou refuser la demande de certificat
4. Si la demande est approuvée, alors Kubernetes générera le certificat et l'admin devra le partager au nouvel user pour que ce dernier puisse l'utiliser pour se connecter à kubernetes.



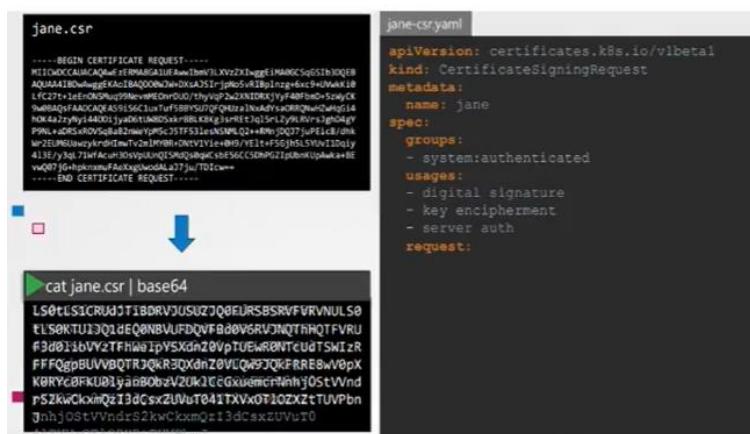
Pour Kubernetes, le certificat représente le userID et la clé représente de mot de passe, donc pour se connecter au cluster, il faut fournir pour un user, son certificat et sa clé privée. Il est également possible lors de la création du CSR de préciser le groupe dans lequel un user sera créé afin de lui accorder les privilège dudit groupe. Cela est possible grâce à l'option : `-subj "CN=<username>/O=<groupname>"` : le groupe ici est system :masters (groupe admin par défaut du cluster kubernetes)



Il est possible de signer soit même ses CSR si l'on possède le csr, le certificat et la clé privée de l'autorité de certificat.

Pour créer une demande de certificat, le nouvel utilisateur devra :

1. Générer la clé sur sa machine : `openssl genrsa -out jane.key 2048`
2. Créer une nouvelle demande de certificat à laquelle il attache la clé précédemment générée, fournit un Common name pour le CSR, il sauvegarde cette clé dans un fichier *.csr : `openssl req -new -key jane.key -subj '/CN=jane' -out jane.csr`
3. Une fois la demande de certificat créée sur sa machine ; il devra l'encoder en base 64 pour pouvoir créer le manifest Yaml devant créer cette ressource sur kubernetes : `cat jane.csr | base64 #permettra d'afficher le contenu du csr en base64`
4. Il recopie ensuite ce contenu en base64 dans le manifest à la section Requests (en une seule ligne)
5. Après avoir créé le manifest, il le partagera avec un admin existant qui se chargera d'exécuter la commande kubernetes : `kubectl create -f jane.csr #permettant de créer la ressource sur kubernetes`
6. Une fois la ressource créée, cet admin devra faire : `kubectl get csr #afin de voir la liste des csr existants`
7. Il devra ensuite approuver le certificat avec `kubectl certificate approve jane` ou refuser `kubectl certificate deny jane`
8. Une fois le certificat approuvé, il devra par la suite le décrypter et l'envoyer au nouvel utilisateur qui le consommera : `kubectl get csr jane -o yaml` et après l'avoir copié echo '`'copie-cert'`' | `base64 --decode`



kubectl get csr

NAME	AGE	REQUESTOR	CONDITION
jane	1m	admin@example.com	Pending

kubectl certificate approve jane

jane approved!

kubectl get csr jane -o yaml

```
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  creationTimestamp: 2019-02-13T16:36:43Z
  name: new-user
spec:
  groups:
  - system:masters
  - system:authenticated
  usages:
  - digital signature
  - key encipherment
  - server auth
  username: kubernetes-admin
status:
  certificate:
```

ls0tLS1CRUJdTiB0RVJUSUZjQ8URSBsRvFVRVNUlS0
ls0tKTU3Q1HEQNBvUFDQVFBD0V6RVJNQTHQTFVRU
#3d81bVv2TfWipYsXdn20VpTUEnR0NtCtUDTSW1zR
FFFQgbUUVBQTRQkR3QxhZ0VLQWQJQkFRREbvwOpX
K0RYcBFKU0jyaaObzj2Uk1cgXuiaetnHnjGStVnd
p52KwClxiQzI3dCsx2Ulu0t41TxVx0T1OZxTzTUVpn
jnhj0stVVndrS2kwckmpti3dcasxUVUto

echo "ls0t...Qo=" | base64 --decode

```
-----BEGIN CERTIFICATE -----
MIICWDCCAUACQAwezERMABGAlUEAwvIbmV3LXVzzXIWggE1MA0GCSqGSIb3DQE
AQAA4IBDwAwgEKAoTBAQD0WlDU/hDXAS7S1rjphNo5vR1Rp1nzg+6xc9+UWkL0
Lfc27t+1eN0NSMu99NemMEOnrDUO/thyqP2x2XNDRKjYyF4F8FmD+5zkyCK
9wBAQsFAAOCAQEA59156C1uxTu5BB8ySU7QFHUza1NxAdyvzqfzqHwIzWgG14
hOK4aZzyNy144001jyaD6tIw805xkr8BLK8Kg3sREtq15rLz9yLRVr3ghM4gY
P9NL+aDRSkx0V5qgab2zneYpMScJ5TF53lesNSNMLQ2++MnJ0Q7juPe1c8/dhk
WzEUM6UawzykrdrHmwTv2n1MY0R+DNvV1Yie+0H9/YEl1+FSGjh5LSYvI1DqjY
413E/y3ql71wfAcuH30yvpuUnQISMdQs0qWcsE56CC5DhPGIpubnUpAwka+8E
vwQ87jG+hpknxmuFeXxgJuwodAlJ77ju/TD1cw==

-----END CERTIFICATE -----
```

KODEKLOUD

Comme toute ressource kubernetes on peut effectuer les actions suivantes sur les csr

- Kubectl certificate approve/deny <csr-name>
- Kubectl get csr # la liste des csr
- Kubectl get csr <csr-name> -o yaml #Détails objet
- Kubectl delete csr <csr-name>
- Kubectl describe csr <csr-name>

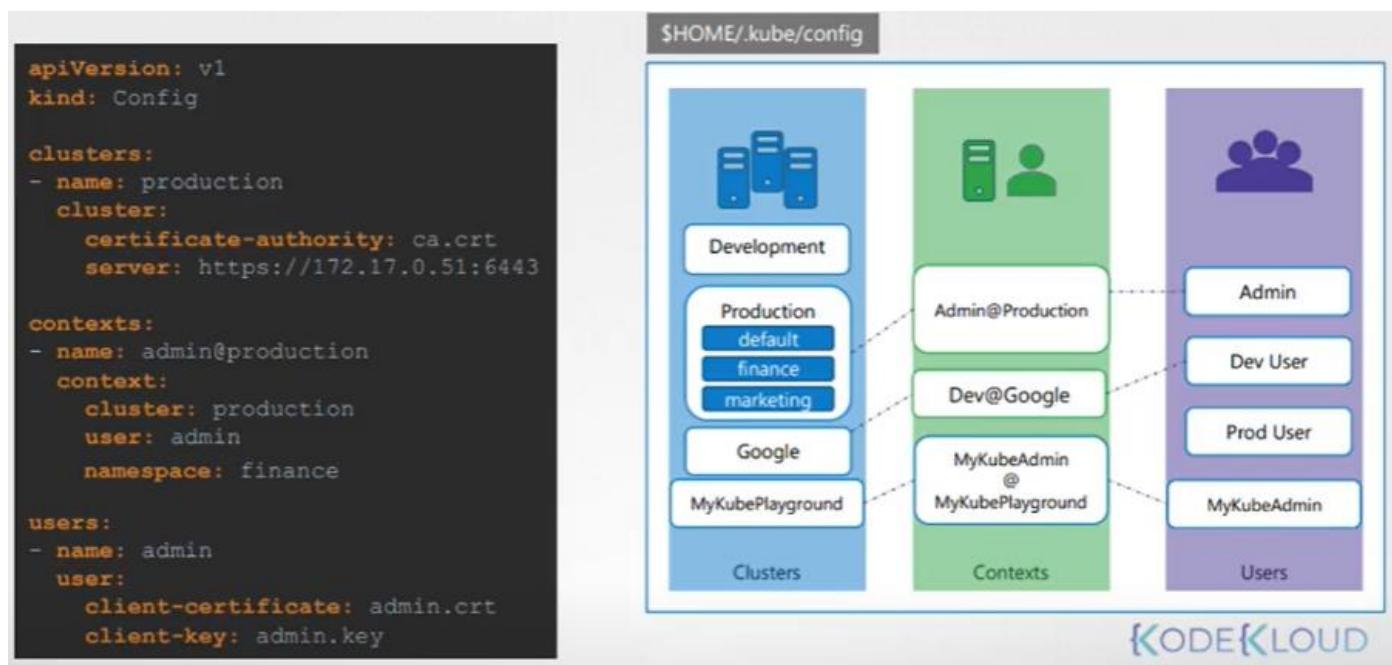
10) KUBECONFIG

Cette partie suit la partie précédente et permet au nouvel utilisateur de configurer son fichier de config kubernetes lui permettant d'accéder au kubernetes distant pour lequel il a reçu le certificat.

NB : le fichier de config kubernetes par défaut se trouve dans la variable d'environnement KUBECONFIG

Ce fichier de configuration /home/user/.kube/config nous permet en effet de configurer et déterminer ainsi les accès qui seront utilisés sur les différents clusters kubernetes existants. Il est généralement divisé en plusieurs sections :

- **Clusters** : permettant de définir la liste de clusters existants (leur nom, le certificat de l'autorité de certificat à utiliser et le nom ou l'adresse IP dudit cluster)
- **Users** : permettant de définir la liste des user en spécifiant pour chaque user sa clé et son certificat reçu du cluster kubernetes vers lequel il souhaite se connecter
- **Contexts** : contenant la liste des context qui permettent en effet de faire l'association entre un user et un cluster et dans l'interface de kubernetes, quand on souhaite utiliser un contexte, cela veut dire qu'on souhaite se connecter au cluster défini dans ce cluster avec le user également défini.



```
kubectl config view
apiVersion: v1
kind: Config
current-context: kubernetes-admin@kubernetes

clusters:
- cluster:
  certificate-authority-data: REDACTED
  server: https://172.17.0.5:6443
  name: kubernetes

contexts:
- context:
  cluster: kubernetes
  user: kubernetes-admin
  name: kubernetes-admin@kubernetes

users:
- name: kubernetes-admin
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

```
kubectl config view -kubeconfig=my-custom-config
apiVersion: v1
kind: Config
current-context: my-kube-admin@my-kube-playground

clusters:
- name: my-kube-playground
- name: development
- name: production

contexts:
- name: my-kube-admin@my-kube-playground
- Name: prod-user@production

users:
- name: my-kube-admin
- name: prod-user
```

```
kubectl config view
apiVersion: v1
kind: Config
current-context: my-kube-admin@my-kube-playground

clusters:
- name: my-kube-playground
- name: development
- name: production

contexts:
- name: my-kube-admin@my-kube-playground
- Name: prod-user@production

users:
- name: my-kube-admin
- name: prod-user
```

```
kubectl config use-context prod-user@production
apiVersion: v1
kind: Config
current-context: prod-user@production

clusters:
- name: my-kube-playground
- name: development
- name: production

contexts:
- name: my-kube-admin@my-kube-playground
- Name: prod-user@production

users:
- name: my-kube-admin
- name: prod-user
```

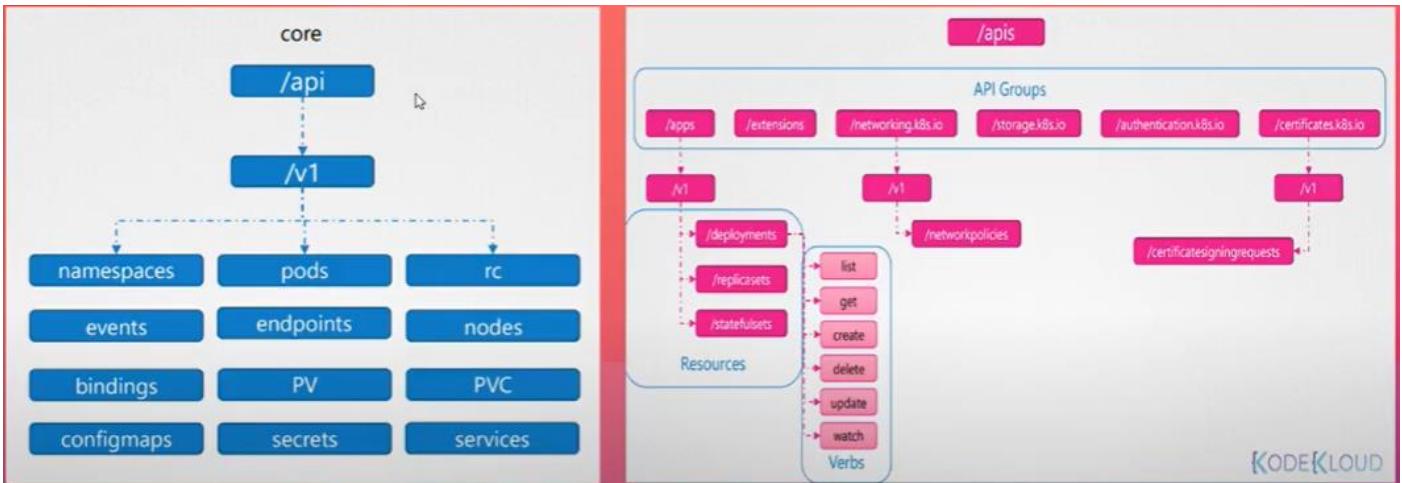
On peut également appliquer un contexte se trouvant dans un fichier de configuration autre que celui par défaut :

`Kubectl config use-context -kubeconfig=<my-custom-config-file> <context-name>`

`Kubectl config view -kubeconfig=<my-custom-config-file> # pour voir le contenu du custom config-file`

`Kubectl config use-context <context-name> #utilise le config-file par défaut et bascule le context`

11) API GROUPS



Lorsqu'on interagit avec kubernetes, il est intéressant de savoir exactement sur quel objet on travaille. Il existe à cet effet les api group Core et les autres apis. (NB : cf. fichier manifest de création des ressources). Il est important de connaitre la structure de ces différents api group car ils sont utilisés pour remplir les champs apiversion et kind lors de la création des manifests.

Kubectl proxy : permet d'exposer les api group d'un cluster kubernetes en local sur le port 8001 afin qu'il puisse être accessibles sans avoir besoin de fournir les informations liées aux certificats.

```
curl http://localhost:6443 -k
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {

  },
  "status": "Failure",
  "message": "Forbidden: User \"system:anonymous\" cannot get path \"/\"",
  "reason": "Forbidden",
  "details": {
  },
  "code": 403
}

curl http://localhost:6443 -k
--key admin.key
--cert admin.crt
--cacert ca.crt
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/",
    "/healthz",
    "/logs",
    "/metrics",
    "/openapi/v2",
    "/swagger-2.0.0.json",
    "/metrics"
  ]
}
```

```
kubectl proxy
Starting to serve on 127.0.0.1:8001

curl http://localhost:8001 -k
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/",
    "/healthz",
    "/logs",
    "/metrics",
    "/openapi/v2",
    "/swagger-2.0.0.json",
    "/metrics"
  ]
}
```

Curl http://localhost:6443 -k #permet d'afficher la liste des api groups utilisables par l'utilisateur kubernetes connecté. L'attribut « -k » permet de ne pas fournir les infos d'authentifications, on remarque dans ce cas que la requête s'affiche mais kubernetes refuse d'afficher la liste d'api car il n'y'a pas d'infos d'autorisation.

Kubectl api-resources : permet de voir la liste de toutes les ressources existantes dans kubernetes

12) RBAC (Role Based Access Controls)

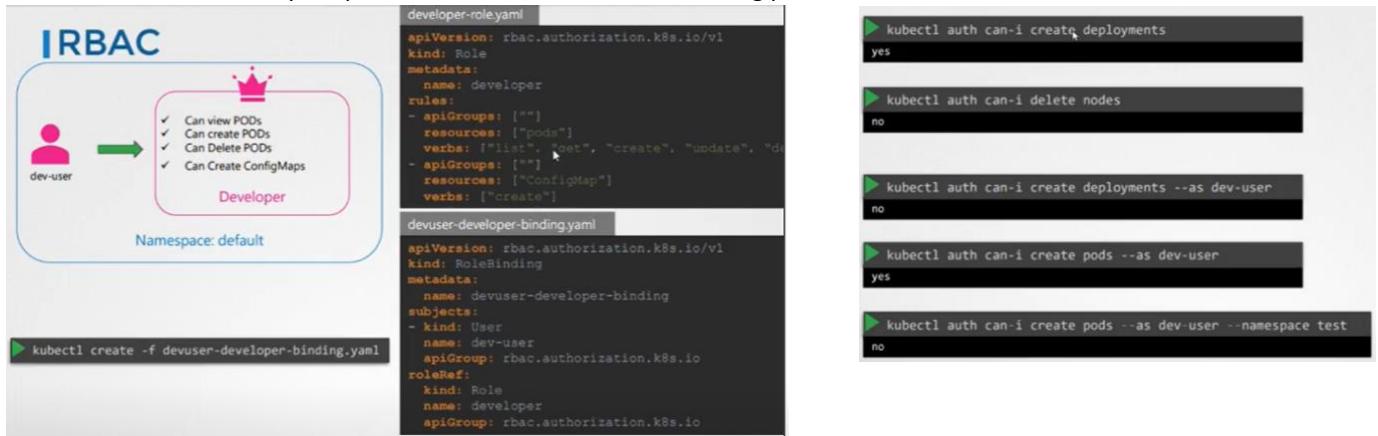
Cette notion permet lors de l'administration d'un cluster kubernetes, de définir de façon détaillée les droits d'accès aux ressources des différents users existants.

Cela permet de créer un certain nombre de droits et de les assigner à des users spécifiques.

Pour vérifier le type d'autorisations définis sur un pod, il faut taper la commande `kubectl describe` sur ce pod et regarder la valeur de son paramètre `authorization`

```
kubectl get role --all-namespaces : Affiche la liste des rôles existants dans l'ensemble des namespaces  
kubectl get role -n kube-system kube-proxy -o yaml: affiche les details sur creation du role kube-proxy  
kubectl get rolebindings.rbac.authorization.k8s.io --all-namespaces  
kubectl get rolebindings.rbac.authorization.k8s.io -n kube-system kube-proxy -o yaml  
kubectl --as dev-user get po
```

NB: utiliser des manifests yaml pour créer des rôles et des roleBinding pour associer ces rôles là aux users.



NB : les clusters Rôles permettent de manipuler ou de donner des autorisations aux objets n'ayant pas de namespace ou n'étant pas cloisonnés à un namespace, donc s'appliquant à tout le cluster.

Use the command `kubectl create` to create a new `ClusterRole` and `ClusterRoleBinding`.

Assign it correct `resources` and `verbs`.

After that test the access using the command `kubectl auth can-i list storageclasses --as michelle`.

`Kubectl api-resources` : permet de voir la liste de toutes les ressources existantes dans kubernetes.

```
---  
kind: ClusterRole  
apiVersion: rbac.authorization.k8s.io/v1  
metadata:  
  name: storage-admin  
rules:  
- apiGroups: [""]  
  resources: ["persistentvolumes"]  
  verbs: ["get", "watch", "list", "create", "delete"]  
  
- apiGroups: ["storage.k8s.io"]  
  resources: ["storageclasses"]  
  verbs: ["get", "watch", "list", "create", "delete"]
```

