# RAPPORT TP

*Elèves:*

Mathis FOUCADE
Jules PUGET
Axelle ROUZIER

*Enseignant:*

M LOUTSCH ETIENNE

## *Session 3: Kubernetes*

Paris 2025

1. **Déployer un Cluster Kubernetes avec Kind**

```
axelle@securite-containeur:~$ curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.20.
0/kind-linux-amd64
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    97  100    97    0     0    657      0 --:--:-- --:--:-- --:--:--   659
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100 6304k  100 6304k    0     0  3853k      0  0:00:01  0:00:01 --:--:-- 8041k
axelle@securite-containeur:~$ chmod +x ./kind
axelle@securite-containeur:~$ sudo mv ./kind /usr/local/bin/kind
[sudo] password for axelle:
```

```
axelle@securite-containeur:~$ curl -LO "https://dl.k8s.io/release/$(curl -L -s h
ttps://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   138  100   138    0     0    715      0 --:--:-- --:--:-- --:--:--   718
100 57.3M  100 57.3M    0     0  7908k      0  0:00:07  0:00:07 --:--:-- 8341k
axelle@securite-containeur:~$ chmod +x kubectl
axelle@securite-containeur:~$ sudo mv kubectl /usr/local/bin/
```

```
  GNU nano 6.2                        kind-config.yaml *
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
- role: control-plane
- role: worker
- role: worker
```

```
axelle@securite-containeur:~$ kind create cluster --config kind-config.yaml
Creating cluster "kind" ...
 ✓ Ensuring node image (kindest/node:v1.27.3) 🖼
 ✓ Preparing nodes 📦 📦 📦 📦
 ✓ Configuring the external load balancer ⚖️
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹️
 ✓ Installing CNI 🔌
 ✓ Installing StorageClass 💾
 ✓ Joining more control-plane nodes 🎮
 ✓ Joining worker nodes 🚜
Set kubectl context to "kind-kind"
You can now use your cluster with:

kubectl cluster-info --context kind-kind

Thanks for using kind! 😊
```

```
axelle@securite-containeur:~$ kubectl get nodes
NAME                 STATUS     ROLES           AGE     VERSION
kind-control-plane   Ready      control-plane   5m37s   v1.27.3
kind-control-plane2  Ready      control-plane   4m14s   v1.27.3
kind-worker          Ready      <none>          107s    v1.27.3
kind-worker2         NotReady   <none>          107s    v1.27.3
```

```
axelle@securite-containeur:~$ kubectl get namespaces
NAME                 STATUS   AGE
default              Active   6m7s
kube-node-lease      Active   6m7s
kube-public          Active   6m7s
kube-system          Active   6m8s
local-path-storage   Active   5m1s
```

```
axelle@securite-containeur:~$ kubectl version
Client Version: v1.33.0
Kustomize Version: v5.6.0
Server Version: v1.27.3
WARNING: version difference between client (1.33) and server (1.27) exceeds the
supported minor version skew of +/-1
```

Dans cette première partie du TP, nous avons déployé un cluster Kubernetes en utilisant Kind (Kubernetes IN Docker). Kind est un outil qui permet de créer des clusters Kubernetes locaux en utilisant des conteneurs Docker comme nœuds. Nous avons commencé par télécharger et installer Kind via la commande curl, puis nous avons installé kubectl, l'outil de ligne de commande pour Kubernetes.

Nous avons ensuite créé notre configuration dans un fichier kind-config.yaml qui définit l'architecture de notre cluster avec 2 nœuds control-plane et 2 nœuds worker. Après avoir lancé la commande de création du cluster, Kind a déployé l'ensemble des composants nécessaires, notamment en configurant le load balancer externe, en démarrant les nœuds control-plane, en installant CNI (Container Network Interface) et en joignant les nœuds worker.