```
---  
kind: ClusterRoleBinding  
apiVersion: rbac.authorization.k8s.io/v1  
metadata:  
  name: michelle-storage-admin  
subjects:  
- kind: User  
  name: michelle  
  apiGroup: rbac.authorization.k8s.io  
roleRef:  
  kind: ClusterRole  

```

`Kubectl get clusterrole --no-headers |wc -l` #affiche le nombre de cluster role existants

13) Kubelet Security

Kubelet est en quelque sorte l'agent kubernetes qui se trouve sur les différents nœuds workers du cluster, permettant ainsi la communication entre le master (kube-apiserver) et le worker (kubelet). Il est donc indispensable et important de bien sécuriser l'agent kubelet se trouvant sur le nœud master afin que les communications entre kube-apiserver et kubelet soient contrôlées et sécurisées.

Kubelet Rôles

Register Node
Create PODS
Monitor Node & PODS

a) Kubelet installation and configuration

L'installation de kubelet se fait en téléchargeant ses binaires et en les configurant comme service. Il existe 02 façons de configurer kubelet :

- Soit directement au niveau du fichier service
- Ou alors à partir d'un fichier de config.yaml généralement situé dans /var/lib/kubelet/config.yaml et en passant l'option "`--config= <path-config>`" dans le fichier service.

Vous remarquerez que dans le fichier de config on a transcrit les mêmes paramètres présents dans le service (`--healthz-port = healthzPort`)

NB : Si jamais vous précisez le même argument dans le fichier de config et directement dans le service, l'argument passé dans le service est prioritaire sur celui du fichier de config.

The screenshot shows the terminal output for installing kubelet from storage.googleapis.com. It includes the command to download the binary, the creation of a kubelet.service file, and the creation of a kubelet-config.yaml file. The kubelet-config.yaml file is highlighted in yellow, showing its contents which include the --config option pointing to the file's path.

NB : lorsque vous installez kubernetes à l'aide de kubeadm, ce dernier vous aide à configurer automatiquement kubelet (bien entendu kubelet doit être installé au préalable sur le worker). Lorsque vous faites la commande "`kubeadm join`", il crée automatiquement le fichier "`/var/lib/kubelet/config.yaml`" qui va bien sur le node en question.

`ps -aux | grep kubelet` Une fois kubelet configuré, on peut taper la commande "`ps -aux | grep kubelet`" afin de voir ses différents paramètres.

b) Kubelet Security

Ici il s'agit de mettre en œuvre des mécanismes garantissant que kubelet répond ou communique seulement avec votre serveur kube-apiserver et non quelqu'un d'autre.

Kubelet utilise 02 ports :

- Le 10250 qui concerne les API qui ont les droits et autorisations sur kubelet
- Le 10255 qui concerne les API non autorisés ou qui ont seulement l'accès en lecture seule.

Par défaut, ces deux ports sont ouverts et donnent l'accès au moins en lecture pour les users authentifiés ou pas. Ce qui est en réalité une énorme faille de sécurité car peu importe si vous êtes authentifié ou pas, il vous suffit de connaître l'adresse IP du nœud afin d'accéder en lecture à ses métriques par exemple.

The screenshot shows a curl command to access the kubelet metrics endpoint at port 10255. The output shows various metrics related to audit events and client certificate expiration.

The screenshot shows the kubelet-service file and the kubelet-config.yaml file. The kubelet-service file has the --anonymous-auth=false option. The kubelet-config.yaml file also has the anonymous authentication section set to enabled: false. A blue box labeled "Authentication" is placed over the kubelet-service file.

Pour remédier à cela, il faut mettre en place un mécanisme sur kubelet qui authentifie au préalable chaque requête avant de l'autoriser ou exécuter. Pour cela, il faut refuser toute requête anonyme et après définir la méthode d'authentification à utiliser. Pour refuser les requêtes anonyme, il faut spécifier l'option "`--anonymous-auth=false`" dans le fichier de configuration ou dans le service kubelet.

Et concernant les méthodes d'authentification à mettre en place dans kubelet, il existe 02 méthodes : à l'aide des certificats (x509) ou à l'aide des API Bearer Tokens.

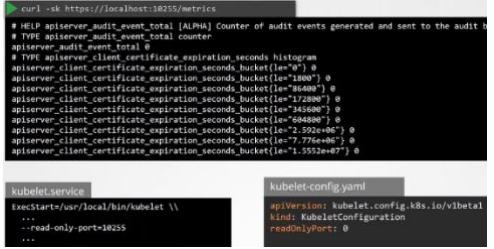
Pour activer la méthode d'authentification par certificat X509, il faut spécifier le certificat de l'autorité de certificat que kubelet utilisera pour vérifier les certificats clients à l'aide de `--client-ca-file=/path/to/ca.crt` dans le service kubelet. Et dorénavant pour accéder à kubelet, il faudra présenter un certificat et une clé privée. (cf. image ci-après) Etant donné que Kubelet est dorénavant sécurisé et accessible uniquement par authentification via certificat, vous devez également correctement configurer kube-apiserver en lui donnant les certificats clients kubelet lui permettant de se connecter pour communiquer avec kubelet

The screenshot shows the kubelet-service file and the kubelet-config.yaml file. The kubelet-service file has the --client-ca-file option. The kubelet-config.yaml file has the x509: clientCAFile option. Below, a terminal command shows curl https://localhost:10258/pods -k using a key and certificate. Another terminal command shows the kubelet-apiserver.service file with the --key=kubelet-key.pem and --cert=kubelet-cert.pem options.

Concernant les autorisations, par défaut une fois authentifié, kubelet autorise toutes les requêtes, ce qui peut aussi être une faille de sécurité dans certains cas. Afin de checker les autorisations, il faut modifier le paramètre `--autorisation-mode=AlwaysAllow` à `--autorisation-mode=Webhook` ce qui permettra à kubelet d'aller vérifier au niveau de l'api-server s'il peut approuver ou rejeter les requêtes.

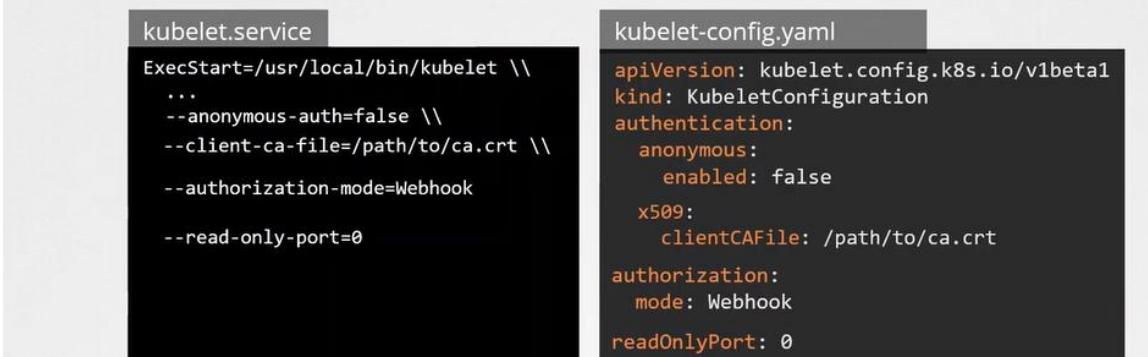


Afin de sécuriser davantage kubelet, il faut également désactiver le port 10255 qui autorise les accès anonymes en lecture seules pour lire les métriques ; pour cela, il faut modifier le paramètre `--read-only-port=10255` par `--read-only-port=0`. Le read only port doit être égal à 0 afin de désactiver le service de lecture seule sur kubelet. Si jamais il a une autre valeur que 0, alors on pourra passer par ce port là pour avoir les métriques du serveur kubelet. Il faut donc bien vérifier les deux fichiers de configuration de kubelet (`kubete.service` et `/var/lib/kubelet/config.yaml`) afin de détecter et modifier les différents paramètres nécessaires.



c) Kubelet Security résumé

Kubelet Security



14) Kubectl proxy et kubectl port-forward

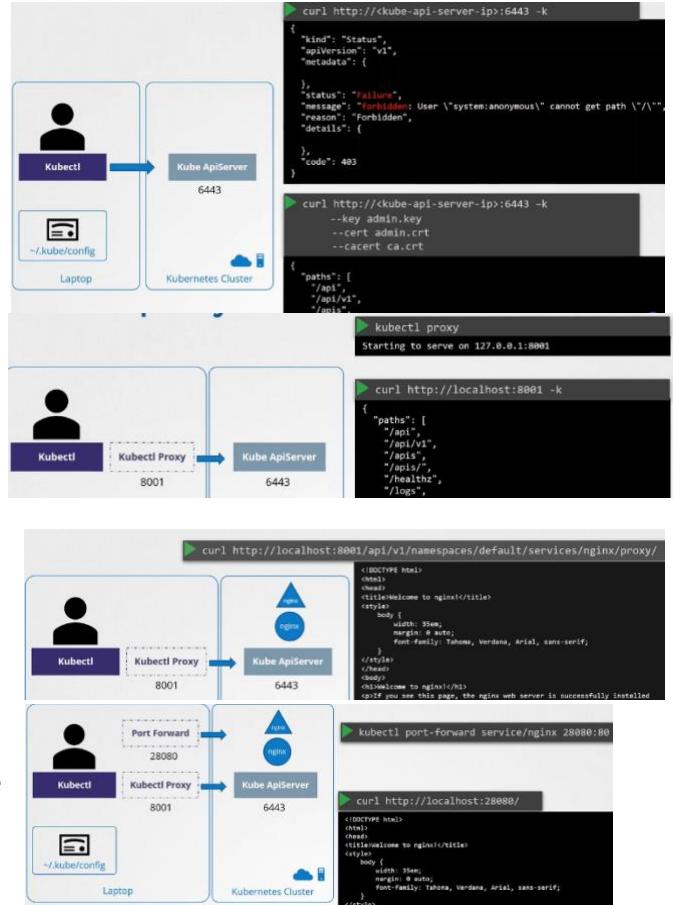
Nous savons que kubectl est l'utilitaire qui nous permet d'interagir avec kube-apiserver et que nous pouvons l'utiliser sans avoir à saisir les paramètres d'identifications (certificats, clés et autre) parce que ces derniers ont déjà été renseigné au préalable dans Kubeconfig (.kube/config) de l'utilisateur. Il est également possible d'interagir avec kube-apiserver en utilisant le port 6443 `"curl http://<api-server-ip>:6443 -k"` mais ça ne marchera pas car il faut en plus de cela fournir les informations d'authentification à l'apiserver.

Il est possible d'utiliser une autre méthode afin d'accéder à notre kube-apiserver sans fournir les informations d'authentification, cela est possible grâce à `kubectl proxy` qui va exposer automatiquement kube-apiserver sur le port 8001 de votre machine. Cette commande permet à kubeproxy d'utiliser les credentials de kubectl. **NB :** le kubeproxy ne fonctionne que sur la machine qui l'a lancé via l'adresse 127.0.0.1

`kubectl proxy --port 8002` (lancer sur un port spécifique)

kubectl proxy --port 5555 --address=0.0.0.0 --disable-filter (kube-apiserver accessible à l'extérieur à partir de l'IP de la machine : très risqué)

Kubectl proxy permet de lancer n'importe quelle requête à notre cluster via kube-apiserver, exemple on peut requêter un service existant sur le cluster



Il existe également une autre commande pour accéder au service Nginx se trouvant sur le cluster sans avoir à utiliser un Nodeport : `kubectl port-forward`

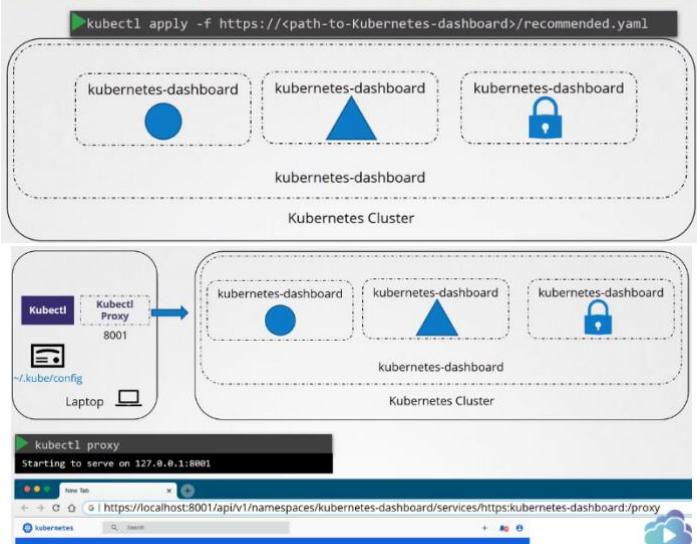
15) Kubernetes Dashboard

Il s'agit d'un autre projet kubernetes permettant d'avoir une représentation graphique d'un cluster kubernetes accessible via une page web. Kubernetes Dashboard permet d'avoir une représentation graphique du cluster, de monitorer les éléments et même également de provisionner de nouvelles ressources dans le cluster. Compte tenu de l'ensemble de actions qu'il permet de réaliser dans le cluster, il est important pour nous de bien le sécuriser.

Pour déployer kubernetes dashboard, il suffit de télécharger et d'appliquer le fichier manifest ayant les configurations de création des ressources nécessaires. Ce fichier se trouve sur le repo github de kubernetes et permet de déployer un namespace appelé kubernetes-dashboard, un deployment kubernetes-dashboard qui héberge l'application web, un service de type cluster IP kubernetes -dashboard qui expose le pod en interne et un configmap kubernetes-dashboard contenant tous les paramètres, certificats et secrets nécessaires.

Il est déconseillé pour des raisons de sécurité d'exposer directement le Dashboard à l'extérieur du cluster à l'aide d'un service de type Nodeport ou LoadBalancer. Il est préférable de passer par un serveur proxy d'authentification qui vérifiera dans un premier temps les autorisations avant de forwarder vers le service ClusterIP si OK.

Deploying Kubernetes Dashboard



```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.4.0/aio/deploy/recommended.yaml
kubectl -n kubernetes-dashboard create clusterrolebinding dashboard-admin-list-namespace-binding --clusterrole list-namespace --serviceaccount kubernetes-dashboard:dashboard-admin
kubectl -n kubernetes-dashboard create rolebinding dashboard-admin-binding --clusterrole cluster-admin --serviceaccount dashboard-admin --namespace=kubernetes-dashboard
```

```
# Recupération du token du service account du dashboard
TOKENNAME=`kubectl -n kubernetes-dashboard get serviceaccount/kubernetes-dashboard -o jsonpath='{.secrets[0].name}'`
TOKEN=`kubectl -n kubernetes-dashboard get secret $TOKENNAME -o jsonpath='{.data.token}' | base64 --decode`
```

```
# exposition de l'api à pectorieur
kubectl proxy --address 0.0.0.0 --accept-hosts '.*'
```

16) Securing Kubernetes Dashboard

Il existe 02 façons de s'authentifier à kubernetes Dashboard :

- **Avec un Token:** Vous devez créer un serviceaccount et lui donner les autorisation via RBAC puis récupérer son token pour vous connecter o Kubernetes dashboard
- **Avec un kubeconfig file** permettant d'accéder au cluster : donc vous devez fournir un fichier kubeconfig ayant un user disposant les droits ou autorisations sur le cluster

Kubernetes Dashboard

Token
Every Service Account has a Secret with valid Bearer Token that can be used to Bear Tokens, please refer to the [Authentication section](#).

Kubeconfig
Please select the kubeconfig file that you have created to configure access to file, please refer to the [Configure Access to Multiple Clusters section](#).

Below are some references:

- <https://redlock.io/blog/cryptojacking-tesla>
- <https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>
- <https://github.com/kubernetes/dashboard>
- <https://www.youtube.com/watch?v=od8TnlvuADq>
- <https://blog.heptio.com/on-securing-the-kubernetes-dashboard-16b09b1b7aca>
- <https://github.com/kubernetes/dashboard/blob/master/docs/user/access-control/creating-sample-user.md>

17) Verify platform binaries

Il est important de vérifier que les fichiers binaires téléchargés depuis un serveur distant soient bel et bien les mêmes que ceux sauvegardés sur nos disques. En effet, lors d'un téléchargement de fichier depuis internet, un hacker peut se mettre au milieu de la communication et modifier le fichier téléchargé en y insérant un malware. Une fois le fichier modifié, il n'est plus le même que celui présent sur le site serveur et a désormais une signature (Hash) différente du fichier présent sur ce serveur distant. Certains site web en plus de fournir le fichier, fournissent également le Hash permettant de vérifier l'authenticité de ce fichier une fois téléchargé en local. Et pour ce faire on utilise les commandes "`shasum -a 512 <fichier>` ou `sha512sum <fichier>`" pour générer

Downloads for v1.20.0

filename	sha512 hash
kubernetes.tar.gz	ebfe49552bbda02807034488967b3b62bf9e3e507d56245e298c4c19090387136572c1fc789e772a5e8a19535531d01dcdb61980e42ca7b0461d3864-df2c14

```
curl https://dl.k8s.io/v1.20.0/kubernetes.tar.gz -L -o kubernetes.tar.gz
```

```
shasum -a 512 kubernetes.tar.gz
```

```
ebfe49552bbda02807034488967b3b62bf9e3e507d56245e298c4c19090387136572c1fc789e772a5e8a19535531d01dcdb61980e42ca7b0461d3864
```

MacOS

```
shasum -a 512 kubernetes.tar.gz
```

Linux

```
sha512sum kubernetes.tar.gz
```

le hash du fichier reçu et vérifier sa valeur avec celui présent sur le site distant.

curl https://dl.k8s.io/v1.20.0/kubernetes.tar.gz -L -o /opt/kubernetes.tar.gz

tar -czf kubernetes-modified.tar.gz kubernetes // Compress repo kubernetes to archive kubernetes-modified.tar.gz

tar -xf kubernetes-modified.tar.gz // décompresser l'archive...

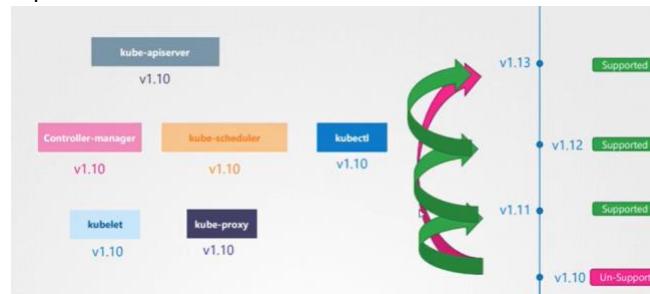
18) Kubernetes Upgrade



NB : si le Kube-Apiserver est dans une version X, alors le controller-manager et le kube-scheduler doivent être au moins en version X-1 et kubelet et kube-proxy quant à eux devront être également au moins en version X-2

Le Kubectl quant à lui peut être au plus à une version X+1 ou au moins à une version X-1

Les bonnes pratiques d'upgrade Kubernetes demandent de faire l'upgrade progressivement d'une version à une autre jusqu'à arriver à la cible qui est la version la plus récente



Il existe plusieurs méthodes d'upgrade de kubernetes, dépendant de la façon dont il a été initialement installé :

- Si supporté par un cloud provider : juste cliquer sur le bouton update available pour lancer
- Si déployé à l'aide de kubeadm : saisir les commandes [kubeadm upgrade plan] et [kubeadm upgrade apply]
- Pour le faire à la mains : installer les package à la main avec les apt et en récupérant les binaires

Il est également important de bien choisir la méthode à utiliser pour l'upgrade de kubernetes sur l'ensemble des nodes :

- Soit upgrader en simultané tous les worker : créera de l'indisponibilité au niveau des services
- Soit upgrader chaque nœud à son tour (drain, cordon pour le rendre indisponible)
- Déployer un nouveau nœud avec la nouvelle version, basculer les pods sur ce nouveau nœud là et après rendre indisponible tous les autres nœuds et procéder à leur upgrade.

```
kubeadm upgrade plan
[preflight] Running pre-flight checks.
[upgrade] Making sure the cluster is healthy:
[upgrade/config] Making sure the configuration is correct:
[upgrade/versions] Fetching available versions to upgrade to
[upgrade/versions] Cluster version: v1.11.8
[upgrade/versions] kubeadm version: v1.11.3
[upgrade/versions] Latest stable version: v1.13.4
[upgrade/versions] Latest version in the v1.11 series: v1.11.8

Components that must be upgraded manually after you have
upgraded the control plane with "kubeadm upgrade apply":
COMPONENT CURRENT AVAILABLE
Kubelet 3 x v1.11.3 v1.13.4

Upgrade to the latest stable version:

COMPONENT CURRENT AVAILABLE
API Server v1.11.8 v1.13.4
Controller Manager v1.11.8 v1.13.4
Scheduler v1.11.8 v1.13.4
Kube Proxy v1.11.8 v1.13.4
CoreDNS 1.1.3 1.1.3
Etcd 3.2.18 N/A

You can now apply the upgrade by executing the following command:
  kubeadm upgrade apply v1.13.4
Note: Before you can perform this upgrade, you have to update kubeadm to v1.13.4.
```

```
apt-get upgrade -y kubeadm=1.12.0-00
kubeadm upgrade apply v1.12.0
-
[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.12.0". Enjoy!
[upgrade/kubelet] Now that your control plane is upgraded, please proceed with
upgrading your kubelets if you haven't already done so.

kubectl get nodes
NAME STATUS ROLES AGE VERSION
master Ready master 1d v1.11.3
node-1 Ready <none> 1d v1.11.3
node-2 Ready <none> 1d v1.11.3

apt-get upgrade -y kubelet=1.12.0-00
systemctl restart kubelet
KODEKLoud
```

Sur un cluster installé à partir de kubeadm, lorsqu'on entre la commande [kubeadm upgrade plan], ce dernier nous affiche les possibilités d'upgrade comme nous pouvons le voir ci-dessus, toutefois il nous met en garde qu'il faudra déjà upgrade kubeadm vers cette nouvelle version avant de pouvoir upgrade ses composants et pour upgrade kubelet :

- Apt-get upgrade -y kubeadm=1.12.0-00 (le -00 de la fin est indispensable) après avoir terminé l'upgrade de kubeadm, on

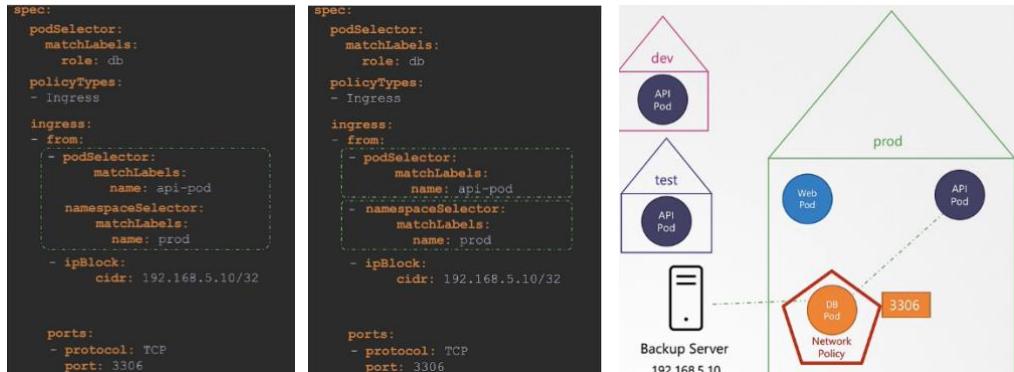
procède à l'upgrade de ses composants avec :

- Kubeadm upgrade apply v1.12.0

Il faut également par la suite mettre à jour le kubelet qui est l'agent kubelet indispensable au fonctionnement de kubernetes.

NB : répéter les mêmes actions sur les workers l'un après les autres en décommissionnant les pods et en mettant à jour le kubeadm (kube-proxy) et le kubelet des workers.

19) Network Policy



Les networks Policy sont des sortes de pare-feu ou Security groupe qu'on peut mettre sur les différents pods afin de filtrer le trafic réseau entrant (ingress) ou sortant (egress) vers ce pods via des ports particuliers.

NB : toutes les fonctions communes aux objets kubernetes s'appliquent également aux network Policy.

Solutions that Support Network Policies:

- Kube-router
- Calico
- Romana
- Weave-net

Solutions that DO NOT Support Network Policies:

- Flannel

20) INGRESS

Example - `kubectl create ingress ingress-test --rule="wear.my-online-store.com/wear*=wear-service:80"`

Ingress



Ingress

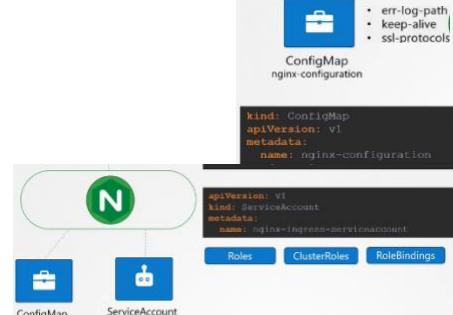


L'ingress est un composant Kubernetes qui nous permet d'exposer nos applications plus facilement et de façon professionnelle à nos clients. Pour cela il vous faut au préalable un ingress Controller (Nginx, HAProxy, traefik ...) et créer des ressources ingress à l'aide des manifests.

NB : il faut installer au préalable l'ingress Controller car il n'est pas installé par défaut dans le cluster.

INGRESS CONTROLLER

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-ingress
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    targetPort: 443
    protocol: TCP
    name: https
  selector:
    name: nginx-ingress
```

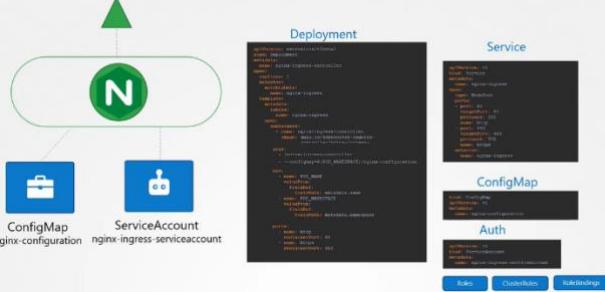


```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-ingress-controller
spec:
  replicas: 1
  selector:
    matchLabels:
      name: nginx-ingress
  template:
    metadata:
      labels:
        name: nginx-ingress
    spec:
      containers:
      - name: nginx-ingress-controller
        image: quay.io/kubernetes-ingress-controller/nginx-ingress-controller:0.21.0
      args:
      - /nginx-ingress-controller
      - --configmap=$(POD_NAMESPACE)/nginx-configuration
```

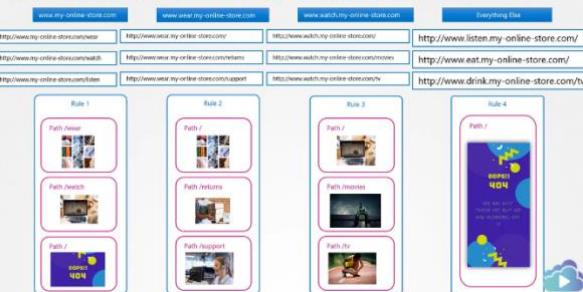
21) Ingress – Annotation and rewrite-target

Il s'agit d'une option que l'on peut configurer dans les metadata lors de la création d'une règle ingress. En effet c'est cette option qui empêche que l'adresse **hôte/path** soit entièrement retranscrite par **<service :port>/path** mais juste part

INGRESS CONTROLLER



INGRESS RESOURCE - RULES



```
args:
  - /nginx-ingress-controller
  --configmap=${POD_NAMESPACE}/nginx-ingress-controller
env:
  - name: POD_NAME
    valueFrom:
      fieldRef:
        fieldPath: metadata.name
  - name: POD_NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:
  rules:
    - http:
        paths:
          - backend:
              serviceName: wear-service
              servicePort: 80
          - path: /watch
            backend:
              serviceName: watch-service
              servicePort: 80

```

<service :port>/. C'est-à-dire que le path fournit ici n'est pas retranscrit dans le site web, il est juste utilisé par l'ingress pour déterminer le service vers lequel il doit rediriger le trafic.

`http://<ingress-service>:<ingress-port>/watch` → `http://<watch-service>:<port>/`

`http://<ingress-service>:<ingress-port>/watch` → `http://<watch-service>:<port>/`

Par défaut, la valeur du paramètre rewrite-target est «/» ce qui permet de renvoyer juste sur le service en question. Néanmoins, il est possible de réécrire ou de modifier cette valeur afin de changer le lien qui sera renvoyé dans le navigateur fonction de votre application web.