La vérification de l'état du cluster avec la commande *kubectl get nodes* nous a permis de constater que les 4 nœuds étaient bien déployés, dont 2 en tant que control-plane et 2 en tant que worker. La commande *kubectl get namespaces* nous a montré les namespaces par défaut du cluster, notamment default, kube-node-lease, kube-public, kube-system et local-path-storage. La version de Kubernetes que nous avons déployée est la v1.27.3, comme indiqué par la commande *kubectl version*.

## 2. Expérimentation des RBAC

```
axelle@securite-containeur:~$ kubectl create ns test-rbac
namespace/test-rbac created
```

```
  GNU nano 6.2                          mon-pod.yaml *
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: test-rbac
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

```
axelle@securite-containeur:~$ kubectl logs nginx -n test-rbac
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2025/04/27 10:46:42 [notice] 1#1: using the "epoll" event method
2025/04/27 10:46:42 [notice] 1#1: nginx/1.27.5
2025/04/27 10:46:42 [notice] 1#1: built by gcc 12.2.0 (Debian 12.2.0-14)
2025/04/27 10:46:42 [notice] 1#1: OS: Linux 6.8.0-58-generic
2025/04/27 10:46:42 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2025/04/27 10:46:42 [notice] 1#1: start worker processes
2025/04/27 10:46:42 [notice] 1#1: start worker process 35
2025/04/27 10:46:42 [notice] 1#1: start worker process 36
```

```
  GNU nano 6.2                                  role-pod-reader.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: test-rbac
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
```

```
axelle@securite-containeur:~$ kubectl apply -f role-pod-reader.yaml
role.rbac.authorization.k8s.io/pod-reader created
```

```
axelle@securite-containeur:~$ kubectl get role -n test-rbac
NAME         CREATED AT
pod-reader   2025-04-27T10:54:23Z
axelle@securite-containeur:~$ kubectl describe role pod-reader -n test-rbac
Name:         pod-reader
Labels:       <none>
Annotations:  <none>
PolicyRule:
  Resources  Non-Resource URLs  Resource Names  Verbs
  ---------  -----------------  --------------  -----
  pods       []                 []              [get list]
```

```
  GNU nano 6.2                                  rolebinding-pod-reader.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods-binding
  namespace: test-rbac
subjects:
- kind: User
  name: titi
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

```
axelle@securite-containeur:~$ kubectl apply -f rolebinding-pod-reader.yaml
rolebinding.rbac.authorization.k8s.io/read-pods-binding created
```

```
axelle@securite-containeur:~$ docker cp kind-control-plane:/etc/kubernetes/pki/ca.crt .
Successfully copied 3.07kB to /home/axelle/.
axelle@securite-containeur:~$ docker cp kind-control-plane:/etc/kubernetes/pki/ca.key .
Successfully copied 3.58kB to /home/axelle/.
```

```
axelle@securite-containeur:~$ openssl genrsa -out titi.key 2048
axelle@securite-containeur:~$ openssl req -new -key titi.key -out titi.csr -subj "/CN=titi"
axelle@securite-containeur:~$ openssl x509 -req -in titi.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out titi.crt -d
ays 365
Certificate request self-signature ok
subject=CN = titi
```

```
axelle@securite-containeur:~$ kubectl config set-credentials titi \
> --client-certificate=titi.crt \
> --client-key=titi.key
User "titi" set.
```

```
axelle@securite-containeur:~$ kubectl config set-context titi-context \
> --cluster=kind-kind \
> --namespace=test-rbac \
> --user=titi
Context "titi-context" created.
```

```
axelle@securite-containeur:~$ kubectl config use-context titi-context
Switched to context "titi-context".
```

```
axelle@securite-containeur:~$ kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          32m
```

```
axelle@securite-containeur:~$ kubectl run test-pod --image=nginx
Error from server (Forbidden): pods is forbidden: User "titi" cannot create resource "pods" in API group "" in the nam
espace "test-rbac"
```

```
axelle@securite-containeur:~$ kubectl config use-context kind-kind
Switched to context "kind-kind".
```

Pour la seconde partie du TP, nous avons exploré le système RBAC (Role-Based Access Control) de Kubernetes. Nous avons d'abord créé un namespace dédié nommé "test-rbac" avec la commande *kubectl create ns test-rbac*. Nous y avons déployé un pod nginx simple pour nos tests d'accès.