<code>replace("/path","/")</code>	<code>replace("/something(/ \$)(.*)", "/\$2")</code>
<pre>apiVersion: extensions/v1beta1 kind: Ingress metadata: name: test-ingress namespace: critical-space annotations: nginx.ingress.kubernetes.io/rewrite-target: /</pre>	<pre>apiVersion: extensions/v1beta1 kind: Ingress metadata: annotations: nginx.ingress.kubernetes.io/rewrite-target: /\$2 name: rewrite namespace: default spec: rules: - host: rewrite.bar.com http: paths: - backend: serviceName: http-svc servicePort: 80 path: /something(/ \$)(.*) </pre>

22) Ingress TLS

- Create certificate (for a test, an auto sign certificate can be used)
- Create secret tls which contain certificate and key
- Create or edit ingress rule to add tls section (tls > secretName)
-

#creation du certificat

```
export DOMAIN=$(minikube ip).nip.io
openssl req -x509 -newkey rsa:4096 -sha256 -nodes -keyout tls_self.key -out tls_self.crt -subj "/CN=*.${DOMAIN}" -days 365
```

```
SECRET_NAME=$(echo $DOMAIN | sed 's/\.-/g')-tls; echo $SECRET_NAME
kubectl create secret tls $SECRET_NAME --cert=tls_self.crt --key=tls_self.key
```

23) Docker Service Configuration and securing

Lorsque vous installer docker sur votre machine, cela installe en effet 02 composants principaux qui sont :

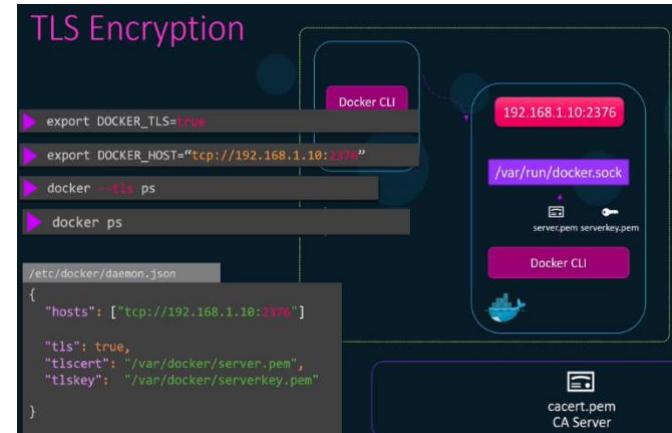
- **Le docker CLI** qui permet d'interagir avec le daemon docker grâce aux commandes docker
- **Le daemon Docker** qui est en quelque sorte le serveur docker représenté par le socket docker /var/run/docker.sock. Le daemon docker est configurable grâce à la commande dockerd.

Le daemon docker étant donc le serveur docker, est accessible par défaut juste en local à l'aide du docker CLI installé également sur la même machine en local. Néanmoins, il est possible de configurer le daemon docker afin qu'il puisse être accessible de l'extérieur à l'aide d'un docker CLI installé sur une autre machine dans le même réseau. Il est déconseillé d'exposer son daemon docker à l'extérieur, car par défaut également il n'est pas sécurisé et une fois exposé, peut être accessible par tout le monde et donne tous les priviléges (c'est-à-dire vous pouvez tout y faire une fois connecté). Trouvez ci-dessous les commandes ou configurations permettant d'exposer votre docker daemon, de crypter les échanges avec votre daemon docker via TLS, de configurer une vérification d'identité au préalable à l'aide des certificats avant d'autoriser les connexions au daemon docker.

Pour rendre notre démon docker accessible de l'extérieur, vous devez créer ou modifier le fichier de configuration docker `/etc/docker/daemon.json`

Et ajouter la ligne dans le fichier au format JSON `{"hosts": ["tcp://192.168.1.10:2375"]}` cette ligne dans le fichier de configuration de docker permet à votre serveur ou daemon docker d'être accessible via `<@IP>:<port>` renseignée. Mais cette configuration ne permet pas d'encrypter ou de sécuriser les communications.

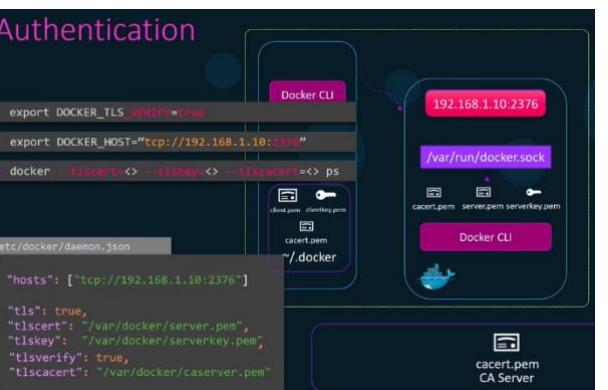
Pour crypter les communications avec le daemon docker, il faut configurer une autorité de certificat et créer un certificat avec une clé privée que l'on doit fournir au daemon docker afin qu'il les utilise pour crypter les communications. **NB** : une fois les communications cryptées, le port passe de 2375 à 2376, mais cela ne sécurise pas pour autant votre serveur docker, car il est toujours accessible par tout le monde sans vérification d'identité préalable. Pour y accéder, il suffira juste de mettre l'option `--tls=true` dans votre commande docker ou alors d'exporter les variables d'environnement docker qui vont bien



Pour activer la vérification d'identité avant d'autoriser la connexion à votre daemon docker, il faut ajouter l'option `"tlscacert": true` dans votre fichier de configuration et également lui fournir le certificat de l'autorité de certification "tlscacert" utilisé au préalable pour crypter les communications, il devra utiliser ce certificat pour vérifier les certificats clients des machines distantes qui essayeront de s'y connecter. Pour se connecter dorénavant, il faudra au préalable créer un certificat + clé client qui sera signé par la même autorité de certification que l'on attribuera à la machine cliente. L'attribution du certificats peut se faire soit dans la ligne de commande docker ou alors en copiant les certificats et clé dans le répertoire qui va bien (répertoire `/home/.docker` ou `~/.docker`)

NB : il est important en plus de sécuriser votre docker daemon de bien sécuriser également le serveur sur lequel est installé docker en

- Supprimant l'authentification par mot de passe et activer juste l'authentification par SSH
- Désactiver le compte root
- Contrôler qui a accès au serveur et désactiver les ports inutilisés.



Summary

```

/etc/docker/daemon.json
{
  "hosts": ["tcp://192.168.1.10:2376"]
  "tls": true,
  "tlscert": "/var/docker/server.pem",
  "tlskey": "/var/docker/serverkey.pem"
}

```

`docker --tls ps`

Without Authentication

```

/etc/docker/daemon.json
{
  "hosts": ["tcp://192.168.1.10:2376"]
  "tls": true,
  "tlscert": "/var/docker/server.pem",
  "tlskey": "/var/docker/serverkey.pem",
  "tlsv1": true,
  "tlscacert": "/var/docker/caserver.pem"
}

docker --tlscacert --tlsv1 --tlscert --tlskey --tlscacert --ps

```

With Authentication

<code>dockerd --debug</code>	Lancer le daemon docker
<code>INFO[2020-10-24T08:29:00.331925176Z] Starting up</code> <code>INFO[2020-10-24T08:29:00.332463283Z] parsed scheme: "unix"</code> <code>INFO[2020-10-24T08:29:00.375510773Z] scheme "unix" not registered</code> <code>INFO[2020-10-24T08:29:00.375657667Z] ccResolverWrapper: sending connection to: 127.0.0.1:2376</code>	
<code>dockerd --debug</code> <code>--host=tcp://192.168.1.10:2376</code> <code>--tls=true</code> <code>--tlscert=/var/docker/server.pem</code> <code>--tlskey=/var/docker/serverkey.pem</code>	Lancer le daemon docker en mode debug avec plus de logs
<code>dockerd --debug</code> <code>--host=tcp://192.168.1.10:2376</code> <code>--tlscacert=/var/docker/caserver.pem</code> <code>--tlsv1</code> <code>--tlscert=/var/docker/server.pem</code> <code>--tlskey=/var/docker/serverkey.pem</code>	Lancer le daemon docker en en ajoutant les options nécessaires dans la ligne de commande

```

dockerd --debug=false
unable to configure the Docker daemon with file /etc/docker/daemon.json: the following directives are specified both as a flag and in the configuration file: debug: (from flag: false, from file: true)

```

On peut également créer un fichier de configuration dans lequel on mettra les paramètres pour les options souhaitées

Attention aux conflits pouvant survenir si jamais vous passez des paramètres en ligne de commande différents de ceux présents dans le fichier `daemon.json`

III) SYSTEM HARDENING

Afin de sécuriser son environnement, il est important de respecter le principe de moindre privilège pour les différents utilisateurs que ce soient des serveurs linux ou du cluster Kubernetes.



3.1) Minimize host OS footprint (Reduce the Attack Surface)

Pour limiter la surface d'attaque d'un serveur ou cluster, il faut juste mettre le système dans un état simple et consistant en respectant les principes suivants :

- Utiliser le principe du moindre privilège pour les autorisations
- Supprimer les applications obsolètes du système
- Limiter au maximum les accès
- Supprimer les services obsolètes
- Restreindre les modules Kernel obsolètes
- Identifier et bloquer les ports ouverts inutilement

3.2) Limit Node Access

Pour limiter l'accès aux différents nœuds d'un cluster, il faudrait déjà se rassurer que ces nœuds ne soient pas disponibles sur internet. Si jamais il s'agit de machines déployées dans le cloud, il faudrait de préférences les avoir créés dans un réseau privé accessible par VPN ou alors il faudrait configurer un pare-feu dessus n'autorisant l'accès depuis internet qu'à une plage d'adresse spécifique. Ensuite, il faut décider de qui pourra avoir accès en SSH à ces différents nœuds ?, généralement ce sont juste les administrateurs car les développeurs ou les utilisateurs finaux n'ont pas besoin d'un accès SSH.

Il existe 04 types de comptes sous Linux :



- les User Account : Utilisateurs normaux
- Le Superuser Account : utilisateur root à l'UID=0
- Les System Accounts : comptes systèmes créés automatiquement durant l'installation de l'OS
- Les Service Accounts : compte d'application créé manuellement ou automatiquement pour des applications particulières.

```
▶ id  
uid=1000(michael) gid=1000(michael) groups=1000(michael) 1010(admin)
```

La commande « `id` » donne les informations sur l'utilisateur connecté

```
▶ who  
michael pts/2 Apr 28 06:48 (172.16.238.187)
```

La commande « `who` » permet d'avoir la liste des utilisateurs actuellement connectés au système

```
▶ last  
michael :1 :1 Tue May 12 20:00 still logged in  
sarah :1 :1 Tue May 12 12:00 still running  
reboot system boot 5.3.0-758-gen Mon May 11 13:00 - 19:00 (06:00)
```

La commande « `last` » permet d'avoir les informations sur la dernière connexion de chaque utilisateur

Les fichiers permettant de configurer les accès sous Linux sont :

/etc/passwd

```
▶ grep -i ^michael /etc/passwd  
michael:x:1001:1001::/home/michael:/bin/bash
```

Ce fichier contient les informations basiques sur les users du système (nom utilisateur, groupe, bash ...)

/etc/shadow

```
▶ grep -i ^michael /etc/shadow  
michael:$6$0h0ut0t0$5JcuRxR7y72LLQk4Kdog7u09L  
YcWF/7.eJ3TfGfG01j4JF63PyuPwKC18tJS.:18188:0:
```

Ce fichier contient les mots de passe cryptés des différents utilisateurs du système

/etc/group

```
▶ grep -i ^bob /etc/group  
developer:x:1001:bob,michael
```

Fichier contenant les informations sur les groupes du système, le nom, leur identifiant unique ...

Il faut ensuite se servir de ces fichiers et des commandes vues précédemment pour vérifier les informations sur les users du système et se rassurer que le principe de moindre privilège soit bien respecté. Les commandes de manipulation des users sont les suivantes :

- **`usermod -s /bin/nologin michael`** : permet de désactiver un compte d'utilisateur en remplaçant son Bash par défaut par le /bin/nologin.

```
▶ usermod -s /bin/nologin michael
```

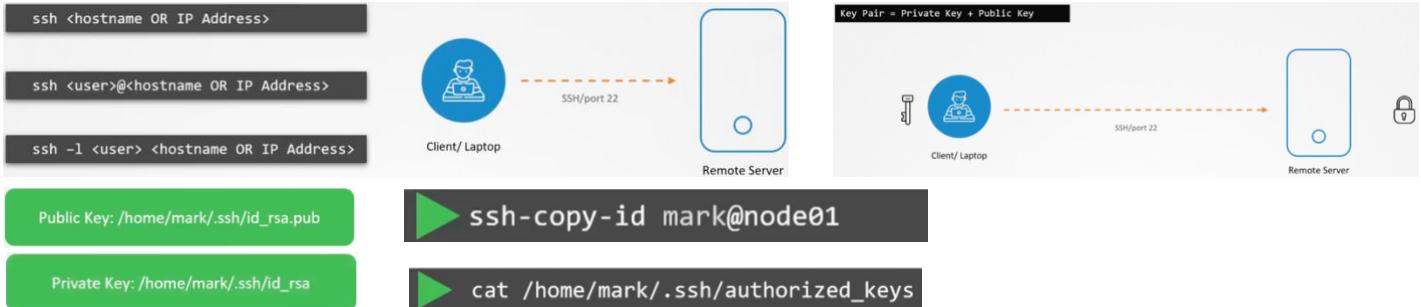
- **`userdel bob`** : Supprimer l'utilisateur bob

```
▶ grep -i bob /etc/passwd
```

- **`deluser michael admin`** : retire l'utilisateur michael du groupe admin

```
▶ deluser michael admin
Removing user `michael` from group `admin` ...
Done.
```

3.3) SSH Hardening



Pour désactiver la connexion SSH avec le compte root et également désactiver la connexion SSH par mot de passe, il faut configurer le service SSH en éditant son fichier de configuration `/etc/ssh/sshd_config`

Après avoir édité les paramètres souhaités, redémarrer le service.

```
▶ vi /etc/ssh/sshd_config
PermitRootLogin no
PasswordAuthentication no
```

```
▶ systemctl restart sshd
```

3.4) Privilege Escalation in Linux

Le fichier par défaut pour la configuration de l'escalation de privilège via sudo dans Linux est `/etc/sudoers`. Ce fichier peut être configuré ou modifié à l'aide de la commande « `visudo` »

```
▶ cat /etc/sudoers
User privilege specification
root    ALL=(ALL:ALL) ALL
# Members of the admin group may gain root privileges
%admin  ALL=(ALL:ALL) ALL
# Allow members of group sudo to execute any command
%sudo   ALL=(ALL:ALL) ALL
# Allow Bob to run any command
mark    ALL=(ALL:ALL) ALL
# Allow Sarah to reboot the system
sarah   localhost=/usr/bin/shutdown -r now
# See sudoers(5) for more information on "#include"
directives:
#include /etc/sudoers.d
```

Field	Description	Example
1	User or Group	bob, %sudo (group)
2	Hosts	localhost, ALL(default)
3	User	ALL(default)
4	Command	/bin/ls, ALL(unrestricted)

Dans cet exemple, l'utilisateur mark a tous les priviléges du super admin tandis que Sarah ne pourra que redémarrer l'hôte à l'aide du sudo.

3.5) Remove Obsolete Packages and Services

Il est important de toujours vérifier les applications installées sur les différents nœuds de notre cluster et se rassurer que ce ne sont que les applications utiles ou nécessaires qui sont installées.

De la même façon qu'avec les applications, il est également indispensable de toujours vérifier et également supprimer tous les services non nécessaires présents dans notre système. Les commandes usuelles pour la manipulation des services :

- **`systemctl list-units --type service`** : permet de lister tous les services présents sur notre machine (Si on se rend compte qu'un service n'est pas nécessaire, il faut tout de suite le stopper et le désinstaller ou le désactiver).

- **`apt list --installed`** : permet de lister tous les paquets installés sur un hôte Linux

```
▶ systemctl list-units --type service
apache2.service          loaded active running The Apache HTTP Server
apparmor.service          loaded active exited AppArmor initialization
containerd.service        loaded active running containerd container runtime
dbus.service              loaded active running D-Bus System Message Bus
docker.service            loaded active running Docker Application Container Engine
ebtables.service          loaded active exited ebtables ruleset management
kmod-static-nodes.service loaded active exited Create list of required static device n
kubelet.service           loaded active running kubelet: The Kubernetes Node Agent
proxy.service             loaded active running kubectl proxy 8888
systemd-journal-flush.service loaded active exited Flush Journal to Persistent Storage

▶ systemctl stop apache2
▶ systemctl disable apache2
Synchronizing state of apache2.service with
install.

▶ apt remove apache2
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

3.6) Restrict Kernel Modules

Pour des raisons de sécurité, il est également important de restreindre l'utilisation de certains modules Kernel dans Linux. Ces modules kernel correspondent en réalité aux différents hardware connectés sur l'hôte ; ces modules peuvent être chargé automatiquement à la connexion du nouveau matériel, ou alors manuellement par un utilisateur au privilège admin. Les commandes usuelles sont :

- **modprobe pcspkr** : permet de charger manuellement le module kernel pcspkr (pc speaker)
- **lsmod** : lister l'ensemble des modules chargé sur le système
-

Après avoir listé la liste des modules, on peut décider de bloquer le chargement de certains modules pas utiles pour le système, pour cela, il suffit de blacklister ces modules en les mettant dans le fichier **/etc/modprobe.d/blacklist.conf**

Après avoir créé le fichier dans le répertoire **/etc/modprobe.d/**, il faut redémarrer l'hôte et vérifier que le module n'aît plus été chargé à l'aide de **lsmod**.

```
▶ cat /etc/modprobe.d/blacklist.conf
blacklist sctp
blacklist dccp
▶ shutdown -r now
▶ lsmod | grep dccp
```

3.7) Identify and disable open ports

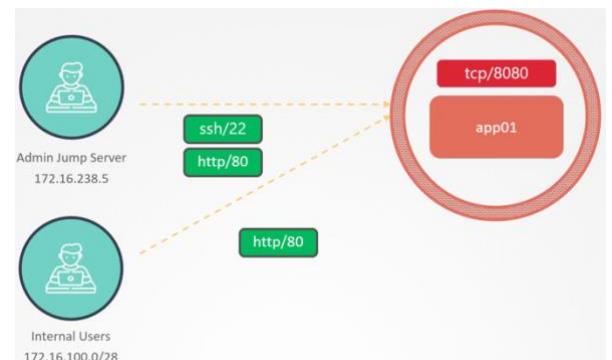
Pour identifier les ports ouverts et checker les services utilisant ces ports, on peut utiliser les commandes suivantes :

- **netstat -an | grep -w LISTEN** : affiche la liste de tous les ports ouverts
- **cat /etc/services | grep -w 53** : affiche le service utilisant le port 53

3.8) UFW Firewall Basics

Il s'agit du pare feu basique de Linux qui permet de contrôler les différents ports du système afin de limiter les accès réseau. Par exemple autoriser l'accès à un port spécifique de la machine juste à un seul hôte ou encore à une plage d'adresse.

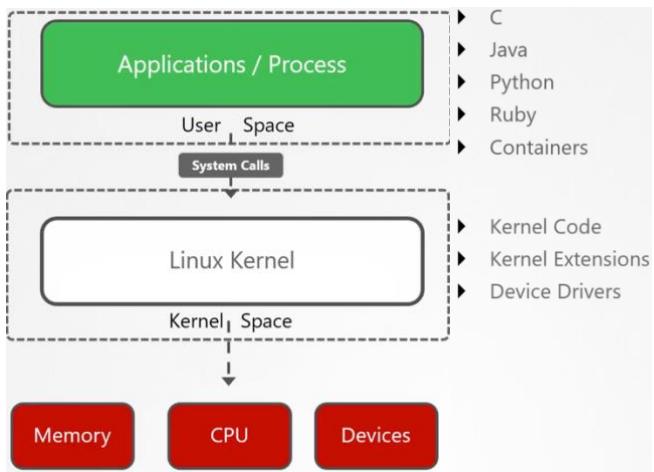
iptables	ufw (Uncomplicated Firewall)
▶ apt-get update	▶ systemctl enable ufw
▶ apt-get install ufw	▶ systemctl start ufw



Les commandes usuelles du pare feu « ufw » sont les suivantes :

- **ufw status** : pour avoir le statut du pare-feu (actif ou inactif)
- **ufw default allow outgoing** : autoriser par défaut toutes les connexions sortantes
- **ufw default deny incoming** : refuser par défaut toutes les connexions entrantes (valables pour toutes les nouvelles connexions, celles déjà établies sont maintenues).
- **ufw allow from 172.16.238.5 to any port 22 proto tcp** : autoriser les connexions entrantes sur le port 22 seulement pour l'hôte défini.
- **ufw allow from 172.16.100.0/28 to any port 80 proto tcp** : autoriser les connexions entrantes sur le port 80 seulement pour la plage d'adresse définie
- **ufw allow 1000 :2000/tcp** : ouvre la plage de ports entre 1000 et 2000 y compris
- **ufw deny 8080** : refuse les connexions sur le port 8080 pour tout le monde
- **ufw enable** : activer le pare-feu, bien entendu après avoir ajouté les précédentes règles
- **ufw delete deny 8080** : permet de supprimer la règle liée à la commande « deny 8080 »
- **ufw delete 5** : permet de supprimer la règle d'index 5 (numéro du haut vers le bas de la liste des règles existantes avec un « ufw status »)

3.9) Linux SYSCALLS



Ci-après une image qui montre l'architecture d'un système d'exploitation LINUX. En effet l'élément central de ce système d'exploitation est le Kernel. Le Kernel est divisé en deux espaces mémoires qui sont :

- **User Space** : espace mémoire dédié aux applications ou processus lancés par des utilisateurs

- **Kernel Space** : espace mémoire dédié aux composants du kernel (le code kernel, les extensions ou les drivers des équipements).

Les applications ou processus lancés par des utilisateurs se trouvant dans le User Space, pour fonctionner, on besoin soit de la mémoire, du CPU ou alors de certains équipements (Hardwares, disques, données...) et donc pour avoir accès à ces ressources, ces applications dans le User Space doivent faire des requêtes spéciales au Kernel situé dans le Kernel Space. Ces requêtes spéciales entre les applications et le Kernel sont appelés « **System Calls** »).

Il existe des commandes Linux pour tracer (lister) ou interagir avec les System Calls effectués par un processus.

- **strace touch /tmp/error.log** : la commande strace est une commande par défaut déjà installée sur la plupart des distribution Linux, elle permet d'inspecter l'ensemble des SYSCALLS effectués par un processus. Dans notre exemple, elle nous permettra de lister les Sys calls nécessaires au processus de création d'un fichier

- **strace -c touch /tmp/error.log** : Permet d'afficher la liste exhaustive de tous les Sys calls utilisés par la commande passée en paramètre

```
▶ strace touch /tmp/error.log
execve("/usr/bin/touch", ["touch", "/tmp/error.log"], 0x7ffce8f874f8 /* 23 vars */) = 0
```

execve() est en fait un system call qui prend en paramètres des arguments afin de réaliser l'action demandée.

- **pidof etcd** : permet d'avoir l'ID du processus etcd, nous avons besoin de l'ID d'un processus afin d'utiliser strace dessus afin d'avoir la liste des Sys calls dudit processus.

- **strace -p <pid number>** : affiche la liste des system call utilisé par le processus d'ID passé en paramètre.

```
▶ pidof etcd
3596
▶ strace -p 3596
strace: Process 3596 attached
futex(0x1ac6be8, FUTEX_WAIT_PRIVATE, 0, NULL) = 0
futex(0xc000540bc8, FUTEX_WAKE_PRIVATE, 1) = 1
```

3.10) Aquasec Tracee

Tracee est un outil Open Source de Aqua Security qui permet de tracer les Sys calls générés par un container. Tracee utilise la technologie eBPF(Extended Berkeley Packet Filter) pour scanner le système en cours d'exécution. La technologie eBPF peut en effet lancer des processus directement dans le Kernel Space sans interférer ou générer de conflit avec le code kernel. eBPF est donc utilisé pour créer l'outil Tracee de Aqua Security permettant de monitorer L'OS et détecter les comportements suspects. Il est possible d'installer Tracee directement sur le serveur en installant ses paquets et dépendances, mais afin d'éviter les problèmes de dépendances et autres, il est fortement recommandé de lancer ou d'installer Tracee à partir d'un container Docker.



Pour lancer Tracee dans un container, il y'a des prérequis nécessaires du au fait que Tracee utilise la technologie eBPF.
Ces prérequis sont les suivants :

- Faire un Bind Mount entre le **/tmp/tracee** du container et celui de l'hôte ; il s'agit du répertoire par défaut contenant le programme pour la technologie eBPF
- Faire un bind mount entre le container et l'hôte du répertoire **/lib/modules** qui contient en effet les fichiers kernel nécessaires à Tracee pour compiler le programme eBPF
- Faire un bind mount entre le container et l'hôte du répertoire **/usr/src** qui contient également les dépendances des modules kernel nécessaires à l'exécution du programme eBPF (NB ces bind mount doivent être fait en mode Read Only)
- Le container Tracee a également besoin de privilèges administrateurs additionnels pour fonctionner ; raison pour laquelle il faut utiliser l'option « -- privileged »

Ci-dessous quelques exemples de commandes de lancement de container Tracee :

```
▶ docker run --name tracee --rm --privileged --pid=host \
-v /lib/modules/:/lib/modules/:ro -v /usr/src:/usr/src:ro \
-v /tmp/tracee:/tmp/tracee aquasec/tracee:0.4.0 --trace comm=ls
```

TIME(s)	UID	COMM	PID	TID	RET	EVENT
1263.457188	0	ls	27461	27461	-2	access
1263.457218	0	ls	27461	27461	-2	access
1263.457238	0	ls	27461	27461	0	security_file_open

Container Tracee, affichant tous les Sys call invoqués lors de l'exécution de la commande « ls »

Pour avoir les sys calls invoqués pour tout nouveau processus, on peut lancer le container tracee suivant :

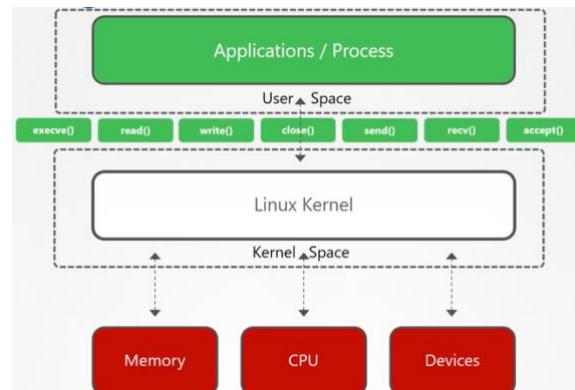
```
▶ sudo docker run --name tracee --rm --privileged --pid=host \
-v /lib/modules/:/lib/modules/:ro -v /usr/src:/usr/src:ro \
-v /tmp/tracee:/tmp/tracee aquasec/tracee:0.4.0 --trace pid=new
```

1613.769845	0	wc	1619	1619	-2	openat
langpack/en/LC_MESSAGES/coreutils.mo						
1613.846148	0	kubectl	1617	1621	-2	openat
flags: O_RDONLY O_CLOEXEC						
1613.848222	0	kubectl	1617	1621	-2	openat

Quant aux sys calls appellés par tou nouvel container on peurt lancer le container tracee suivant : Au lancement d'un nouvel container, bien entendu dans un autre invite de commande, on observe les sys calls suivants en sortie.

```
▶ sudo docker run --name tracee --rm --privileged --pid=host \
-v /lib/modules/:/lib/modules/:ro -v /usr/src:/usr/src:ro \
-v /tmp/tracee:/tmp/tracee aquasec/tracee:0.4.0 --trace container=new
```

.	.	.				
821.928334	3b392a8f3c57	0	echo	1	/12719 1	/12719 0
821.928354	3b392a8f3c57	0	echo	1	/12719 1	/12719 0
821.928551	3b392a8f3c57	0	echo	1	/12719 1	/12719 0



3.11) Restrict Sys Calls with SECCOMP

Il est important pour des raisons de sécurité, ne n'autoriser que les Sys Calls qui sont indispensables pour le bon fonctionnement de notre système. Nous avons vu précédemment qu'une simple commande « touch » peut invoquer une multitude de Sys Calls parmi les plus de 435 Sys Calls existants à ce jour sous Linux. Par défaut, le kernel Linux autorise l'appel de n'importe quel Sys Calls par toutes les applications ou processus du User Space. C'est la raison pour laquelle on utilise l'outil SECCOMP(Secure Computing) pour encapsuler les applications afin qu'elles ne puissent utiliser que les Sys Calls dont elles ont besoin.

SECCOMP a été introduit en 2005 et a été intégré à la plupart de noyau Kernel Linux. Pour vérifier que votre OS Linux supporte SECCOMP, il faut vérifier si votre fichier de configuration du boot (boot configuration) contient le mot SECCOMP. La commande vérification est la suivante :

- `grep -i seccomp /boot/config-$(uname -r)` : affiche toutes les lignes du fichier de boot contenant le mot SECCOMP.

Si ce fichier contient la ligne `CONFIG_SECCOMP=y`, alors votre kernel supporte le SECCOMP.

Par défaut, pour des raisons de sécurité, Docker bloque plus de 60 Sys calls parmi les plus de 300 existants, raison pour laquelle en se connectant à un container, il y'a des commandes qui ne sont pas permises comme

- le changement de la date
- le redémarrage de la machine...

```
grep -i seccomp /boot/config-$(uname -r)
CONFIG_HAVE_ARCH_SECCOMP_FILTER=y
CONFIG_SECCOMP_FILTER=y
CONFIG_SECCOMP=y
```

```
docker run -it --rm docker/whalesay /bin/sh
#
# date -s '19 APR 2012 22:00:00'
date: cannot set date: Operation not permitted
```

Pour avoir l'état du SECCOMP pour un processus, il faut dans un premier temps récupérer l'ID du processus grâce à la commande «`ps -ef`», et ensuite chercher le mot Seccomp dans le fichier de status de ce processus se trouvant au `/proc/pid/status`

On se rend compte que la valeur du Seccomp pour le processus `/bin/sh` lancé dans ce container est 2.

```
ps -ef
UID          PID    PPID   C STIME TTY
root           1      0  0 15:44 pts/0
root          12      1  0 15:47 pts/0

grep Seccomp /proc/1/status
Seccomp:          2
```

Le SECCOMP peut opérer sous trois modes :

- **Mode 0** : Dans ce mode le Seccomp est désactivé
- **Mode 1** : Le Seccomp est appliqué dans un mode strict ou restreint (qui bloque tous les Sys calls exceptés 04 qui sont le Read(), write(), exit(), execve())
- **Mode 2** : Mode filtré, ici le Seccomp sélectionne les Sys Calls à autoriser ou à refuser fonction de la configuration qui lui est donnée.



La configuration du filtrage Seccomp se fait au travers d'un fichier JSON qui peut prendre la forme de :

- Whitelist : On liste un certain nombre de Sys Calls pour lesquels on autorise l'accès et tous les autres sont bloqués
- Blacklist : On liste un certain nombre de Sys Calls à rejeter et tous les autres sont autorisés.

Dans un fichier de configuration Seccomp, on a 3 blocs ou paramètres généraux à définir :

- Les Architectures : définir la liste des architectures prises en compte par le profile Seccomp
- Les Sys calls : la liste des Sys calls à refuser ou autoriser et à l'afin définir l'action à faire (autoriser ou refuser) pour ladite liste
- L'action par défaut : Action (refuser ou autoriser) pour tout autre Sys Call n'appartenant pas à la liste définie plus bas.

```
whitelist.json
{
  "defaultAction": "SCMP_ACT_ERRNO",
  "architectures": [
    "SCMP_ARCH_X86_64",
    "SCMP_ARCH_X86",
    "SCMP_ARCH_X32"
  ],
  "syscalls": [
    {
      "names": [
        "<syscall-1>",
        "<syscall-2>",
        "<syscall-3>"
      ],
      "action": "SCMP_ACT_ALLOW"
    }
  ]
}
```

```
blacklist.json
{
  "defaultAction": "SCMP_ACT_ALLOW",
  "architectures": [
    "SCMP_ARCH_X86_64",
    "SCMP_ARCH_X86",
    "SCMP_ARCH_X32"
  ],
  "syscalls": [
    {
      "names": [
        "<syscall-1>",
        "<syscall-2>",
        "<syscall-3>"
      ],
      "action": "SCMP_ACT_ERRNO"
    }
  ]
}
```

Il est possible d'utiliser des filtres Seccomp personnalisés lors du lancement d'un container afin de bloquer ou autoriser certains Sys Calls.

```
docker run -it --rm --security-opt seccomp=/root/custom.json \
docker/whalesay /bin/sh
/ #
/ # mkdir test
mkdir: can't create directory 'test': Operation not permitted
```

Il est également possible d'autoriser tous les Sys calls lors du lancement d'un container :

```
docker run -it --rm --security-opt seccomp=unconfined docker/whalesay /bin/sh
```

3.12) SECCOMP in Kubernetes

Il existe une image docker amicontained qui permet de lancer un container et fournir un ensemble d'informations d'exécution du container parmi lesquelles la plus importante, la liste des Sys calls bloqués.

```
▶ docker run r.j3ss.co/amicontained amicontained
Container Runtime: docker
Has Namespaces:
  pid: true
  user: false
AppArmor Profile: docker-default (enforce)
Capabilities:
  BOUNDING -> chown dac_override fowner fsetid kill setgid setuid setpcap net_bind_service net_raw
  sys_chroot mknod audit_write setfcap
Seccomp: filtering

Blocked Syscalls (64):
  MSGRCV SYSLOG SETPGID SETSID USELIB USTAT SYSFS VHANGUP PIVOT_ROOT _SYSCTL_ACCT_SETTIMEOFDAY MOUNT
  UMTIME2 SWAPON SWAPOFF REBOOT SETHOSTNAME SETDOMAINNAME IOPL IOPERM CREATE_MODULE INIT_MODULE
  DELETE_MODULE GET_KERNEL_SYMS QUERY_MODULE QUOTACTL NFSSERVCTL GETPMSG PUTPMSG AFS_SYSCALL TUXCALL
  SECURITY_LOOKUP_DCOOKIE CLOCK_SETTIME VSERVER MBIND SET MEMPOLICY GET_MEMPOLICY KEXEC LOAD ADD KEY
  REQUEST_KEY KEYCTL MIGRATE_PAGES UNSHARE MOVE_PAGES PERC_EVENT_OPEN_FANOTIFY_INIT NAME_TO_HANDLE_AT
  OPEN_BY_HANDLE_AT CLOCK_ADJTIME SETNS PROCESS_VM_READV PROCESS_VMWRITEV KCMP_FINIT_MODULE_KEXEC_FILE_LOAD
  BPF USERFAULTFD MEMBARRIER_PKEY_MPROTECT PKEY_ALLOC PKEY_FREE RSEQ
Looking for Docker.sock
```

En lançant la même image sur Kubernetes, on a les retours suivants :

```
▶ kubectl run amicontained --image r.j3ss.co/amicontained amicontained -- amicontained
pod/test created

▶ kubectl logs amicontained
Container Runtime: docker
Has Namespaces:
  pid: true
  user: false
AppArmor Profile: docker-default (enforce)
Capabilities:
  BOUNDING -> chown dac_override fowner fsetid kill setgid setuid setpcap
  net_bind_service net_raw sys_chroot mknod audit_write setfcap
  Seccomp: disabled

Blocked Syscalls (21):
  SYSLOG SETPGID SETSID VHANGUP PIVOT_ROOT_ACCT_SETTIMEOFDAY UMTIME2_SWAPON
  SWAPOFF REBOOT SETHOSTNAME SETDOMAINNAME INIT_MODULE_DELETE_MODULE_LOOKUP_DCOOKIE
  KEXEC_LOAD_FANOTIFY_INIT_OPEN_BY_HANDLE_AT_FINIT_MODULE_KEXEC_FILE_LOAD
  Looking for Docker.sock
```

On observe que le nombre de Syscalls bloqués est de 21 et également que le Seccomp est désactivé. En fait cela est due au fait que Kubernetes dans sa version actuelle 1.20 n'implémente pas le Seccomp par défaut.

Pour implémenter le Seccomp dans Kubernetes, nous sommes obligés d'utiliser un manifest de définition de ressource et de rajouter la propriété Seccomp dans la section Security context des specs du Pod.

```
pod-definition.yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: amicontained
    name: amicontained
spec:
  securityContext:
    seccompProfile:
      type: RuntimeDefault
  containers:
    - args:
        - amicontained
      image: r.j3ss.co/amicontained
      name: amicontained
      securityContext:
        allowPrivilegeEscalation: false
```

Ici on utilise le bloc `security Context` pour définir les paramètres liés au profile Seccomp à utiliser. On spécifie dans l'exemple ci-après le type de Seccomp : `RuntimeDefault` qui correspond en effet au profil Seccomp par défaut de Docker qui bloque automatiquement 64 Syscalls.

Par la suite il est important d'avoir dans chaque container un autre bloc `securityContext` avec un paramètre permettant de refuser l'escalation de privilège, car parfois certains processus dans des container peuvent recourir à l'escalation de privilège pour utiliser des Syscalls interdits par le profil Seccomp défini plus haut. `allowPrivilegeEscalation: false`; cela nous rassure également que les applications ou processus du container

n'utiliseront que le strict minimum de privilèges qu'ils ont besoin pour fonctionner correctement.

Il est également possible d'utiliser le type Seccomp « Unconfined » afin de désactiver manuellement le Seccomp.

```
spec:
  securityContext:
    seccompProfile:
      type: Unconfined
  containers:
```

Il est également possible d'utiliser un fichier Seccomp personnalisé comme nous avons vu précédemment, pour cela le type doit être Localhost, précisant ainsi que le fichier de configuration JSON du Seccomp sera chargé localement à partir d'une arborescence à définir. Ce path doit obligatoirement être relatif par rapport au path par défaut des fichiers de configurations Seccomp de Kubernetes qui est : `/var/lib/kubelet/seccomp`

```
securityContext:
  seccompProfile:
    type: Localhost
    localhostProfile: <path to the custom JSON file>
  containers:
```

`/var/lib/kubelet/seccomp`

Il est important afin de créer son propre fichier de configuration Seccomp, de créer un répertoire profiles dans le `/var/lib/kubelet/seccomp/profiles` et ensuite créer ses fichier JSON dans ce répertoire-là.

```
▶ mkdir -p /var/lib/kubelet/seccomp/profiles
/var/lib/kubelet/seccomp/profiles/audit.json
{
  "defaultAction": "SCMP_ACT_LOG"
}

▶ grep syscall /var/log/syslog
```

Nous pouvons alors utiliser l'outil Tracee pour sniffer tous les Syscalls appelés par tout nouveau container.

```
▶ test-audit.yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-audit
spec:
  securityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: profiles/audit.json
  containers:
    - command: ["bash", "-c", "echo 'I just made some syscalls' && sleep 100"]
      image: ubuntu
      name: ubuntu
      securityContext:
        allowPrivilegeEscalation: false
```

NB : Un Pod qui utilise un profile Seccomp qui bloque tous les Syscalls ne peut pas se lancer et va rester à l'état ContainerCreating car tous les Syscalls nécessaires au lancement du container sont également bloqués.

3.13) APPARMOR

AppArmor est un outil qui peut être utilisé pour limiter les capacités des applications Linux et par la même occasion, réduire ainsi la surface d'attaque. Seccomp permet en effet de bloquer l'accès à certains Syscalls de sorte à empêcher la création d'un répertoire ou nouveau fichier par exemple, néanmoins Seccomp ne peut pas empêcher un processus d'écrire dans un fichier ou répertoire par exemple. Pour implémenter un control granulaire sur les différents processus tournant dans un container, on utilise AppArmor. AppArmor est un outil de sécurité Linux qui est utilisé afin de confiner un processus à un set de ressources limité. AppArmor est installé par défaut dans la plupart des distributions Linux. Pour vérifier que AppArmor est bien installé et actif, on utilise la commande `systemctl status apparmor`.

Pour utiliser AppArmor, on doit d'abord se rassurer que le Kernel AppArmor est bien chargé sur tous les nœuds devant héberger des containers. Pour ce faire on doit vérifier ledit fichier pour chaque nœud et se rassurer que sa valeur soit Y (yes)

```
▶ cat /sys/module/apparmor/parameters/enabled
Y
```

Tout comme Seccomp, AppArmor est configuré à l'aide de fichier de config profiles qui doit être chargé dans le Kernel.

Ce fichier est accessible au `/sys/kernel/security/apparmor/profiles`.

Les profiles AppArmor sont de simples fichiers texte qui définissent quelles ressources peuvent être utilisées par une application.

```
apparmor-deny-write
profile[apparmor-deny-write]flags=(attach_disconnected) {
  file,
  # Deny all file writes.
  deny /* w,
}
```

Ce profil AppArmor a deux règles :

- La première autorise l'accès à tous les systèmes de fichiers
- La deuxième refuse l'accès en écriture à tous les fichiers du répertoire racine et de ses sous répertoires (`/*` : tous fichiers du répertoire racine et `/**` : tous fichiers du répertoire racine et sous répertoires).

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

Pour restreindre l'accès en écriture à tous les fichiers se trouvant juste dans le répertoire `/proc`

Afin de voir l'état des profil AppArmor sur un hôte, on utilise la commande `aa-status`. Un profil AppArmor peut avoir 03 modes de fonctionnement :

- **Enforce** : dans ce mode, AppArmor monitore et applique les règles définies dans les différents profils utilisés par ces applications.
- **Complain** : dans ce mode AppArmor autorise les applications à effectuer les taches sans la moindre restriction, mais enregistre ces taches comme étant des évènements
- **Unconfined** : AppArmor autorise les applications à tout faire mais ne garde pas de logs.

3.14) Creating APPARMOR profiles

Il existe sous Linux, un outil qui nous permet de créer facilement des profils AppArmor. La commande d'installation de cet outil sur Linux est : `apt-get install -y apparmor-utils`

Pour créer un profil AppArmor sur un script Bash par exemple, on exécute la commande `aa-gen` suivie du chemin absolu vers le script Bash et dans un autre terminal, on lance l'exécution dudit script ; puis on revient dans le précédent terminal afin de répondre aux questions de l'outil AppArmor quant aux différentes permissions à accorder fonctions des taches de noter script. Après avoir répondu aux questions ; entrez S pour sauvegarder le profil et ensuite F pour quitter

```
▶ aa-genprof /root/add_data.sh
root@kodekloud:~# aa-genprof /root/add_data.sh
Writing updated profile for /root/add_data.sh.
Setting /root/add_data.sh to complain mode.

Before you begin, you may wish to check if a
profile already exists for the application you
wish to confine. See the following wiki page for
more information:
https://gitlab.com/apparmor/apparmor/wikis/Profiles

Profiling: /root/add_data.sh

Please start the application to be profiled in
another window and exercise its functionality now.

Once completed, select the "Scan" option below in
order to scan the system logs for AppArmor events.

For each AppArmor event, you will be given the
opportunity to choose whether the access should be
allowed or denied.

[(S)can system log for AppArmor events] / (F)inish
```

```
Once completed, select the "Scan" option below in
order to scan the system logs for AppArmor events.

For each AppArmor event, you will be given the
opportunity to choose whether the access should be
allowed or denied.

[(S)can system log for AppArmor events] / (F)inish

.

.

Reading log entries from /var/log/syslog.
Updating AppArmor profiles in /etc/apparmor.d.

Profile: /root/add_data.sh
Execute: /usr/bin/mkdir
Severity: unknown

(I)nherit / (C)hild / (N)amed / (X) ix On / (D)eny / (A)bort / (T)imeout
```

Après avoir créé ce profil, on peut constater qu'il s'est ajouté automatiquement aux profils existants se trouvant dans le `/etc/apparmor.d/` `cat /etc/apparmor.d/root.add_data.sh` et concerne le script `add_data.sh` se trouvant dans le répertoire `root`. Pour tester si le profil fonctionne bien, on peut modifier le répertoire à créer dans le script et l'exécuter, on se rendra compte qu'il sera bloqué car ce script n'a pas le droit de créer un autre répertoire que `/opt/app` comme défini dans son profil AppArmor.

```
▶ cat add_data.sh
#!/bin/bash
data_directory=/opt
mkdir -p ${data_directory}
echo "> File created at `date`" | tee ${data_directory}/create.log

▶ ./add_data.sh
./add_data.sh
tee: /opt/create.log: Permission denied
=> File created at Mon 22 Mar 2021 04:04:47 PM EDT
```

Il existe également des commandes AppArmor qui permettent de travailler avec des profils existants :

- **apparmor_parser /etc/apparmor.d/root.add_data.sh** : permet de charger dans le kernel le profil existant se trouvant dans le répertoire par défaut des profils `/etc/apparmor.d` . les profils chargés dans le Kernel sont visibles dans le fichier `/sys/kernel/security/apparmor/profiles` en faisant un simple `cat` dessus.
- **apparmor_parser -R /etc/apparmor.d/root.add_data.sh** : permet de désactiver un profil, mais après il faut créer un lien de ce profil vers le répertoire disable grâce à la commande ci-dessous
- **In -s /etc/apparmor.d/root.add_data.sh /etc/apparmor.d/disable/**

3.15) APPARMOR in Kubernetes

AppArmor permet de sécuriser les déploiements Kubernetes en limitant ce que les containers peuvent ou ne peuvent pas faire. Les prérequis pour pouvoir exécuter AppArmor dans Kubernetes sont :

- Le module kernel AppArmor doit être activé sur tous les nœuds
- Les profils AppArmor doivent être chargé dans le Kernel
- Le container runtime (containerd, cri-docker...) doit être supporté par AppArmor.

Pour appliquer un profil AppArmor sur un pod, on se rassure au préalable garce à la commande `aa-status` que ledit profil est bien chargé dans le kernel de l'ensemble des nœuds du cluster et ensuite, on utilise un manifest Kubernetes de Pod dans lequel on ajoute une annotation. L'annotation utilisée pour ajouter un profil AppArmor à un pod est :

`container.apparmor.security.beta.kubernetes.io/<container_name>: localhost/<AppArmor_profileName>`

annotations:

```
{ container.apparmor.security.beta.kubernetes.io/<container_name>: localhost/<profile-name> }
```

```
▶ ubuntu-sleeper.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper
  annotations:
    container.apparmor.security.beta.kubernetes.io/ubuntu-sleeper: localhost/apparmor-deny-write
spec:
  containers:
  - name: ubuntu-sleeper
    image: ubuntu
    command: [ "sh", "-c", "echo 'Sleeping for an hour!' && sleep 1h" ]
```

3.16) Linux Capabilities

Il est important de savoir ajouter ou supprimer des capacités Linux à nos différents pods Kubernetes

Dans les anciennes versions du kernel Linux (<2.2) ; les processus Linux étaient divisés en 0é groupes :

- Privileged Process : il s'agissait des processus lancés par le root à UID=0 et ces processus pouvaient presque tout faire sur le système en by-passant les vérifications ou restrictions du Kernel
- Unprivileged Process : Processus exécuté par tout autre utilisateur différent du root, donc UID différent de 0, et ces processus avaient de nombreuses restrictions imposées par le kernel

< Kernel 2.2

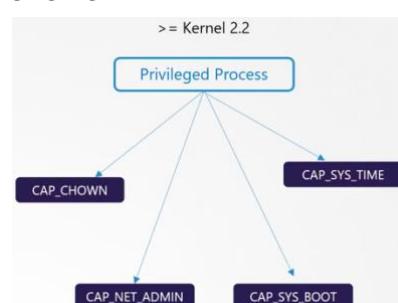
Privileged Process

Unprivileged Process

Dans les versions supérieures à la version du kernel 2.2, les super priviléges de l'utilisateur root ont également été divisées en plusieurs unités appelés CAPABILITIES. Et de ce fait au lieu d'utiliser l'utilisateur root pour faire tout cela, on peut juste assigner à certains processus privilégiés un certain nombre de CAPABILITIES

Le concept de Linux capabilities

- `CAP_CHOWN` : Capacité permettant de modifier le propriétaire d'un fichier
- `CAP_NET_ADMIN` : Capacité permettant de modifier le paramètres réseaux avancés
- `CAP_SYS_BOOT` : Capacité permettant de redémarrer le système d'exploitation
- `CAP_SYS_TIME` : capacité permettant de modifier le temps, date et heure du système



Il existe une douzaine de capabilities disponibles sous Linux. Pour vérifier quelles capabilities une commande Linux a besoin, on peut :

- `getcap /usr/bin/ping` : affiche les capacités nécessaires pour exécuter la commande ping
- `getpcaps pid` : affiche les capabilities nécessaires pour un processus particulier