Pour visualiser les logs de ce pod, nous avons utilisé la commande *kubectl logs nginx -n test-rbac*, qui nous a permis d'observer les journaux d'initialisation du serveur nginx.

Nous avons ensuite créé un rôle "pod-reader" limité au namespace test-rbac, avec les permissions "get" et "list" uniquement sur les ressources de type "pods". Ce rôle a été visualisé avec la commande *kubectl get role -n test-rbac* puis en détail avec *kubectl describe role pod-reader -n test-rbac*, confirmant les permissions accordées.

L'étape suivante a consisté à lier ce rôle à un utilisateur fictif nommé "titi" via un RoleBinding. Pour que cet utilisateur puisse effectivement accéder au cluster, nous avons généré des certificats client en utilisant les autorités de certification du cluster. Cela a impliqué l'extraction des fichiers CA depuis le nœud principal, puis la génération d'une clé privée et d'un certificat pour titi.

Après avoir configuré le contexte kubernetes pour cet utilisateur, nous avons switché vers le contexte "titi-context" pour tester les permissions. Comme prévu, l'utilisateur titi pouvait lister les pods dans le namespace test-rbac grâce à la permission "list" accordée, mais ne pouvait pas créer de nouveaux pods du fait de l'absence de la permission "create" dans son rôle. Cette limitation a été confirmée par le message d'erreur explicite lors de la tentative de création d'un

pod: "pods is forbidden: User "titi" cannot create resource "pods" in API group "" in the namespace "test-rbac"".

### 3. Scanner un Cluster Kubernetes avec Kube-Bench

```
axelle@securite-containeur:~$ cat <<EOF | kubectl apply -f -
apiVersion: batch/v1
kind: Job
metadata:
  name: kube-bench
spec:
  template:
    spec:
      hostPID: true
      containers:
        - name: kube-bench
          image: aquasec/kube-bench:latest
          command: ["kube-bench"]
          volumeMounts:
            - name: var-lib-kubelet
              mountPath: /var/lib/kubelet
              readOnly: true
            - name: etc-systemd
              mountPath: /etc/systemd
              readOnly: true
            - name: etc-kubernetes
              mountPath: /etc/kubernetes
              readOnly: true
      restartPolicy: Never
      volumes:
        - name: var-lib-kubelet
          hostPath:
            path: "/var/lib/kubelet"
        - name: etc-systemd
          hostPath:
            path: "/etc/systemd"
        - name: etc-kubernetes
          hostPath:
            path: "/etc/kubernetes"


EOF

job.batch/kube-bench created
```

```
axelle@securite-containeur:~$ kubectl wait --for=condition=complete job.batch/kuaxelle@securite-containeur:~$ kubectl wait --for=con
dition=complete job.batch/kube-bench --timeout=300s
job.batch/kube-bench condition met
```

```
axelle@securite-containeur:~$ kubectl logs job.batch/kube-bench
[INFO] 4 Worker Node Security Configuration
```

```
== Summary policies ==
0 checks PASS
6 checks FAIL
29 checks WARN
0 checks INFO

== Summary total ==
16 checks PASS
8 checks FAIL
35 checks WARN
0 checks INFO
```

Dans cette partie, nous avons utilisé kube-bench pour évaluer la conformité sécuritaire de notre cluster Kubernetes selon les recommandations du CIS (Center for Internet Security). Kube-bench est un outil qui vérifie si la configuration de Kubernetes suit les bonnes pratiques de sécurité.

Nous avons déployé kube-bench en tant que job Kubernetes à l'aide d'un fichier de configuration YAML. Ce job accède aux configurations des différents composants de Kubernetes pour vérifier leur sécurité.