```
▶ getcap /usr/bin/ping
```

```
/usr/bin/ping = cap_net_raw+ep
```

```
▶ ps -ef | grep /usr/sbin/sshd | grep -v grep
```

```
root 779 1 0 03:55 ? 00:00:00 /usr/sbin/sshd -D
```

```
▶ getpcaps 779
```

```
capabilities for `779': =
cap_chown, cap_dac_override, cap_dac_read_search, cap_fowner, cap_fsetid, cap_kill,
, cap_setuid, cap_setpcap, cap_linux_immutable, cap_net_bind_service, cap_net_broad
```

Par défaut, un container est lancé avec un nombre de capabilities très limité, le profile Seccomp par défaut de Docker lance les containers avec juste 14 capabilities. C'est la raison pour laquelle même en désactivant complètement le Seccomp autorisant ainsi tous les Syscalls, qu'on ne puisse toujours pas réaliser certaines commandes dans les containers comme changer la date ou redémarrer la machine . eh bien c'est tout simplement parce que par défaut Docker lance les containers sans ces capabilities là.

Pour ajouter ou supprimer des capabilities à un pod Kubernetes, on utilise les manifest de la façon suivante :

`ubuntu-sleeper.yaml`

```
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper
spec:
  containers:
  - name: ubuntu-sleeper
    image: ubuntu
    command: [ "sleep", "1000" ]
    securityContext:
      capabilities:
        add: ["SYS_TIME"]
        drop: ["CHOWN"]
```

`kubectl exec -ti ubuntu-sleeper -- bash`

```
root@ubuntu-sleeper:/# date
Sat Apr 3 05:32:06 UTC 2021
root@ubuntu-sleeper:/# date -s '19 APR 2012 22:00:00'
Thu Apr 19 22:00:00 UTC 2012
root@ubuntu-sleeper:/# date
Thu Apr 19 22:00:02 UTC 2012
```

`kubectl exec -ti ubuntu-sleeper -- bash`

```
root@ubuntu-sleeper:~# touch /tmp/test
root@ubuntu-sleeper:~# ls -l /tmp/test
-rw-r--r-- 1 root root 0 Apr 3 05:46 /tmp/test
root@ubuntu-sleeper:~# chown backup /tmp/test
chown: changing ownership of '/tmp/test': Operation
not permitted
```

IV) Minimize Microservice Vulnerabilities

4.1) Security Contexts

Comme nous l'avons vu précédemment, lorsqu'on lance un container à l'aide de Docker, il existe un certain nombre d'option de sécurité que l'on peut définir au lancement tel que (un user particulier à utiliser ou une capability que l'on souhaite donner au container). Tout cela est également possible sur Kubernetes au lancement d'un Pod à l'aide des Security Contexts. Si le security Context est défini au niveau du Pod, alors le paramètre de sécurité devra s'appliquer à tous les containers dudit pod ; par contre si il est défini à l'intérieur d'un container, il ne s'appliquera qu'à ce container là. et si la configuration est faite à la fois au niveau du Pod et au niveau du container, alors celle du container a la priorité.

Note: Capabilities are only supported at the container level and not at the POD level

```
docker run --user=1001 ubuntu sleep 3600
docker run --cap-add MAC_ADMIN ubuntu
apiVersion: v1
kind: Pod
metadata:
  name: web-pod
spec:
  containers:
  - name: ubuntu
    image: ubuntu
    command: ["sleep", "3600"]
    securityContext:
      runAsUser: 1000
      capabilities:
        add: ["MAC_ADMIN"]
```

4.2) Admission Controllers

Lorsque nous utilisons l'utilitaire Kubectl pour envoyer une commande à un cluster Kubernetes, cette commande arrive au niveau de l'ApiServer et suit une succession d'étapes avant de s'exécuter qui sont :



- **L'authentification** : qui se fait généralement à l'aide d'échange de certificat en ApiServer et Kubectl (à travers le kubeconfig file).
- **L'autorisation** : Après avoir confirmé que vous avez le droit d'accéder au cluster, l'ApiServer vérifie ensuite si vous avez les autorisations nécessaires pour exécuter la commande que vous tentez (cela est fait à l'aide des ressources rôles et rôles binding RBAC liées à l'utilisateur).
- **Admission Controllers** : Après avoir validé vos autorisations, Apiserver vérifie auprès de l'Admission Controller s'il n'existe pas de règles de sécurité dans le cluster empêchant d'exécuter votre commande (par exemple cela pourrait être une règle empêchant de lancer un pod avec une image provenant d'un registre spécifié ; de lancer un pod avec un utilisateur autre que celui défini, d'ajouter un certain nombre de Capabilities au Pod...)

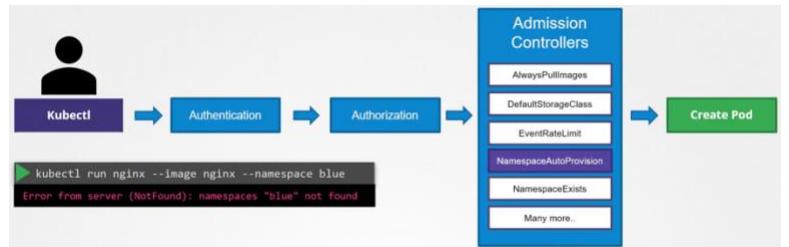
```
web-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: web-pod
spec:
  containers:
  - name: ubuntu
    image: ubuntu:latest
    command: ["sleep", "3600"]
    securityContext:
      runAsUser: 0
      capabilities:
        add: ["MAC_ADMIN"]
```

L'Admission Controller permet de mettre en place un certain nombre de règles de sécurité additionnelles afin d'imposer comment un cluster doit fonctionner ou être utilisé. Ces règles peuvent être mise en place dans le but de renforcer la sécurité du cluster ou alors de respecter une certaine politique de gouvernance de l'entreprise. En plus de mettre en place ces règles de contrôle, l'Admission Contrôleur dispose de plusieurs autres fonctionnalités comme modifier la requête envoyée par le kubectl fonction de certaines conditions ou alors créer d'autres ressources ou modifier certains composants du cluster avant d'exécuter la requête.

Par défaut sur Kubernetes, il y'a un certain nombre d'Admission Controllers qui existent , par exemple

- **AllwaysPullImages** : qui se rassure lors de la création d'une image de la télécharger depuis son registre si jamais elle n'existe pas
- **DefaultStorageClass** : qui permet d'attribuer une storageClass par défaut à tout PV ou PVC crée sans StorageClass
- **EventRateLimit** : qui permet de définir un nombre maximum de requête pouvant être traité en simultané par l'Apiserver afin d'éviter les surcharges.
- **NamespaceExists** : qui vérifie que le namespace existe bien avant de créer une ressource dans ledit namespace, dans le cas contraire il rejette la requête si le namespace n'existe pas.

Il existe également plusieurs admission Controllers Kubernetes qui ne sont pas activés par défaut, mais peuvent être activé par un administrateur(exemple l'admission contrôleur **NamespaceAutoProvision** qui permet de créer automatiquement un namespace si jamais il n'existe pas et qu'on tente d'y créer un pod, dans ce cas la requête ne sera pas rejetée comme en fonctionnement par défaut, mais le namespace sera d'abord créé avant la création du Pod)



Pour avoir la liste des admissions controllers activés par défaut dans un cluster kubernetes, il faut taper la commande **kube-apiserver -h | grep enable-admission-plugins**

```

▶ kube-apiserver -h | grep enable-admission-plugins
--enable-admission-plugins strings      admission plugins that should be enabled in addition to default enabled ones
(NamespaceLifecycle, LimitRanger, ServiceAccount, TaintNodesByCondition, Priority, DefaultTolerationSeconds,
DefaultStorageClass, StorageObjectInUseProtection, PersistentVolumeClaimResize, RuntimeClass, CertificateApproval,
CertificateSigning, CertificateSubjectRestriction, DefaultIngressClass, MutatingAdmissionWebhook,
ValidatingAdmissionWebhook, ResourceQuota). Comma-delimited list of admission plugins: AlwaysAdmit, AlwaysDeny,
AlwaysPullImages, CertificateApproval, CertificateSigning, CertificateSubjectRestriction, DefaultIngressClass,

```

Dans le cas où on se trouve dans un cluster kubernetes déployé à l'aide de kubeadm, alors cette commande doit être saisi à l'intérieur du pod apiserver du nœud master à l'aide de kubectl exec

```

▶ kubectl exec kube-apiserver-controlplane -n kube-system -- kube-apiserver -h | grep enable-admission-plugins
ps -ef | grep kube-apiserver | grep admission-plugins (affiche la liste activés manuellement)

```

Pour activer un admission controller, il suffit de modifier le fichier de création de l'apiserver (le manifest **kube-apiserver.yaml** si cluster déployé à l'aide de kubeadm ; ou alors le service **kube-apiserver.service** si jamais le cluster a été déployé from scratch) en ajoutant l'admission controller que vous souhaitez activer dans la liste des admissions controller à activer au niveau de l'option **--enable-admission-plugins**. réciproquement, pour désactiver un admission controller, on peut également utiliser l'option **--disable-admission-plugins**.

<code>kube-apiserver.service</code>	<code>/etc/kubernetes/manifests/kube-apiserver.yaml</code>
<pre> ExecStart=/usr/local/bin/kube-apiserver \ --advertise-address=\${INTERNAL_IP} \ --allow-privileged=true \ --apiserver-count=3 \ --authorization-mode=Node,RBAC \ --bind-address=0.0.0.0 \ --enable-swagger-ui=true \ --etcd-servers=https://127.0.0.1:2379 \ --event-ttl=1h \ --runtime-config=api/all \ --service-cluster-ip-range=10.32.0.0/24 \ --service-node-port-range=30000-32767 \ --v=2 --enable-admission-plugins=NodeRestriction,NamespaceAutoProvision --disable-admission-plugins=DefaultStorageClass </pre>	<pre> apiVersion: v1 kind: Pod metadata: creationTimestamp: null name: kube-apiserver namespace: kube-system spec: containers: - command: - kube-apiserver - --authorization-mode=Node,RBAC - --advertise-address=172.17.0.107 - --allow-privileged=true - --enable-bootstrapping-token-auth=true - --enable-admission-plugins=NodeRestriction,NamespaceAutoProvision image: k8s.gcr.io/kube-apiserver-amd64:v1.11.3 name: kube-apiserver </pre>

4.3) Validating and Mutating Admission Controllers

Il existe deux type d'admission controllers

- **Les validating Admission controllers** : qui ont pour rôle de checker et valider une configuration afin de décider si elle est autorisée ou pas.

- **Les Mutating Admission controllers** : Qui quant à eux en plus du check et ou de la validation, apportent des modifications sur la ressources en train d'être crée (par exemple le **NamespaceAutoProvision**, va alors modifier un ressource du cluster en créant le namespace manquant avant de créer ladite ressource ou le **DefaultStorageClass** qui quant à lui va modifier tout PVC n'ayant pas de storageClass définie).

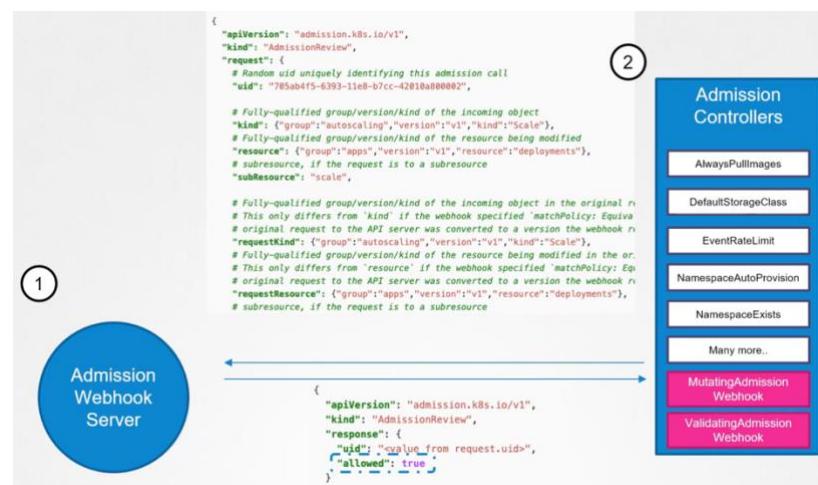
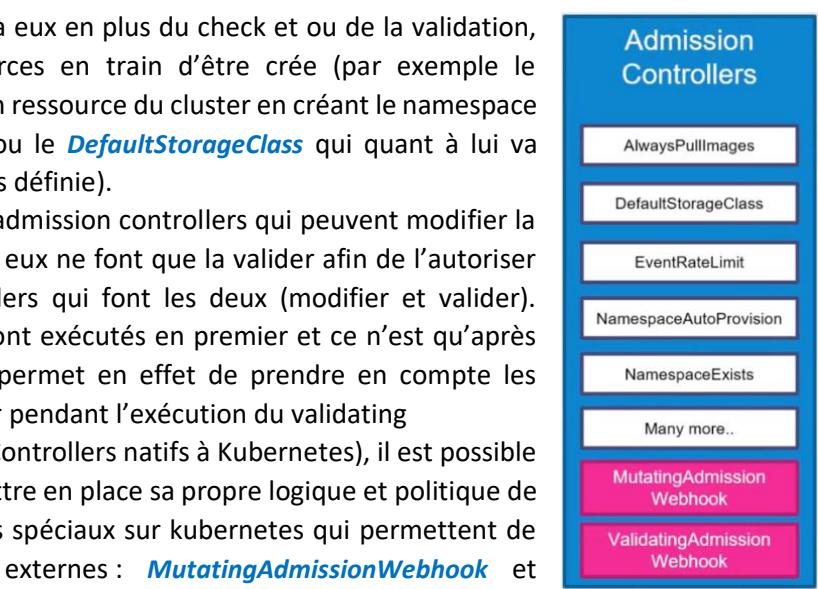
Les Mutating Admission controller sont en effet des admission controllers qui peuvent modifier la requête tandis que les Validating controllers quant à eux ne font que la valider afin de l'autoriser ou non. Il existe également des Admission controllers qui font les deux (modifier et valider). Généralement, les Mutating Admission controllers sont exécutés en premier et ce n'est qu'après que les Validating sont exécutés à leur tour. Cela permet en effet de prendre en compte les changement fait par le Mutating Admission controller pendant l'exécution du validating. En plus des **Built-In** Admission controller (Admission Controllers natifs à Kubernetes), il est possible de créer ses propres Admission Controller afin de mettre en place sa propre logique et politique de sécurité. Pour cela il existe 02 admissions controllers spéciaux sur kubernetes qui permettent de prendre en charge des admission controllers externes : **MutatingAdmissionWebhook** et **ValidatingAdmissionWebhook**

Il faudrait alors par la suite configurer ces Admission controllers spéciaux afin qu'ils pointent sur server d'Admission controller configuré ou crée avec notre propre logique et propre code. Ce serveur d'Admission controllers fonctionne alors comme une API qui reçoit en entrée des requêtes sous format JSON de Kubernetes et en sortie renvoie également une donnée en JSON dans laquelle la réponse d'autorisation ou de refus de la requête est mentionnée. A chaque fois qu'une requête devant être traitée par l'admission controller externe va arriver sur l'Admission controller Kubernetes, ce dernier va envoyer un objet JSON (request review)

contenant le résumé et tous les détails sur la requête au serveur d'Admission Controller afin que la requête soit traitée fonction de la logique définie dans ce serveur. Une fois la requête traitée, le serveur externe d'Admission Controller va à son tour envoyer un objet JSON à Kubernetes contenant la réponse et disant si la requête est approuvée ou refusée.

Le déploiement du webhook server peut se faire sur n'importe quelle plateforme et avec le langage de votre choix (python, go, java, c ...) et peut être déployé comme étant un pod kubernetes exposé par un service ou alors peut être déployé carrément à l'extérieur du cluster et dans ce cas accessible au travers de son URL. Lors de la création de son propre serveur webhook, la seule contrainte imposée par Kubernetes est qu'il doit pouvoir accepter le Mutating et validating API venant de kubernetes et répondre avec un object JSON répondant aux attentes de kubernetes.

Ci-après nous avons un exemple de code d'un serveur Api permettant de traiter les Admission controller externes. On constate que l'API doit avoir obligatoirement les deux méthodes **/validate** et **/mutate** qui devront être appelées par kubernetes afin de lui fournir la request review et ces méthodes une fois la request review reçue, doivent la traiter et renvoyer à leur tour la réponse



```

message = "You can't create objects with your own name"
status = False
return jsonify(
{
  "response": {
    "allowed": status,
    "uid": request.json["request"]["uid"],
    "status": {"message": message},
  }
}

@app.route("/mutate", methods=["POST"])
def mutate():
  user_name = request.json["request"]["userInfo"]["name"]
  patch = [{"op": "add", "path": "/metadata/labels/users", "value": user_name}]
  return jsonify(
  {
    "response": {
      "allowed": True,
      "uid": request.json["request"]["uid"],
      "patch": base64.b64encode(patch),
      "patchType": "JSONPatch",
    }
  }
)

```

d'approbation ou de refus de la requête fonction de la logique définie dans lesdites méthodes. Dans cet exemple, on récupère le nom de l'objet à créer ainsi que le nom du user qui souhaite créer la requête et dans le cas où le nom de l'objet est égal au nom du user, la requête est rejetée.

Après avoir configuré et déployé l'Admission webhook server, il faut par la suite configurer l'Admission webhook dans kubernetes afin que les requêtes soient envoyées vers l'Admission webhook serveur pour validation. Pour cela il faut créer une ressource kubernetes de type [ValidatingWebhookConfiguration](#) or [MutatingWebhookConfiguration](#)

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  name: "pod-policy.example.com"
webhooks:
- name: "pod-policy.example.com"
  clientConfig:
    service:
      namespace: "webhook-namespace"
      name: "webhook-service"
      caBundle: "C10tLS0tQk.....tLS0K"
  rules:
  - apiGroups: [""]
    apiVersions: ["v1"]
    operations: ["CREATE"]
    resources: ["pods"]
    scope: "Namespaced"
```

Dans le cas où le webhook serveur est déployé hors du cluster, alors on le référence à l'aide de son URL et si jamais il déployé dans le cluster en tant que Pod et exposé au travers d'un service, in le référence à l'aide du pod.

Par la suite dans la configuration du webhook Admission controller, on doit spécifier les règles qui permettront de décider quelles seront les requêtes reçues par l'apiserver à envoyer pour validation vers l'Admission webhook serveur. Dans cet exemple ci-après, cette configuration ne sera appellée que lors des requêtes de création de pods dans le cluster (liées à l'apiVersion :v1, la ressource de type pod et l'opération de création)

La communication entre L'apiserver de kubernetes et le serveur de webhook doivent être cryptées, raison pour laquelle il faut également préciser dans cette configuration le certificat et l'autorité de certificat.

4.4) Pod Security Policies

Ce sont des ressources Kubernetes qui permettent de définir une politique de sécurité autour de la création de pods ; en effet les règles définies sont systématiquement vérifiées à la création de tout nouveau pod et en cas de non-respect des règles, la requête de création est rejetée par le cluster. Les Pod Security Policies sont en effet liés aux Admission Controller car pour configurer un PSP, il faut réunir les prérequis suivants :

- 1) Activer l'admission Controller PodSecurityPolicy grâce à l'option « --enable-admission-plugins » du fichier de configuration kubernetes
- 2) Créer les ressources Pod Security Policies à l'aide de manifest kubernetes
- 3) Configurer les autorisations au niveau des Pod afin que tout nouveau pod à créer puisse accéder et utiliser les ressources PodSecurityPolicies existantes afin de vérifier si leur création est permise.

Comme vu précédemment, l'activation de l'admission controller PodSecurityPolicy se fait via le fichier de configuration kubernetes en ajoutant cet admission controller au niveau de l'option « --enable-admission-plugins »

```
--enable-admission-plugins=PodSecurityPolicy
- --enable-bootstrap-token-auth=true
- --enable-admission-plugins=PodSecurityPolicy
image: k8s.gcr.io/kube-apiserver-amd64:v1.11.3
name: kube-apiserver
```

```
psp.yaml
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: example-psp
spec:
  privileged: false
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
```

Pour ce qui est de la création de la ressource PSP dans kubernetes, cela se fait à l'aide du manifest ci-après :

Ici après avoir défini l'ApiVersion, le type de ressource et le nom de la ressource, on entre dans le bloc spec dans lequel on spécifie les règles que les nouveaux pods devront respecter

Dans cet exemple, tous les pods qui auront le paramètres « **Privileged : true** » seront rejettés, donc pour tous les pod, la configuration des paramètres renseignés dans cette PSP devra être la même sinon la création sera refusée Pour la valeur **RunAsAny**, cela signifie que les pod peuvent avoir n'importe quelle valeur pour ces attributs là (**seLinux**, du **fsGroup** ou du **runAsUser...**)

Quant aux autorisations nécessaires aux pods afin de pouvoir utiliser les PSP, il s'agit d'un prérequis important car si jamais l'admission controller PodSecurityPolicy est activé et que les POD n'arrivent pas à accéder aux PSP alors toutes les requêtes de création de nouveau pod seraient rejetés par l'Admission controller car ce dernier n'aura accès à aucune PSP afin de checker.

Pour donner ces accès aux pod, on passe par le security account par défaut attribué à chaque pod lors de sa création afin de lui donner les autorisation nécessaires à l'aide des roles et rolebinding

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-pod
spec:
  serviceAccount: default
  containers:
    - name: ubuntu
      image: ubuntu
      command: ["sleep", "3600"]
      securityContext:
        privileged: True
        runAsUser: 0
        capabilities:
          add: ["CAP_SYS_BOOT"]
  volumes:
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: psp-example-role
rules:
  - apiGroups: ["policy"]
    resources: ["podsecuritypolicies"]
    resourceNames: ["example-psp"]
    verbs: ["use"]
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: psp-example-rolebinding
subjects:
  - kind: ServiceAccount
    name: default
    namespace: default
roleRef:
  kind: Role
  name: psp-example-role
  apiGroup: rbac.authorization.k8s.io
```

Il existe plusieurs règles possibles lors de la configuration ou création d'un PSP, ces règles peuvent être accessible via la documentation officielle de kubernetes sur les POD SECURITY POLICIES

Il est également à noter que l'Admission controller PodSecurityPolicy est à la fois du type validating et Mutating, car en plus de valider les requêtes, il peut également modifier certains paramètres des pods avant leur création grâce à certaines options dans les règles des PSP comme « defaultAddCapabilities » qui modifiera systématiquement le pod afin d'ajouter la capability par défaut définie.

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: example-psp
spec:
  privileged: false
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: 'MustRunAsNonRoot'
  requiredDropCapabilities:
  - 'CAP_SYS_BOOT'
  defaultAddCapabilities:
  - 'CAP_SYS_TIME'
  volumes:
  - 'persistentVolumeClaim'
```

4.5) Open Policy Agent (OPA)

OPA permet d'ajouter une couche de sécurité applicative sur nos différentes applications. En effet dans une application la sécurité se positionne à plusieurs niveaux et peut être configurés de diverses manières. Pour accéder à une application, on passe par la phase d'authentification (user+ mdp, certificats, token...) ensuite après s'être authentifié, l'application doit normalement controller nos autorisations afin de déterminer quel champ d'action on doit avoir dans l'application. Cette configuration des autorisations peut également se faire de diverses manières (soit directement dans le code de l'application ou à l'aide de nombreux outils parmi lesquels OPA). Pour résumer OPA est un outils qui permet d'ajouter une couche de sécurité sur une application afin de limiter les autorisations d'un utilisateur défini sur l'application. L'avantage de OPA est qu'il peut permettre de définir des règles de sécurité niveau autorisations pouvant s'appliquer à plusieurs applications en simultané et cela peu importe le langage dans lequel l'application a été développée.

Pour mettre en place OPA, il faut installer et démarrer OPA Server dans votre système

- 1) Télécharger le binaire de OPA et le rendre exécutable

```
curl -L -o opa https://github.com/open-policy-agent/opa/releases/download/v0.11.0/opa_linux_amd64
chmod 755 ./opa
```

- 2) Lancer le serveur OPA en exécutant son binaire. Une fois cela fait, OPA est disponible sur le port 8181. Par défaut les authentifications et autorisations sont désactivées sur OPA

```
./opa run -s
{"addrs":[":8181"],"insecure_addr":"","level":"info","msg":"First line of log stream.", "time":"2021-03-18T20:25:38+08:00"}
```

- 3) Une fois le serveur OPA démarré, il faut charger les Policies (règles ou stratégies) OPA dessus, ces Policies se chargeront de valider ou refuser les requêtes.

```
example.rego
package httpapi.authz

# HTTP API request
import input

default allow = false

allow {
  input.path == "home"
  input.user == "john"
}
```

Les policies OPA sont créés à travers un langage qui s'appelle le rego, donc ce sont des fichiers à l'extension *.rego

Dans cette exemple ci-après, on utilise l'api « `httpapi.authz` » et au niveau des règles d'autorisations, il est configuré dans ce cas que par défaut toutes les requêtes sont refusées et on observe au niveau du bloc de validation ou d'autorisation des requêtes que seules les requêtes provenant de l'utilisateur « `john` » et avec le path « `/home` » devront être acceptées.

Toutes les lignes du bloc de validation des autorisations doivent être des conditions et pour autoriser une requête, toutes lesdites lignes doivent être vraies.

- 4) Une fois la Policy créée, elle doit être chargée dans le serveur OPA en utilisant l'API du serveur OPA pour lui envoyer une requête PUT

```
curl -X PUT --data-binary @example.rego http://localhost:8181/v1/policies/example1
```

Pour afficher la liste de Policies existantes sur le serveur OPA, on envoie également une requête sur le serveur OPA

```
curl http://localhost:8181/v1/policies
```

- 5) Pour terminer les configurations, il faut ajouter dans le code de l'application, le bloc de validation des autorisations passant par le serveur OPA

Références Links

<https://www.youtube.com/watch?v=R6tUNpRpdnY>

<https://www.youtube.com/watch?v=4mBJSIhs2xQ>

Téléchargement, installation et lancement serveur OPA

```
curl -Lo opa https://github.com/open-policy-agent/opa/releases/download/v0.38.1/opa_Linux_amd64
chmod +x opa
./opa run -s
```

#Chargement d'un Policy sur le serveur OPA

```
curl -X PUT --data-binary @/root/sample.rego http://localhost:8181/v1/policies/samplepolicy
```

#Vérification des Policies présentes sur le serveur OPA

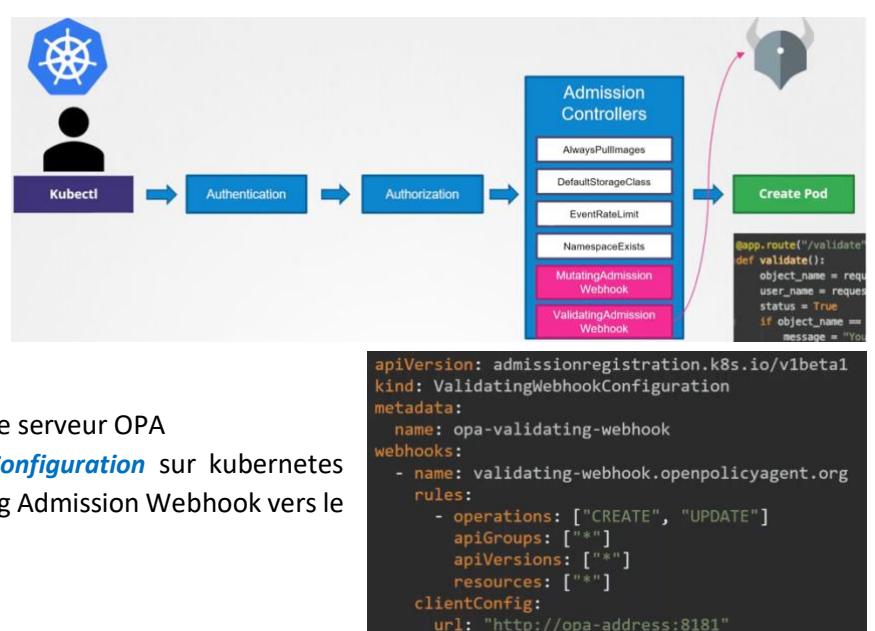
```
curl http://localhost:8181/v1/policies
```

4.6) OPA in Kubernetes

La mise en place du serveur OPA sur Kubernetes permet en effet de remplacer le Serveur Webhook externe par le serveur OPA. Donc les Admission webhook spéciaux de kubernetes devront renvoyer les requêtes en JSON au serveur OPA et ce dernier devra les traiter et renvoyer le résultat également en JSON

Les étapes d'intégration du serveur OPA à kubernetes sont les suivantes :

- 1) Mettre sur pied le serveur OPA
- 2) Créer les OPA Policies et les charger sur le serveur OPA
- 3) Créer la ressource `ValidatingWebhookConfiguration` sur kubernetes afin de rediriger les requêtes du validating Admission Webhook vers le serveur OPA



Il est également possible de déployer le serveur OPA directement dans le cluster kubernetes à l'aide d'un Pod exposé par un service te modifier donc le fichier de configuration kubernetes de création de la ressource ValidatingWebhookConfiguration afin de faire le lien vers le service et penser également à y ajouter le certificat pour sécurisation de la communication

Exemple de OPA Policy pour Kubernetes, qui vérifie la source de l'image avant création du POD

```
kubernetes.rego
package kubernetes.admission

deny[msg] {
    input.request.kind.kind == "Pod"
    image := input.request.object.spec.containers[...].image
    startswith(image, "hooli.com/")
    msg := sprintf("Image '%v' from untrusted registry", [image])
}
```

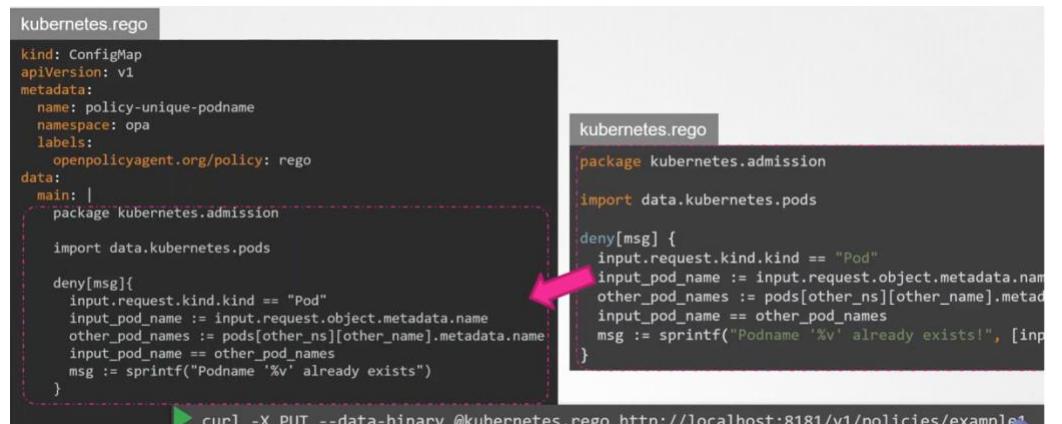
```
apiVersion: admissionregistration.k8s.io/v1beta1
kind: ValidatingWebhookConfiguration
metadata:
  name: opa-validating-webhook
webhooks:
- name: validating-webhook.openpolicyagent.org
  rules:
    - operations: ["CREATE", "UPDATE"]
      apiGroups: ["*"]
      apiVersions: ["*"]
      resources: ["*"]
  clientConfig:
    caBundle: $(cat ca.crt | base64 | tr -d '\n')
service:
  namespace: opa
  name: opa
```

Il est parfois nécessaire dans certaines OPA Policies, d'avoir les informations sur certaines ressources existantes dans le cluster, par exemple pour avoir un règle qui vérifie qu'aucun Pod avec le même nom n'existe dans tout le cluster avant de valider la requête de création. Pour cela, il faudrait bien sur avoir les informations sur les PODS du cluster, mais malheureusement l'admission review envoyé au serveur OPA n'envoie que les informations concernant le POD à créer et non les informations sur tous les Pods. Pour mettre cela en place il faut :

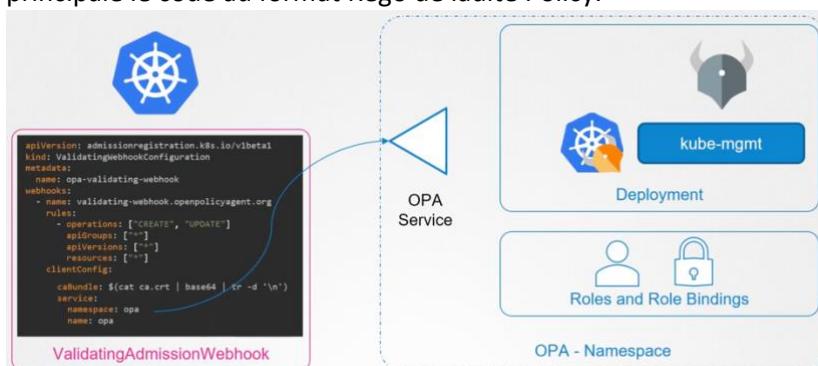
- 1) Utiliser le container kube-mgmt : qui doit être déployé en tant que 2^{ème} container du POD du serveur OPA et qui permettra de réplica la définition des ressources de Kubernetes dans le POD du serveur OPA.
- 2) Après avoir déployé et configuré kube-mgmt, il faut modifier la Policy OPA afin d'y ajouter l'instruction d'import des données souhaitées de kubernetes



En plus de permettre au serveur OPA de récupérer les informations sur les autres ressources du cluster, **Kube-mgmt** permet également de charger automatiquement les OPA Policies sur le serveur OPA ; plus besoin de le faire manuellement à l'aide de « curl -X PUT ». pour rendre le chargement des OPA Policies automatique sur le serveur OPA



en passant par **Kube-mgmt**, il suffit juste de créer un configMap sur kubernetes qui devra être configuré d'une façon particulière afin que Kube-mgmt le récupère afin de charger la Policy automatiquement sur le serveur OPA. Le configMap doit obligatoirement avoir le label « openpolicyagent.org/policy : rego et doit contenir comme donnée principale le code au format Rego de ladite Policy.



Lors du déploiement de OPA dans le cluster Kubernetes, il faut penser à créer les rôles et RolesBinding nécessaires afin que Kube-mgmt puisse avoir les accès au cluster afin de récupérer les informations nécessaires sur les ressources du cluster ou alors sur les configMap contenant les OPA policies .

4.7) Manage Kubernetes Secrets

Il existe deux façons de créer un secret dans kubernetes , de façon impérative (ligne de commande) ou déclarative (manifest)

```

Imperative
▶ kubectl create secret generic <secret-name> --from-literal=<key>=<value>

▶ kubectl create secret generic \
    app-secret --from-literal=DB_Host=mysql \
    --from-literal=DB_User=root \
    --from-literal=DB_Password=paswrd

▶ kubectl create secret generic <secret-name> --from-file=<path-to-file>

▶ kubectl create secret generic \
    app-secret --from-file=app_secret.properties

Declarative
▶ kubectl create -f secret-data.yaml

secret-data.yaml
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
data:
  DB_Host: mysql
  DB_User: root
  DB_Password: paswrd

▶ kubectl create -f secret-data.yaml

```

Lors de la création du secret en mode déclaratif, les valeurs doivent être cryptées en base 64.

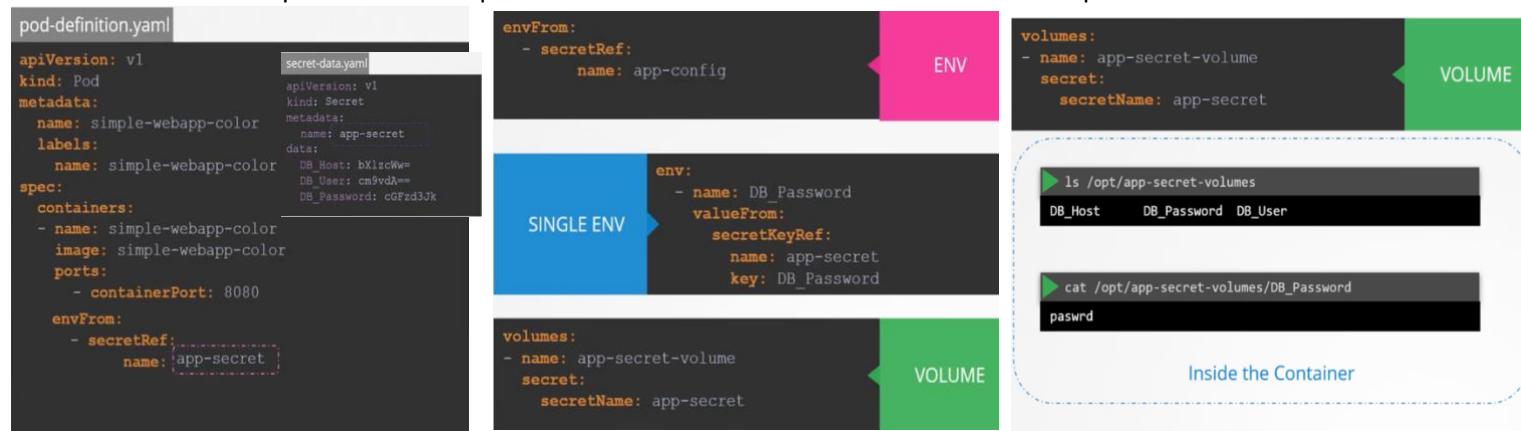
```

▶ echo -n 'paswrd' | base64
cGFzd3Jk

▶ echo -n 'cGFzd3Jk' | base64 --decode
paswrd

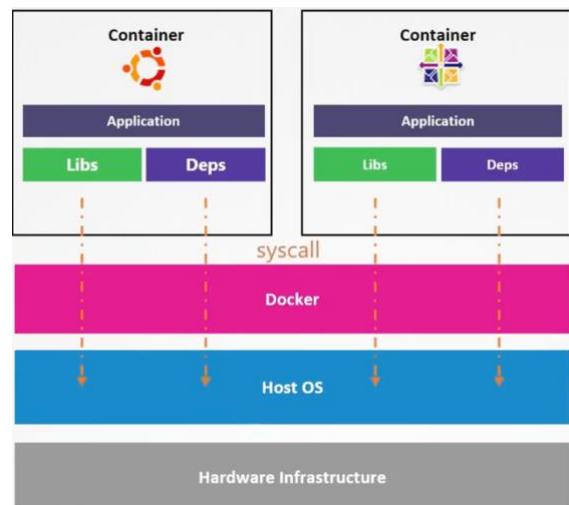
```

Une fois le secret créé, il peut être injecté au pod soit en utilisant des variables d'environnement, ou alors en utilisant des volumes. Lors de l'utilisation des volumes, kubernetes va créer autant de fichiers que de données secrètes dans le volume et dans chaque fichier se trouvera la valeur du secret. Dans l'exemple ci-dessous on remarque qu'il existe 03 fichiers dans le répertoire dans lequel a été monté le volume et ces 03 secrets correspondent aux données secrètes



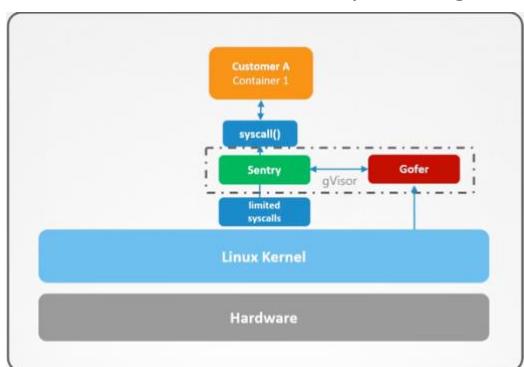
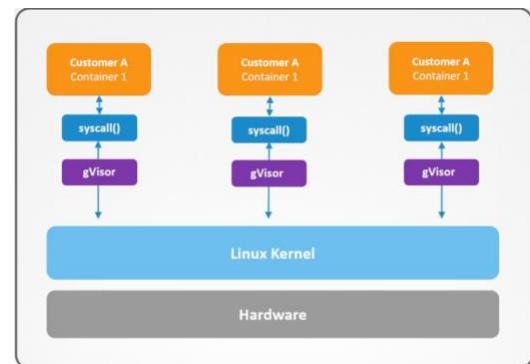
4.8) Container Sandboxing

La virtualisation permet d'avoir des machines virtuelles qui ont chacune un Kernel dédié et de ce fait sont parfaitement isolées les unes des autres car les processus de chaque machine fonctionnent envoient des Syscalls à leur propre kernel. Contrairement en environnement de container, où les containers d'un même hôte partagent tous le même Kernel qui est celui de l'hôte. Le Sandboxing est en effet une méthode de sécurité qu'on met en place pour isoler au maximum les containers les uns des autres. Nous avons déjà parlé plus haut de certaines techniques de Sandboxing comme le Seccomp qui permet de se rassurer que certains containers ne puissent pas effectuer des Syscalls pouvant mettre en péril les autres containers ou l'hôte. C'est également le cas avec la méthode APPARMOR qui permet également de réduire la surface d'attaque dans un container en limitant les ressources auxquelles elles peuvent avoir accès sur l'hôte. Ces méthodes permettent en effet de définir un périmètre de sécurité autour des containers afin de les isoler davantage les uns des autres et également de l'hôte ; néanmoins avec les méthodes de Seccomp et AppArmor qui fonctionnent par profils, il devient difficile de mettre en place une isolation efficace sur plusieurs containers en utilisant seulement ces méthodes-là. C'est la raison pour laquelle il est impératif de voir d'autres méthodes d'isolation de container plus ou moins facile à mettre en place et pouvant s'appliquer de façon dynamique à plusieurs containers à la fois.



4.9) gVisor

Il est important de noter que le Kernel Linux est très complexe et également très puissant et que dès qu'un processus ou container communique directement avec le kernel, cela augmente grandement la surface de sécurité. L'objectif général de sécurité concernant le sandboxing de container est d'optimiser l'isolation des différents containers entre eux mais également entre les containers et l'hôte ou le kernel. C'est la raison pour laquelle il peut être intéressant d'ajouter une autre couche applicative dans le schéma de communication d'un container avec le Kernel afin d'éviter que le container communique directement avec le kernel. gVisor nous permet de mettre en place cette nouvelle couche entre le kernel et les container ; donc les containers effectuent des Syscalls à gVisor, ce dernier les traite, les valide et les transmet au kernel pour exécution.



gVisor est un Outil Google qui permet d'avoir une couche d'isolation additive entre le container et le kernel.

Il possède en effet deux composants qui s'occupent de gérer et de limiter les Syscalls entre le container et le Kernel

- **Sentry** : qui en effet un mini kernel applicatif indépendant et dédié aux containers, qui intercepte et répond aux Syscalls des containers. Étant conçu pour les containers, il possède très peu de fonctionnalité comparé au kernel Linux, cela réduit également les possibilités des containers.

- **Gofer** : c'est un serveur proxy de fichiers qui implémente la logique nécessaires aux containers pour accéder aux systèmes de fichiers.

NB : Il est important que sur un hôte, pour mettre en place une isolation optimale, il faut un gVisor kernel pour chaque container et non un seul gVisor par hôte pour tous les containers de cet hôte.

Les inconvénients à prendre en compte pour gVisor :

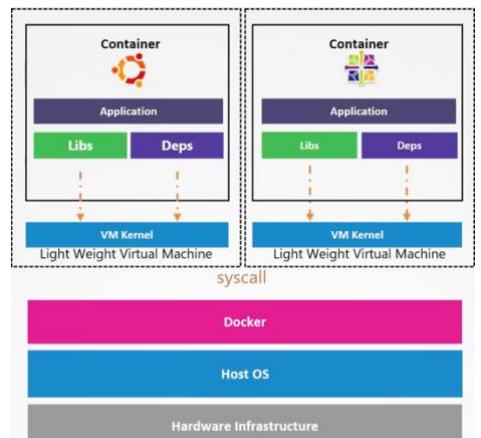
- Il n'est pas compatible avec tous les containers, donc il faudrait vérifier s'il fonctionne correctement
- Il ajoute du traitement donc ralentit les traitements dans le container comparativement à l'architecture traditionnelle.

4.10) Kata Containers

Kata Containers est également une méthode de sécurité d'isolation de container.

Elle utilise une approche différente de celle de gVisor.

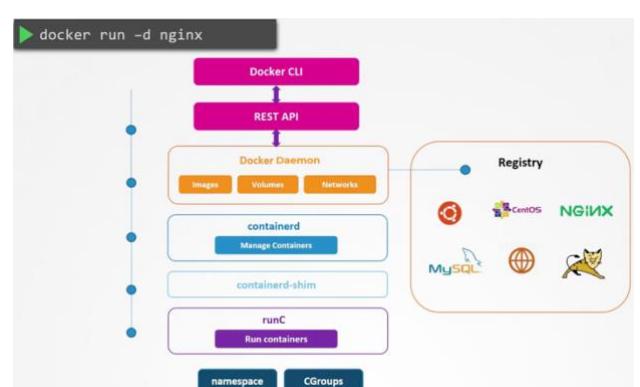
En effet Kata Container est un outil qui va créer à chaque lancement de container une micro machine virtuelle avec son propre kernel dédié dans lequel sera lancé le container, ce qui aura pour effet d'isoler les container entre eux et de séparer également les kernel des container et celui de l'hôte. Kata container est également gourmand en ressources car nécessite un peu plus de CPU et de RAM afin de créer la petite machine virtuelle et étant donné qu'il fonctionne comme un environnement virtuel, l'hôte doit au préalable supporter la technologie de virtualisation, ce qui n'est pas le cas des hôtes fournis par la plupart des cloud providers.



4.11) Runtime Classes

Pour cette partie, il est important de comprendre le workflow de lancement d'un container. Ce workflow suit les étapes suivantes :

- 1) Lorsqu'on exécute une commande sur la CLI Docker, cette dernière convertit en commande en requête API et l'envoie au démon Docker
- 2) Dès réception de la commande, le démon docker commence par vérifier si l'image existe en local, sinon l'image est téléchargée depuis le registre concerné



- 3) Une fois l'image téléchargée, le démon docker fait un appel au composant containerd afin qu'il lance le container. Containerd est responsable de convertir l'image en package compatible OCI (Open Container Initiative), ensuite ce package est passé au composant containerd-shim
- 4) Containerd-shim à son tour une fois le package OCI reçu, fait appel au container Runtime runC afin qu'il lance le container
- 5) runC interagit avec le nouveau namespace et CGroup du Kernel afin de créer le container.

runC est en effet le composant par défaut de docker qui a la charge d'effectivement créer le container, c'est également l'un des container runtime par défaut qui est compatible aux normes OCI (Open Container Initiative) qui est en réalité l'initiative qui spécifie les standards autour du format des containers et de leur exécution.

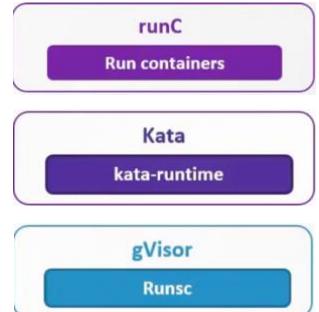
Avec runC directement installé sur une machine, il est possible de lancer directement un container en utilisant la CLI de runC « runc run nginx » mais il serait difficile de manager le cycle de vie de ce container sans les fonctionnalités additionnelles de Docker.

runC est le container runtime par défaut utilisé par docker, Podman, CRI-O mais il existe d'autres container runtime notamment le **kata-runtime** utilisé par kata container (vu précédemment et permettent de lancer des mini VM afin d'isoler les container) et **Runsc** qui est le container runtime utilisé par gVisor.