Les résultats du scan ont montré un bilan mitigé de la sécurité de notre cluster. Sur l'ensemble des vérifications, nous avons obtenu 16 tests réussis (PASS), 8 tests échoués (FAIL) et 35 avertissements (WARN). Plus précisément, pour les politiques de sécurité, aucun test n'a été réussi, 6 ont échoué et 29 ont généré des avertissements.

Ces résultats indiquent que notre cluster Kind, bien que fonctionnel pour un environnement de développement, présente des lacunes significatives en matière de sécurité selon les normes CIS. Les principales préoccupations identifiées concernent les configurations par défaut trop permissives, l'absence de certaines restrictions de sécurité, et le manque d'audit logging approprié. Ces faiblesses devraient être adressées dans un environnement de production pour garantir la sécurité du cluster.

1. **Détection et alerte d'intrusions dans kubernetes avec l'outil Falco**

```
axelle@securite-containeur:~$ helm repo add falcosecurity https://falcosecurity.
github.io/charts
"falcosecurity" has been added to your repositories
axelle@securite-containeur:~$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "falcosecurity" chart repository
Update Complete. ⎈Happy Helming!⎈
```

```
axelle@securite-containeur:~$ kubectl create ns falco
namespace/falco created
```

```
axelle@securite-containeur:~$ helm -n falco install falco falcosecurity/falco --
set falcosidekick.enabled=true --set falcosidekick.webui.enabled=true
NAME: falco
LAST DEPLOYED: Sun Apr 27 14:39:09 2025
NAMESPACE: falco
STATUS: deployed
REVISION: 1
NOTES:
Falco agents are spinning up on each node in your cluster. After a few
seconds, they are going to start monitoring your containers looking for
security issues.


No further action should be required.
```

```
axelle@securite-containeur:~$ kubectl get pods -n falco
No resources found in falco namespace.
```

La dernière partie de notre TP concernait l'implémentation de Falco, un outil de détection d'intrusion pour Kubernetes. Nous avons commencé par ajouter le dépôt Helm de Falco puis créé un namespace dédié pour son déploiement.

L'installation de Falco a été réalisée via Helm avec l'activation de falcosidekick et de son interface web pour visualiser les alertes. Cependant, nous avons rencontré plusieurs obstacles qui ont empêché son fonctionnement optimal:

Le manque d'espace disque initial a été problématique pour le déploiement des composants de Falco, qui nécessitent des ressources significatives. Les conteneurs Falco étant assez volumineux, ils n'ont pas pu être correctement téléchargés et déployés sur notre environnement limité.

Des erreurs de connexion au cluster sont apparues lors des tentatives d'interaction entre les composants de Falco, particulièrement lors de la communication avec l'API Kubernetes. Ces erreurs ont empêché la bonne configuration des règles de détection.

Les pods Falco ne démarraient pas correctement, comme indiqué par la commande *kubectl get pods -n falco* qui ne retournait aucune ressource active. Malgré plusieurs tentatives de désinstallation et réinstallation, nous n'avons pas réussi à résoudre ce problème.

Pour tenter de résoudre ces problèmes, nous avons d'abord essayé de libérer de l'espace disque en supprimant des images Docker et des données temporaires. Nous avons également tenté de recréer le cluster dans sa totalité, espérant partir d'un environnement plus propre. Enfin, nous avons fait plusieurs tentatives d'installation de Falco avec différentes configurations, malheureusement sans succès.

Théoriquement, Falco aurait dû jouer un rôle crucial dans notre environnement Kubernetes en détectant les comportements anormaux au niveau du système et des conteneurs. Son fonctionnement repose sur l'analyse des appels système pour identifier des activités suspectes comme les exécutions de shell dans les conteneurs, les accès non autorisés aux fichiers sensibles, ou les tentatives d'escalade de privilèges.

Dans un environnement Kubernetes sécurisé, Falco complète parfaitement les mécanismes RBAC en offrant une couche de détection dynamique qui va au-delà des contrôles d'accès statiques. Cette complémentarité est essentielle pour une défense en profondeur, où les règles RBAC empêchent les accès non autorisés tandis que Falco détecte les comportements malveillants même lorsque l'accès initial est légitime.