Étant donné que ces autres container runtime sont également compatibles OCI, on peut les utiliser via la CLI de docker afin de lancer les container : `docker run --runtime kata -d nginx`

```
▶ docker run --runtime kata -d nginx
6d9994bf88bc3538f23db7aa4b01133e3d58dcdbc0e0d5d1a44c31a5ae06fad0
```

```
▶ docker run --runtime runsc -d nginx
dg2dc994289338f23db7aa4b01133e3d58dcdbc0e0d5d1a44c31a5ae06fadabc
```



4.12) Runtime Class in Kubernetes

En effet gVisor est installé par défaut dans un cluster kubernetes et pour utiliser spécifiquement le runtime de gVisor lors de la création des container d'un pod il faut :

- 1) Créer un ressource kubernetes appelée RuntimeClass qui correspondra en effet au runtime désiré
- 2) Modifier le manifest de création du Pod en ajoutant à ses specs le paramètre runtimeClassName

```
gvisor.yaml
apiVersion: node.k8s.io/v1beta1
kind: RuntimeClass
metadata:
  name: gvisor
handler: runsc
```

```
▶ kubectl create -f gvisor.yaml
runtimeclass.node.k8s.io/gvisor created
```

```
gvisor-nginx.yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: nginx
    name: nginx
spec:
  runtimeClassName: gvisor
  containers:
  - image: nginx
    name: nginx
```

Le Handler correspond ici au runtime à utiliser (runsc pour gVisor et kata pour kata-container)

Runtime	Handlers
gVisor	runsc
Kata	kata

Afin de tester la réelle isolation du container lancé à l'aide du runtime

gVisor, on peut tout simplement se rassurer que les processus exécutés dans le container ne soient plus visibles sur l'hôte. On peut également observer que le processus de **runsc** s'exécute bien sur le nœud hébergeant le container

```
▶ node01:~# pgrep -a nginx
```

```
▶ node01:~# pgrep -a runsc
27259 runsc --root=/run/containerd/runsc/k8s.io --
log=/run/containerd/io.containerd.runtime.v2.task/k8s.io/e6d7we978d7d6f7g98b9g8d8723424g77567
fds/log.json --log-format=json
```

4.13) Using mTLS to secure Pod to Pod communication

Il est parfois indispensable de sécuriser(encrypter) les communication entre deux pod afin d'éviter les attaques d'hacker (man in the middle). Pour cela on utilise le mutual TL plutôt que le One Way TLS (généralement utilisé par le sites web)

en effet dans le One Way TLS, seule une entité vérifié l'identité de l'autre dans le cas des sites web, c'est généralement le client qui vérifie l'identité du serveur, donc le serveur présente son certificat et le client grâce à son navigateur vérifie et confirme l'authenticité du serveur avant toute transaction.

Dans le mTLS (Mutual TLS), les deux entité communiquant entre elles vérifient mutuellement afin d'assurer l'authenticité de chacune d'elles avant la moindre transaction.

Cela s'implémente de la façon suivante :

- 1) Lorsque le Pod A veut envoyer des données au Pod B, il commence par demander au Pod B de lui présenter son certificat
- 2) Le Pod B répond en envoyant son certificat et demande à son tour celui du POD A
- 3) Après avoir validé le certificat du Pod B, le Pod A va répondre à son tour en lui envoyant son certificat et sa clé privée y associé
- 4) Le Pod B va également valider le certificat du Pod A et vérifier s'il est valide
- 5) A partir de là, tous les échanges entre ces deux pods seront dorénavant encryptés à l'aide des clés privées.

Dans un environnement cluster à plusieurs applications et Pods, il devient difficile de mettre en place cette sécurité applicative afin de crypter les communications entre les pod, il est donc recommandé de laisser les applications des Pods communiquer normalement sans cryptage et utiliser des outils à cet effet afin d'ajouter une couche de cryptage en amont de chaque pod. C'est dans ce sens qu'on utilise les applications comme **Istio** et **Linkerd** afin de mettre en place le cryptage mTLS entre tous ou certains pods d'un cluster



V) Supply Chain Security

5.1) Minimize base image footprint

Il est important de connaître les bonnes pratiques en matière de création d'images personnalisées et également en matière du choix de l'image de base à utiliser.

Il est important dans un premier temps de rappeler le concept d'image de base. En effet une image de base est une image qui est buildée FROM Scratch, une image qui n'a pas de parent. Mais par abus de langage, on appelle généralement image de base toute image parent utilisée pour builder nos images personnalisées.

Les bonnes pratiques de sécurité lors de la construction d'images personnalisées sont :

- 1) Faire attention dans le choix de l'image de base et privilégier des images officielles (vérifiées et validées par Docker)
- 2) Réduire au maximum possible la taille de l'image, supprimer les fichiers non nécessaires ou temporaires
- 3) Installer juste les paquets nécessaires, supprimer certains paquets pouvant augmenter la surface d'attaque comme (le package manager, l'outil curl, wget, les outils réseaux...)
- 4) Créer et maintenez plusieurs images du même service fonctionnant dans vos différents environnements (Dev, Production...)
- 5) Utiliser le build multi-stage afin de réduire davantage la taille de l'image de prod

```

Dockerfile - debian:buster-slim
FROM scratch
ADD rootfs.tar.xz /
CMD ["bash"]

Dockerfile - httpd
FROM debian:buster-slim
ENV HTTPD_PREFIX /usr/local/apache2
ENV PATH $HTTPD_PREFIX/bin:$PATH
WORKDIR $HTTPD_PREFIX
<content trimmed>

Parent

Dockerfile - My Custom Webapp
FROM httpd
COPY index.html htdocs/index.html

```

Il existe un autre type d'images dites distroless (sans distribution) qui ne contiennent que l'application et ses dépendances et ne contiennent pas de Shell, pas de package managers, pas d'outils réseau, pas d'éditeur de texte pas d'applications non nécessaires.

<https://github.com/GoogleContainerTools/distroless>

Distroless Docker Images
 Contains:

- Application
- Runtime Dependencies

 Does not contain:

- Package Managers
- Shells
- Network Tools
- Text editors
- Other unwanted programs

5.2) Image Security

Sur kubernetes, pour utiliser une image d'un registre privé, il faut correctement écrire le nom de l'image en prenant en compte l'adresse et le compte utilisateur, puis il faut créer un secret de type docker-registry qu'on fournira au pod à l'aide du paramètre `imagePullSecrets` qui se trouve au niveau de ses specs.



```

image: docker.io/library/nginx
  Registry   User/ Account   Image/ Repository
  docker login private-registry.io
  docker run private-registry.io/apps/internal-app
  kubectl create secret docker-registry regcred \
    --docker-server= private-registry.io \
    --docker-username= registry-user \
    --docker-password= registry-password \
    --docker-email= registry-user@org.com
  
```

```

nginx-pod.yaml

apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: private-registry.io/apps/internal-app
  imagePullSecrets:
  - name: regcred
  
```

5.3) Whitelist Allowed Registries

Nous avons vu précédemment qu'il existe 02 Admission webhook spéciaux capables d'envoyer des requêtes vers des serveurs externes afin de valider ou refuser la requête fonction de leur propre logique. On pourrait utiliser ces concepts comme vu précédemment afin de mettre cela sur pied grâce à un admission webhook serveur ou OPA serveur. Mais compte tenu de la partie code et logique propre à mettre en place, cela peut s'avérer fastidieux.

Néanmoins Kubernetes possède un autre admission controller « [ImagePolicyWebhook](#) » permettant de configurer la politique liée aux images. Les étapes à suivre pour le mettre en place sont les suivantes :

- 1) Créer le serveur Admission Webhook serveur contenant la logique de validation des images fiables (whitelist)
- 2) Créer le fichier KubeConfig devant permettre de se connecter au serveur Webhook et créer également le manifest de configuration de l'admission controller kubernetes ([/etc/kubernetes/admission-config.yaml](#))
- 3) Activer l'admission webhook grâce au paramètre [--enable-admission-plugins=ImagePolicyWebhook](#)
- 4) Mettre à jour le fichier de configuration de l'ApiServer en lui fournissant le fichier de configuration de l'admission controller via l'option [--admission-control-config-file=/etc/kubernetes/admission-config.yaml](#)

Dans ce fichier de configuration, le Type de ressource est `AdmissionConfiguration`



```

/etc/kubernetes/admission-config.yaml

apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
  - name: ImagePolicyWebhook
    configuration:
      imagePolicy:
        kubeConfigFile: <path-to-kubeconfig-file>
        allowTTL: 50
        denyTTL: 50
        retryBackoff: 500
        defaultAllow: true
  
```

```

kube-apiserver.service

ExecStart=/usr/local/bin/kube-apiserver \\
  --advertise-address=${INTERNAL_IP} \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --authorization-mode=Node,RBAC \\
  --bind-address=0.0.0.0 \\
  --enable-swagger-ui=true \\
  --etcd-servers=https://127.0.0.1:2379 \\
  --event-ttl=1h \\
  --runtime-config=api/all \\
  --service-cluster-ip-range=10.32.0.0/24 \\
  --service-node-port-range=30000-32767 \\
  --v=2
--enable-admission-plugins=ImagePolicyWebhook
--admission-control-config-file=/etc/kubernetes/admission-config.yaml
  
```

Ce fichier de configuration est celui qui sera utilisé par kubernetes afin de se connecter au serveur webhook précédemment déployé. Dans ce fichier, on doit fournir obligatoirement un fichier de configuration au format KubeConfig (dans lequel le cluster représente en effet le serveur Webhook et le user celui qui utilisé par kubernetes pour se connecter au dit serveur webhook) et également la stratégie de validation des requêtes à utiliser « `defaultAllow: true :false` ». Dans cet exemple par défaut si on

Admission
Webhook
Server

n'a pas explicitement refusé la source d'une image, alors toutes les sources sont autorisées, si l'admission webhook Server n'est plus disponible, alors toutes les requêtes seront autorisées.

```
<path-to-kubeconfig-file>
clusters:
- name: name-of-remote-imagepolicy-service
  cluster:
    certificate-authority: /path/to/ca.pem
    server: https://images.example.com/policy

users:
- name: name-of-api-server
  user:
    client-certificate: /path/to/cert.pem
    client-key: /path/to/key.pem
```

/etc/kubernetes/admission-config.yaml

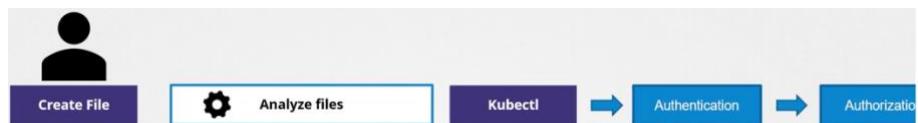
```
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
- name: ImagePolicyWebhook
  configuration:
    imagePolicy:
      kubeConfigFile: <path-to-kubeconfig->
      allowTTL: 50
      denyTTL: 50
      retryBackoff: 500
      defaultAllow: true
```

5.4) Use Static Analysis of User Workloads

L'idée ici est d'ajouter une couche de sécurité au pipeline de création des ressources kubernetes ? comme on l'a vu jusqu'à présent, les étapes de ce pipeline sont les suivantes :

[Create files](#) > [kubectl apply](#) > [Authentication](#) > [authorizations](#) > [Admission Controller](#) > [create resource](#)

En effet, les bonnes pratiques recommandent d'introduire la sécurité dans ce pipeline au début du pipeline, ce qui revient à ajouter une étape de scan des différents manifests créés avant de les pousser sur le cluster kubernetes à l'aide de kubectl



pour cette étape d'analyse des manifests, il existe plusieurs outils permettant de le faire parmi lesquels « [kubesec](#) » que nous allons utiliser

Kubesec peut s'utiliser de plusieurs façons différentes :

- 1) Installer le binaire kubesec et lancer le scan avec la commande « [kubesec scan pod.yaml](#) » à la fin de chaque scan, [kubesec](#) produit un rapport avec recommandations et le score.

▶ kubesec scan pod.yaml

- 2) Utiliser une requête Curl afin d'envoyer votre manifest directement au serveur public kubesec via une requête http « [curl -sSX POST --data-binary @'pod.yaml' https://v2.kubesec.io/scan](#) »

▶ curl -sSX POST --data-binary @"pod.yaml" https://v2.kubesec.io/scan

- 3) Installer le binaire kubesec et lancer une instance kubesec que vous exposerez sur un port de votre machine « [kubesec http 8080 &](#) »

▶ kubesec http 8080 &

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-pod
spec:
  containers:
  - name: ubuntu
    image: ubuntu
    command: ["sleep", "3600"]
    securityContext:
      privileged: True
      runAsUser: 0
      capabilities:
        add: ["CAP_SYS_BOOT"]
  volumes:
  - name: data-volume
    hostPath:
      path: /data
      type: Directory
```

```
{
  "object": "Pod/sample-pod.default",
  "valid": true,
  "fileName": "API",
  "message": "Failed with a score of -30 points",
  "score": -30,
  "scoring": {
    "critical": [
      {
        "id": "Privileged",
        "selector": "containers[] .securityContext .privileged == t",
        "reason": "Privileged containers can allow almost complete"
        "points": -30
      }
    ],
    "advise": [
      {
        "id": "ApparmorAny",
        "selector": ".metadata .annotations .\"container.apparmor.s",
        "reason": "Well defined AppArmor policies may provide great"
        "points": 3
      },
      {
        "id": "ServiceAccountName",
        "selector": ".spec .serviceAccountName",
        "reason": "Service accounts restrict Kubernetes API access"
        "points": 3
      }
    ]
}
```

L'exécution du scan kubesec génère un rapport de scan ressortant dans un premier temps le score et le statut de votre scan ; en suite pour chaque problème soulevé, kubesec affiche dans le détail l'élément qui cause problème et également la raison pour laquelle ce problème est à prendre en compte.

5.5) Scan images for known vulnerabilities (Trivy)

Les scanner de vulnérabilités utilisent généralement la base de données CVE (Common vulnerabilities exposures) qui est en effet une base de données qui regroupe toutes les vulnérabilités connues à ce jour pour certaines applications ou certains systèmes d'exploitation.

Trivy est une solution d'Aqua Security permettant le scan de vulnérabilités de containers et également de certains artefacts ? Il est idéal, simple d'utilisation et s'intègre facilement aux pipelines CI/CD

L'installation de Trivy est simple et le script d'installation se trouve dans la documentation officielle de Trivy

Après avoir installé Trivy, les commandes usuelles sont les suivantes :

- `trivy image nginx :1.18.0` : permet de scanner l'image nginx/1.18.0 (à la fin du scan un rapport est affiché)
- `trivy image --severity CRITICAL nginx:1.18.0` : ajoute l'option --severity afin de n'afficher que les vulnérabilités critiques
- `trivy image --severity CRITICAL,HIGH nginx:1.18.0` : afficher les vulnérabilités critiques et élevées pour l'image nginx :1.18.0
- `trivy image --ignore-unfixed nginx :1.18.0` : affiche les vulnérabilités déjà résolues par le fournisseur de l'image et pouvant être rapidement résolues par vous soit en upgradant le tag de l'image ou une application utilisée par l'image et ignore toutes celles n'ayant pas encore été résolues
- `docker save nginx:1.18.0 > nginx.tar` : permet de créer un artefact contenant l'image nginx ; cet artefact peut être également scanné par Trivy
- `trivy image --input archive.tar` : scan d'un artefact contenant une image

VI) Monitoring, Logging & runtime Security

A ce stade, nous avons vu pas mal de façons de sécuriser son infrastructure ou cluster Kubernetes, mais même après avoir mis en place toutes ces couches de sécurité, nous ne sommes pas totalement à l'abri de failles ou d'intrusions. C'est la raison pour laquelle même après avoir mis en place tous ces outils de sécurité, il est important de mettre sur pied un outil de monitoring capable de filtrer les différents processus afin de détecter ceux qui pourraient s'apparenter à des attaques ou à des intrusions. Pour cela, nous utiliserons l'outil Falco de Sysdig. Falco est un outil capable d'analyser notre système et filtrer les événements qui semblent suspect comme :

- Une tentative d'accéder au fichier `/etc/shadow` (fichier où sont stockés les mots de passe)
- Une tentative de suppression des logs d'une application ...etc



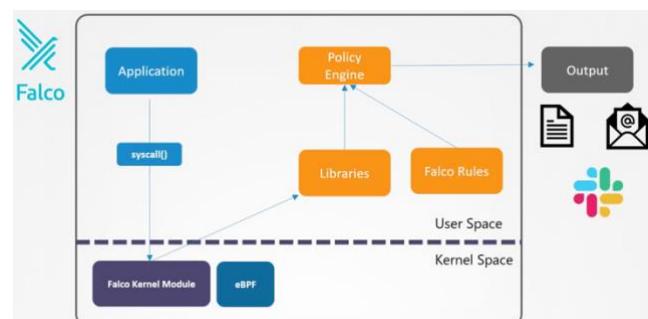
6.1) Falco Overview and installation

Avant d'installer Falco, il est important de savoir comment il fonctionne. En effet, pour fonctionner Falco a besoin d'avoir un accès direct au kernel afin de scanner les Syscalls qui y seront fait et filtrer ceux qui semblent suspects. Pour ce faire, Falco lors de son installation met en place son propre module Kernel dans le kernel Space « Falco Kernel Module ». Falco sait également interagir avec le Kernel en utilisant la technologie eBPF comme Tracee, cette technologie peut être utilisée dans le cas où l'on ne souhaite pas avoir le Falco Kernel Module.

Une fois Falco installé, il utilise les librairies et ses stratégies de filtrage afin de détecter les événements suspects et les envoyer en sortie soit par mail, Slack ou autre méthode de notification.

Pour l'installation de Falco, on peut :

- Installer en utilisant le gestionnaire de paquets Linux à partir de la procédure d'installation (Doc officielle)
- Utiliser Helm afin de l'installer comme daemonSet sur tous les nœuds du cluster



```

curl -s https://falco.org/repo/falcosecurity-3672BA8F.asc | apt-key add -
echo "deb https://download.falco.org/packages/deb stable main" | tee /etc/apt/sources.list.d/falco.list
apt update -y
apt get install -y linux-headers-$(uname -r)
apt install -y falco

```

```

helm repo add falcosecurity https://falcosecurity.github.io/charts
helm repo update
helm install falco falcosecurity/falco
NAME: falco
LAST DEPLOYED: Wed Mar  7 20:19:25 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:

```

6.2) Use Falco to detect Threads

Après avoir installé Falco, nous allons l'utiliser pour détecter les événements suspects et alerter les personnes concernées. Pour commencer, il faut vérifier que Falco soit bien installé et fonctionne correctement

- Installation directement sur l'hôte ou sur un nœud

- ***systemctl status falco*** : permet de voir si le service Falco est bien activé
- ***journalctl -fu falco*** : permet d'avoir les journaux d'évènements du service falco (l'option -f permet d'afficher les logs au fil de l'eau). On se rend compte au début qu'il n'affiche aucun logs, tout simplement parce qu'il n'affiche que les logs suspects. Si on essaie d'accéder au terminal Bash d'un container ou au fichier /etc/shadow, on verra que des logs s'afficheront

```

journalctl -fu falco
.
.
.
22:57:09.163982780: Notice A shell was spawned in a container
user_loginuid=-1 k8s.ns=default k8s.pod=nginx container=c73d9fc1a75d
cmdline=bash terminal=34816 container_id=c73d9fc1a75d image=n
container=c73d9fc1a75d

23:09:03.279503809: Warning Sensitive file opened for reading
user_loginuid=-1 program=cat command=cat /etc/shadow file=/et
ggparent=containererd-shim ggpaprent=containererd-shim container_
k8s.ns=default k8s.pod=nginx container=c73d9fc1a75d k8s.ns=de
container=c73d9fc1a75d

Terminal 1
kubectl exec -ti nginx -- bash
root@nginx:/# cat /etc/shadow

```

Afin de décider quels évènement filtrer, Falco utilise les règles Falco qui sont en effet définies dans un fichiers de configuration Falco « ***rules.yaml*** » qui est en effet un fichier Yaml qui contient 3 types d'éléments (les règles, les listes et les macros).

Les règles du fichier rules.yaml doivent permettre de définir toutes les conditions possibles pouvant déclencher une alerte. Ces conditions sont définies à l'aide de 05 paramètres

- ***rule*** : permet de définir le nom de la règle, doit être unique)
- ***desc*** : longue description de la règle à mettre en place
- ***condition*** : qui est une expression de filtrage qui est appliquée sur les évènements afin d'appliquer la règle
- ***output*** : le message de sortie qui apparaitra dans les logs si cet règles est appliquée
- ***priority*** : ensuite on a priorité qui représente le degré de严重性 donné à la règle

```

rules.yaml
- rule: <Name of the Rule>
  desc: <Detailed Description of the Rule>
  condition: <When to filter events matching the rule>
  output: <Output to be generated for the Event>
  priority: <Severity of the event>

```

Ci-dessous, un exemple de règle s'activant en cas d'ouverture d'un terminal Bash dans un container

```

rules.yaml
- rule: Detect Shell inside a container
  desc: Alert if a shell such as bash is open inside the container
  condition: container.id != host and proc.name = bash
  output: Bash Shell Opened (user=%user.name %container.name)
  priority: WARNING

```

Dans cette règle, on souhaite que l'évènement soit déclaré si et seulement si le process Bash est lancé dans un container, pour cela on utilise le filtre container.id != host qui dit en effet que l'ID du container qui héberge le processus ne doit pas être l'hôte et le nom du processus doit être bash.

« container.id » et « proc.name » sont connu comme étant des filtres Sysdig et ces filtres sont très utilisés par Falco. Il existe plusieurs autres filtres Sysdig comme :

- ***container.id*** : permet d'avoir l'ID du container
- ***proc.name*** : permet d'avoir le nom du processus
- ***fd.name*** : qui est le nom du descripteur de fichier permettant de savoir quelles opérations sont réalisées sur un fichier (Read, write ...)
- ***evt.type*** : utilisés pour avoir ou filtrer les system calls à partir de leur nom (evt.type=execve())
- ***user.name*** : utiliser pour filtrer sur le nom de l'utilisateur dont l'action a généré l'évènement
- ***container.image.repository*** : permet de filtrer une image spécifique à partir de son nom



Les différentes valeurs pour le paramètre priority sont (plus petite à la plus grande) : DEBUG, INFORMATIONAL, NOTICE, WARNING, ERROR, CRITICAL, ALERT, EMERGENCY

```
rules.yaml
- rule: Detect Shell inside a container
  desc: Alert if a shell such as bash is open inside the container
  condition: container.id != host and proc.name in (linux_shells)
  output: Bash Opened (user=%user.name container=%container.id)
  priority: WARNING
- list: linux_shells
  items: [bash, zsh, ksh, sh, csh]
```

```
rules.yaml
- rule: Detect Shell inside a container
  desc: Alert if a shell such as bash is open inside the container
  condition: container.id != host and proc.name in (linux_shells)
  output: Bash Opened (user=%user.name container=%container.id)
  priority: WARNING
- list: linux_shells
  items: [bash, zsh, ksh, sh, csh]
- macro: container
  condition: container.id != host
```

Il est possible d'utiliser les List et les macro pour simplifier notre fichier rules.yaml

6.3) Falco Configuration Files

Dans cette partie, nous verrons les différents fichiers de configuration de Falco et comment appliquer ou mettre à jour les règles personnalisées que nous avons créés à Falco.

Le fichier principal de configuration de Falco est un fichier YAML situé dans le répertoire [/etc/falco/falco.yaml](#). C'est ce fichier de configuration qui est utilisé par falco à son démarrage

Ce répertoire par défaut où se trouve le fichier de configuration de Falco peut être trouvé en inspectant le retour des commandes `systemctl status falco` ou `journalctl -fu falco`

```
▶ journalctl -fu falco
-- Logs begin at Tue 2021-04-13 21:45:35 UTC, end at Tue 2021-04-13 21:51:31 UTC. --
Apr 13 21:45:36 node01 systemd[1]: Starting Falco: Container Native Runtime Security...
Apr 13 21:45:36 node01 systemd[1]: Started Falco: Container Native Runtime Security.
Apr 13 21:45:36 node01 falco[9817]: Falco version 0.28.0 (driver version 5c0b863ddade7a45568c0ac97d03742)
Apr 13 21:45:36 node01 falco[9817]: Tue Apr 13 21:45:36 2021: Falco version 0.28.0 (driver version 5c0b863ddade7a45568c0ac97d03742)
Apr 13 21:45:36 node01 falco[9817]: Falco initialized with configuration file /etc/falco/falco.yaml
Apr 13 21:45:36 node01 falco[9817]: Tue Apr 13 21:45:36 2021: Falco initialized with configuration file /etc/falco/falco.yaml
```

Ce fichier YAML falco.yaml contient en effet toutes les options de configuration utilisées par FALCO (la position du fichier des règles, les options utilisés pour les logs et les messages de sortie ...)

```
/etc/falco/falco.yaml
#
# Copyright (C) 2021 The Falco Authors.
#
rules_file:
- /etc/falco/falco_rules.yaml
- /etc/falco/falco_rules.local.yaml
- /etc/falco/k8s_audit_rules.yaml
- /etc/falco/rules.d

json_output: false
log_stderr: true
log_syslog: true
log_level: info
priority: debug
stdout_output:
  enabled: true

file_output:
  enabled: true
  filename: /opt/falco/events.txt

program_output:
  enabled: true
  program: "jq '{text: .output}' | curl -d @- -X POST https://some.url/some/path"

http_output:
  enabled: true
  url: http://some.url/some/path/
```

- **rules_file** : liste l'ensemble des fichiers de configuration de règles utilisés par Falco. L'ordre est important car le dernier fichier a la priorité sur tous les précédents.
- **Json_output** : quand activés, les logs sont au format JSON, par défaut il est à false et les logs au format texte
- **log_stderr, log_syslog, log_level, priority** : ce sont les paramètres liés à Falco concernant les logs d'utilisation de Falco et non les logs des événements filtrés (par défaut concernant Falco tous les événements à priorité supérieure à debug seront loggés les autres non)
- **stdout_output** : permet d'activer la sortie des événements afin de pouvoir les afficher à l'écran ou les envoyer vers un autre processus
- **file_output** : permet d'enregistrer automatiquement les événements filtrés dans un fichier de votre choix
- **program_output** : permet de renvoyer les logs vers une autre application (come Slack, Gmail...)
- **http_output** : permet de renvoyer les logs vers une url via une requête http

Après chaque modification de ce fichier de configuration Falco, pour que les modifications prennent effet, il faut au préalable recharger ce fichier et redémarrer le service falco.

Concernant les règles Falco, le fichier [/etc/falco/falco_rules.yaml](#) est le fichier par défaut des règles Falco. Par défaut, Falco prend en compte une multitudes de règles permettant de filtrer les événements suspects, ces règles sont définies dans ce fichier et nous pouvons les modifier ou alors ajouter nos propres règles dans le même fichier. Mais cela n'est pas la bonne pratique car toutes les règles soumises faites dans le fichier de règles par défaut de Falco seront supprimées si jamais Falco est mis à jour. La bonne pratique est d'ajouter ses propres règles plutôt dans le fichier [/etc/falco/falco_rules.local.yaml](#) qui bien sûr a la priorité sur le fichier [/etc/falco/falco_rules.yaml](#).

Pareil également pour que les modifications soient prises en compte, il faut recharger la configuration Falco et redémarrer le service.

Pour recharger la configuration Falco sans toutefois redémarrer le service Falco (Hot Reload)

- On recherche l'ID du processus Falco : `cat /var/run/falco.pid`
- Après avoir eu ce processus, on le tue afin qu'un nouveau soit lancé par le service falco : `kill -1 $(cat /var/run/falco.pid)`

6.4) Immutable Infrastructure

Une ressource dite immuable est une ressource dont on ne peut pas modifier les paramètres. Pour les infrastructures immuables, modifier un paramètre revient à supprimer l'infrastructure et en construire une nouvelle avec le paramètre mis à jour. Il est important de rendre nos containers immuables afin d'empêcher d'éventuels hackers de prendre la main sur nos container et d'y modifier quelconque fichiers que ce soit

6.5) Ensure Immutability of Containers at Runtime

Pour rendre un container immuable, il faut juste s'assurer qu'après le lancement du container, qu'on ne puisse plus écrire dans son système de fichiers. Cette configuration peut être mise en place grâce au bloc `SecurityContext` du container et à l'aide du paramètre « `readOnlyRootFilesystem: true` » permettant de mettre le système de fichier en mode lecture seule. Malheureusement en appliquant juste cette configuration, le Pod ou le container ne réussira pas à lancer l'application à l'intérieur car généralement les applications ont besoin d'écrire dans certains fichiers lors de leur fonctionnement. Pour corriger ce problème, il faut juste monter ces fichiers ou répertoires pour lesquels l'application a besoin d'un accès en écriture dans des volumes afin qu'ils puissent échapper à la politique du Read Only du container.

Dans cet exemple ci-après, on rend notre container nginx immuable, et vu que nginx a besoin d'écrire pour son bon fonctionnement dans les répertoires `/var/run` et `/var/cache/nginx`, on les monte dans des volumes vides car l'on ne souhaite pas sauvegarder le contenu de ces répertoire-là dans notre cas d'exemple.

The screenshot shows a terminal session and a YAML configuration file. The terminal shows the creation of a pod named 'nginx' from a YAML file, the listing of pods, and the logs of the 'nginx' pod. The logs indicate errors related to file system permissions when trying to create directories and open files in '/var/run' and '/var/cache/nginx'. The right side shows the 'nginx.yaml' configuration file which defines a Pod with a single container named 'nginx' running an 'nginx' image. It sets the security context to 'readOnlyRootFilesystem: true'. Two volumes are mounted: 'cache-volume' at '/var/cache/nginx' and 'runtime-volume' at '/var/run'. Both volumes are empty directories.

```
nginx.yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: nginx
    name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
  securityContext:
    readOnlyRootFilesystem: true
  volumeMounts:
  - name: cache-volume
    mountPath: /var/cache/nginx
  - name: runtime-volume
    mountPath: /var/run
volumes:
- name: cache-volume
  emptyDir: {}
- name: runtime-volume
  emptyDir: {}
```

```
kubectl create -f nginx.yaml
pod/nginx created

kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
nginx     0/1     Error    0          20s

kubectl logs nginx
root@controlplane:~# kubectl logs nginx
:
2021/04/12 15:14:39 [emerg] 1#1: mkdir() "/var/cache/nginx/client_temp" failed (30: Read-only file system)
:
2021/04/12 16:11:26 [emerg] 1#1: open() "/var/run/nginx.pid" failed (30: Read-only file system)
nginx: [emerg] open() "/var/run/nginx.pid" failed (30:
```

NB : en plus du paramètre `readOnly`, se rassurer que le paramètre ne soit pas lancé en mode privilégiés car si c'est le cas avec les privilégiés du root vous pourrez toujours écrire dans le système de fichier et de ce fait votre container est en réalité toujours mutable.

6.6) Use Audit Logs to monitor access

Il est possible de moniter les événements du cluster kubernetes -, car par défaut kubernetes dispose d'un outil permettant de capturer les différents événements du cluster. Néanmoins cet outil n'est pas activé dans la configuration par défaut de kubernetes et nécessite d'être activé et configuré avant utilisation.

Avant d'activer et configurer cet outil, il est important de présenter déjà les différents états possibles d'une requête dans kubernetes. Une requête dans kubernetes peut prendre les états suivants :

- **RequestReceived** : dès qu'une requête est reçue par ApiServer, elle prend l'état `RequestReceived`, peu importe si elle est valide ou pas, authentifiée ou pas.
 - **ResponseStarted** : événement généré lorsque la requête est authentifiée, autorisée et validée, cela est généralement utilisé pour les longues requêtes avec l'option `Watch` qui retarde la complétion de la requête
 - **ResponseComplete** : lorsque la requête a été complétée, une réponse est envoyée et un événement `ResponseComplete` est créé
 - **Panic** : cet événement est créé en cas de requête invalide ou d'erreur
- | |
|--------------------|
| 1.RequestReceived |
| 2.ResponseStarted |
| 3.ResponseComplete |
| 4. Panic |

NB : Tous ces événements liés aux états des requêtes peuvent être enregistrés et loggés par le kube-apiserver, il suffit juste d'activer et configurer le module de Logging de kubernetes. Pour ce faire les étapes suivantes sont nécessaires :

- 1) Créer le fichier de règles de filtrages des événements que vous souhaitez logger car compte tenu des différents états d'une requête dans kubernetes, il peut être nécessaire de filtrer pour ne logger que les événements concernant un état, une ressource, une action, un namespace ou autre.
- 2) Modifier le fichier de configuration de l'ApiServer afin d'activer et configurer le module de logs grâce au paramètres `--audit-log-path=/var/log/k8-audit.log`, qui permet de spécifier à kubernetes dans quel fichier il doit stocker les logs.

3) Modifier le fichier de configuration de l'ApiServer afin de lui fournir le fichier contenant les règles de filtrage à utiliser via l'option --audit-policy-file=/etc/kubernetes/audit-policy.yaml

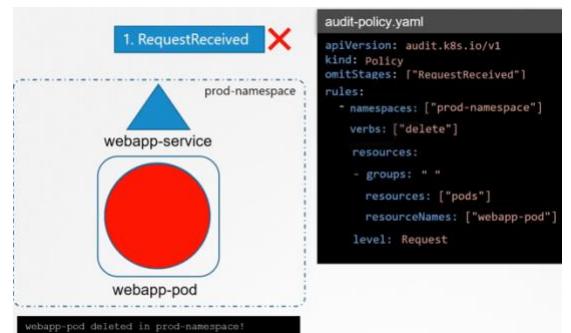
Il existe certains paramètres de configuration optionnel que l'on peut ajouter au fichier de configuration de l'apiserver afin de modifier le comportement du module de log :

- « **--audit-log-maxage=1** »: définir la durée maximum des logs à conserver, exemple 10jours max
- « **--audit-log-maxbackup=5** »: spécifier le nombre maximum de fichier d'audit pouvant être créés sur l'hôte
- « **--audit-log-maxsize=100** » : spécifier la taille max du fichier de log en MB

NB : Lors de la configuration, pensez à monter les répertoire ou fichier de configuration dans des volumes afin qu'ils soient disponibles à l'intérieur des containers.

```
/etc/kubernetes/manifests/kube-apiserver.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
    - command:
        - kube-apiserver
        - --authorization-mode=Node,RBAC
        - --advertise-address=172.17.0.107
        - --allow-privileged=true
        - --enable-bootstrap-token-auth=true
        - --audit-log-path=/var/log/k8s-audit.log
        - --audit-policy-file=/etc/kubernetes/audit-policy.yaml
        - --audit-log-maxage=10
        - --audit-log-maxbackup=5
        - --audit-log-maxsize=100
```

Concernant le fichier de configuration des règles de filtrages, il s'agit d'un fichier similaires à un manifest de création de ressources kubernetes. Par exemple regardons ensemble ce fichier qui contient une règle permettant de filtrer et de ne enregistrer que les événements du cluster concernant tous les états sauf celui de RequestReceived, ensuite ne prendre en compte que les requête provenant d'un namespace particulier et ne concernant que la suppression des ressources de type POD.



audit-policy.yaml

```
audit-policy.yaml
apiVersion: audit.k8s.io/v1
kind: Policy
omitStages: ["RequestReceived"]
rules:
  - namespaces: ["prod-namespace"]
    verbs: ["delete"]
    resources:
      - groups: " "
        resources: ["pods"]
        resourceNames: ["webapp-pod"]
    level: RequestResponse

  - level: Metadata
    resources:
      - groups: " "
        resources: ["secrets"]
```

- **omitStages** : prend en paramètre une liste d'état de requêtes que l'on ne souhaite pas sauvegarder, dans cet exemple on ne souhaite loggés les événements de type RequestReceived
- **rules** : même concept qu'avec les RBAC cela permet de définir les ressources ou type de ressources ou namespaces pour lesquels on souhaite avoir les logs, dans le cas où une option de ce paramètre n'est pas définie, alors cela signifie que tous les logs concernant ce paramètres seront sauvegardés. Par exemple si on ne définit pas de verbs, alors toutes les actions faites sur les pods du namespace webapp-pod seront sauvegardé or là on se souhaite sauvegarder que les action de type delete.
- **level** : il s'agit ici du niveau de verbosité dans les logs liées à cette règle, cet option peut prendre plusieurs valeurs :
 - **Metadata** : dans les logs, pour les événements matchant avec cette règle, seuls les metadata liés à la requête seront affichés
 - **Request** : niveau de verbosité élevé par rapport à metadata, ici toute la requête est incluse dans les logs de l'évènement
 - **RequestResponse** : niveau de verbosité élevé par rapport à Request ; la réponse est incluse dans les journaux de logs concernant l'évènement
- NB** : ici on a 2 règles en tout et la deuxième permet d'enregistrer les logs de tous les événements pour toutes les actions sur les ressources de type secrets

VII) TIPS NOTES FOR EXAMS

1.1) CIS BENCHMARK

Lancer le script Assessor-CLI.sh en mettant les paramètres

```
-i : interactive ; -rp : prefix(index) ; -rd: répertoire de sauvegarde /var/www/html et -nts
```

Les Options de la commande CIS BENCHMARK : [-i -rd -rp -nts]

```
sh ./Assessor-CLI. -h (pour avoir l'aide)
```

```
sh ./Assessor-CLI.sh -i -rd /var/www/html/ -nts -rp index
```

1.2) Kube bench

Les commandes usuelles pour ce module

```
curl -L https://github.com/aquasecurity/kube-bench/releases/download/v0.4.0/kube-bench_0.4.0_linux_amd64.tar.gz -o kube-bench_0.4.0_linux_amd64.tar.gz  
tar -xvf kube-bench_0.4.0_linux_amd64.tar.gz  
./kube-bench -h  
./kube-bench --config-dir `pwd`/cfg --config `pwd`/cfg/config.yaml
```

1.3) Kubelet Security & Proxy

NB : Pour accéder à la documentation officielle (Faire la recherche de [kubelet authentification](#)).

Fichier de configuration de kubelet par défaut /var/lib/kubelet/config.yaml

Port 10250 : pour les utilisateurs ayant les droits et donnant accès à tous

Port 10255 : pour les utilisateurs n'ayant pas de droits donne accès en lecture seule

Kubectl proxy --port 8002 --address 0.0.0.0

```
curl -sk http://localhost:8002
```

1.4) Kubernetes Dashboard

NB : Pour accéder à la documentation officielle (Faire la recherche de [Kubernetes DashBoard](#)).

```
kubectl proxy --address=0.0.0.0 --disable-filter &
```

```
kubectl create clusterrolebinding readonly-user-binding --clusterrole view --serviceaccount kubernetes-dashboard:readonly-user
```

1.5) Verify binaries

NB : Pour accéder à la documentation officielle (Faire la recherche de [Kubernetes download](#)).

```
sha512sum ./downloads/kubernetes.tar.gz
```

```
tar -xvf kubernetes.tar.gz
```

```
tar -cvf /opt/kubernetes-modify.tar.gz kubernetes
```

1.6) SSH and Obsolete Package & Sudo Escalation

Les commandes usuelles pour ce module

```
usermod -s nologin sam : changer Le shell d'un user  
useradd -s /bin/bash -d /home/sam -u 2323 -G admin  
chown -R sam:sam /home/sam  
visudo && jim    ALL=(ALL:ALL) NOPASSWD:ALL  
apt list --installed  
systemctl list units --type=service  
Lsmod : lis kernel modules loaded  
vi /etc/modprobe.d/blacklist.conf : fichier pour blaclister les module kernel  
apt auto-remove nginx  
netstat -nltp  
kill pid or if you want to hot restart process kill -1 pid  
ufw status numbered : afficher L'état de ufw ainsi que la liste ordonée des règles  
ufw allow 1000:2000/tcp : ouvre Les ports 1000 à 2000 en TCP
```

```
ufw reset : réinitialiser ufw à son état initial  
ufw allow from 135.22.65.0/24 to any port 9090 proto tcp  
ufw deny 80/tcp  
ufw --force enable
```

1.7) Tracee

Commandes usuelles

```
strace <commande> : afficher les Syscalls appelés par ladite commande  
strace -c <commande> : afficher tous les Syscalls appelés par ladite commande  
strace -p pid : afficher les Syscalls du processus dont l'ID est passé en paramètre
```

Tracee Dir to mount as a volume when run Tracee as a container

- /lib/modules:ro
- /usr/src:ro
- /tmp/tracee

1.8) Seccomp

NB : Pour accéder à la documentation officielle (Faire la recherche de [Seccomp](#)).

Pour vérifier si Seccomp est activé sur un hôte, on fait : `grep -i seccomp /boot/config-$(uname -r)` et on se rassure que le fichier contienne la ligne `CONFIG_SECCOMP=Y`

Les différents types de profils Seccomp dans Kubernetes

- `RuntimeDefault` : On utilise le profil Seccomp par défaut du container runtime qui bloque environ 64 Syscalls
- `Localhost` : permet de charger ses règles Seccomp à partir d'un fichier sur l'hôte ; ce fichier doit être dans le `/var/lib/kubelet/seccomp/profiles/<custom.json>`
- `unconfined` : désactive le Seccomp sur le pod en question et de ce fait tous les Syscalls peuvent être appelés

1.9) AppArmor

NB : Pour accéder à la documentation officielle (Faire la recherche de [AppArmor](#)).

Pour vérifier si AppArmor est activé sur un hôte, on fait : `cat /sys/module/apparmor/parameters/enabled` et on se rassure que le fichier contienne `Y` qui confirme en effet que AppArmor est bien installé

Commandes usuelles

- `apt install -y apparmor-utils` : install les outils permettant de gérer les profils AppArmor
- `aa-status` : permet de voir les différents profils AppArmor chargés dans le Kernel
- `aa-genprof <bash Script>` : permet de créer le profil AppArmor pour le script bash passé en paramètre
- `ls /etc/apparmor.d` : répertoire par défaut contenant les profils AppArmor
- `apparmor_parser /etc/apparmor.d/<root.add_data.sh>` : permet de charger le profil apparmor dans le kernel

AppArmor dans Kubernetes

Pour charger un profil AppArmor sur un pod, il faut se rassurer que le profil soit bien chargé dans le kernel de l'hôte en question et par la suite dans le manifest de définition du pod, rajouter l'annotation :

- `container.apparmor.security.beta.kubernetes.io/<container_name>: localhost/<AppArmor_profileName>`

1.10) Admission Controllers

NB : Pour accéder à la documentation officielle (Faire la recherche de [admission](#)).

Pour activer un Admission controller par défaut désactivé de kubernetes il faut modifier le fichier de création du static pod kube-apiserver et ajouter l'option `--enable-admission-plugins=PodSecurityPolicy,<AdmissionName>`

Pour désactiver un Admission controller par défaut activé de kubernetes il faut modifier le fichier de création du static pod kube-apiserver et ajouter l'option `--disable-admission-plugins=PodSecurityPolicy,<AdmissionName>`

1.11) Validating and Mutating admission Webhook

NB : Pour accéder à la documentation officielle (Faire la recherche de [admission webhook](#) ou [Dynamic admission control](#)).

Se référer à la documentation officielle de kubernetes afin d'avoir le Template de création de la ressource [ValidatingWebhookConfiguration](#) ou [MutatingWebhookConfiguration](#) nécessaire afin que les admission webhook de kubernetes sache vers quel serveur envoyer ses requêtes admission review

1.12) Pod Security Policy

NB : Pour accéder à la documentation officielle (Faire la recherche de [psp](#)).

Ressource kubernetes permettant de définir un certain nombre de conditions que doit avoir un pod afin d'être accepté dans le système

Pour mettre en place une règle PSP dans le cluster, il faut au préalable :

- 1) créer ladite règle en créant la ressource PodSecurityPolicy
- 2) Créer un RBAC clusterrole afin de donner l'autorisation à tous les pods de pouvoir utiliser cette PSP (serviceaccount : default)
- 3) Activer l'admission controller PSP à l'aide de l'option `--enable-admission-plugins=PodSecurityPolicy` dans le fichier de configuration du kube-apiserver

1.13) OPA in Kubernetes

NB : Pour accéder à la documentation officielle (Faire la recherche de [OPA](#)).

Pour mettre en place un serveur OPA sur kubernetes, il faut :

- 1) Déployer le serveur OPA comme un pod et l'exposer à l'aide d'un service
- 2) Déployer le container kube-mgmt à l'intérieur du pod OPA afin de répliquer la configuration des ressources vers OPA
- 3) Créer la ressource ValidatingWebhookConfiguration qui pointe vers le service OPA afin de lui envoyer les admission review
- 4) Créer un configMap avec le label `openpolicyagent.org/policy : rego` qui contiendra comme donnée principale la règle OPA en rego. Cette règle sera automatiquement chargée au serveur OPA

1.14) Container Sandboxing (gVisor and Kata runtime)

NB : Pour accéder à la documentation officielle (Faire la recherche de [runtime class](#)).

Afin d'encapsuler les container des pods dans le but de les isoler les uns des autres ou de l'hôte, on doit juste changer le container runtime par défaut utilisé par kubernetes (soit `runsc` pour gVisor et `kata` pour kata runtime).

- 1) Il faut créer une ressource kubernetes appelé runtime Class (Template Doc officielle) et définir le handler en fonction du container runtime que l'on souhaite utiliser
- 2) Modifier le fichier de définition du pod et y ajouter dans le bloc spec le paramètre `runtimeClassName : <NewRun>`

1.15) Image Policy Webhook (whitelist)

NB : Pour accéder à la documentation officielle (Faire la recherche de [admission Controller](#) et une fois dans la page [AdmissionConfiguration](#)).

Kubernetes possède également un autre admission webhook permettant de gérer des règles de sécurité liées aux images, pour le mettre en place il faut :

- Déployer également son serveur webhook externe
- Créer le fichier kubeconfig permettant d'accéder à ce serveur webhook
- Créer le fichier de configuration d'admission qui aura les paramètres de connexion à ce serveur (Template Doc k8s)
- Activer l'admission controller dans le fichier de configuration de l'apiserver grâce à l'option `--enable-admission-plugins=ImagePolicyWebhook` et `--admission-control-config-file=<path to AdmissionConf file>`; le fichier de configuration sur l'hôte doit être monté dans un volume et passé au pod kube-apiserver

1.16) Kubesec

Commandes usuelles

- `Kubesec scan pod.yml` : scan du manifest kubernetes à l'aide du binaire kubesec local
- `Curl -sSX PODT --data-binary @"pod.yml" https://v2.kubesec.io/scan` : scan à l'aide d'une requête vers le serveur kubesec
- `Kubesec http 8080 &` : exposition du server kubesec sur le port 8080 de l'hôte.

1.17) Trivy

Commandes usuelles

- `trivy image nginx :1.18.0` : permet de scanner l'image nginx/1.18.0 (à la fin du scan un rapport est affiché)
- `trivy image --severity CRITICAL nginx:1.18.0` : ajoute l'option --severity afin de n'afficher que les vulnérabilités critiques
- `trivy image --severity CRITICAL,HIGH nginx:1.18.0` : afficher les vulnérabilités critiques et élevées pour l'image nginx :1.18.0
- `trivy image --ignore-unfixed nginx :1.18.0` : affiche les vulnérabilités déjà résolues par le fournisseur de l'image et pouvant être rapidement résolues par vous soit en upgradant le tag de l'image ou une application utilisée par l'image et ignore toutes celles n'ayant pas encore été résolues
- `docker save nginx:1.18.0 > nginx.tar` : permet de créer un artefact contenant l'image nginx ; cet artefact peut être également scanné par Trivy
- `trivy image --input archive.tar` : scan d'un artefact contenant une image
- `trivy image --input alpine.tar --format json --output /root/alpine.json`

1.18) Falco & falco configuration files

Commandes usuelles

- `systemctl status falco` : permet de voir si le service Falco est bien activé
- `journalctl -fu falco` : permet d'avoir les journaux d'évènements du service afin de détecter les évènement suspect filtrés

Fichiers de configuration Falco

- `/etc/falco/falco.yaml` : fichier de configuration principal utilisé au démarrage
- `/etc/falco/falco_rules.yaml` : fichier par défaut contenant les règles de filtrages d'évènement Falco
- `/etc/falco/falco_rules.local.yaml` : fichier de modification ou de création de règles personnalisées ayant la priorité sur le fichier de règles par défaut

1.19) Container Immutability

NB : Pour accéder à la documentation officielle (Faire la recherche de [SecurityContext](#) et ensuite [filesystem](#)).

Pour s'assurer qu'un container est immuable, il faut bloquer l'écriture sur le filesystem de ce container et monter sur des volumes les répertoires susceptibles d'être utilisés par ce container .

- 1) Mettre le bloc `securityContext` dans le container et définir le paramètre `readOnlyRootFilesystem` à true
- 2) Monter sur des volumes les répertoires devant être utilisés par les applications du container

1.20) Audit Logs in kubernetes

NB : Pour accéder à la documentation officielle (Faire la recherche de [Auditing](#)).

Pour activer le filtrage et la sauvegarde des journaux d'évènement de kubernetes, il faut :

- 1) Créer la règle de filtrage qui sera utilisé par kubernetes (Template Doc k8s)
- 2) Activer le module de Logging de kubernetes en modifiant le `kube-apiserver.yaml` et y ajouter les options `--audit-policy-file=/etc/kubernetes/audit-policy.yaml` et `--audit-log-path=/var/log/kubernetes/audit/audit.log` et optionnellement `--audit-log-maxage` ; `--audit-log-maxbackup` ; `--audit-log-maxsize`
- 3) Créer les volumes dans le pod kube-apiserver afin de monter les différents fichiers de configuration audit-policy et répertoire de log afin qu'ils puissent être disponibles dans le container Kube-api.