Sécurité des infrastructures basées sur Kubernetes

Xavier Mehrenberger

Airbus Seclab – https://airbus-seclab.github.io/

Résumé. Cet article propose une vue d'ensemble de la sécurité des infrastructures basées sur Kubernetes, un orchestrateur de conteneurs. Kubernetes est présenté sans présupposer que le lecteur est déjà familier avec ce système. Les bonnes pratiques concernant la sécurisation d'un cluster, ainsi que les fonctionnalités de sécurité offertes par Kubernetes aux applications hébergées sont décrites.
Cet article est destiné à la fois aux utilisateurs et administrateurs de clusters Kubernetes, et aux lecteurs qui souhaitent évaluer la sécurité d'infrastructures existantes.

1 Introduction

Kubernetes est un orchestrateur de conteneurs. Sa grande popularité mérite que l'on s'intéresse à la sécurité des infrastructures utilisant ce système.

Cet article suivra le plan suivant :

— Une présentation rapide de Kubernetes : vue d'ensemble, ressources, machinerie interne ;
— Ensuite, des bonnes pratiques concernant la sécurisation d'un cluster Kubernetes ;
— Enfin, les fonctionnalités de sécurité offertes par Kubernetes aux applications hébergées par cet orchestrateur.

Les vulnérabilités logicielles présentes dans les composants de Kubernetes, ou plus largement dans les composants usuellement présents dans ce type d'infrastructures (voir [12]), habituellement rapidement corrigées, ne seront pas abordés.

Une police d'écriture `à chasse fixe` est utilisée dans cet article pour les concepts et composants de Kubernetes. Au moment de la rédaction de cet article, la dernière version publiée de Kubernetes est la v1.18 (23 avril 2020).

2 Kubernetes

Kubernetes, souvent écrit « k8s », est un orchestrateur facilitant le déploiement d'applications exécutées dans des conteneurs, eux-mêmes

exécutés sur une ou plusieurs machines appelées `Nodes`. Kubernetes permet par exemple :

- d'assurer que le nombre de conteneurs est en adéquation avec la charge à chaque instant ;
- de les relancer en cas de crash ;
- de faciliter les montées de versions progressives d'application (*canary deployment*), en testant les nouvelles versions sur une proportion croissante des utilisateurs tant qu'aucune erreur n'est détectée ;
- de faciliter le retour en arrière automatique (*rollback*).

Kubernetes est utilisé pour des applications déployées dans le cloud, mais également de plus en plus par des éditeurs pour fournir des applications *on-premises* qui s'appuient sur plusieurs (micro-)services, déployés sur une à quelques centaines de machines.

Les systèmes de construction de conteneurs permettent l'expression sous forme de code (ex. Dockerfile) de « recettes » de construction automatique d'une image contenant une application et l'ensemble de ses dépendances. Ils permettent d'éviter toute opération manuelle (ex. documentation d'installation suivie par un opérateur humain) et d'obtenir un résultat plus reproductible, au comportement plus déterministe.

Kubernetes prolonge cette idée à l'échelle d'une infrastructure : l'ensemble des composants devant être exécutés sur un *cluster* Kubernetes et leur configuration (ressources nécessaires, besoin d'isolation réseau, etc.) sont spécifiés par un ensemble de fichiers YAML. Ces fichiers décrivent l'état désiré et non pas les instructions permettant d'y arriver : on parle parfois d'infrastructure immutable. Il s'agit d'une forme d'*Infrastructure as Code*, avec une approche déclarative. Sur les infrastructures construites avec Kubernetes, on n'intervient plus interactivement sur les applications (en `ssh` par exemple) pour les mettre à jour ou modifier un fichier de configuration, mais on remplace un conteneur par une nouvelle version. L'infrastructure résultante sera alors plus *reproductible*, et se comportera de la même façon sur les environnements de développement et de production.

2.1 API : les ressources principales

Kubernetes définit un certain nombre de ressources, qui devront être déclarées pour déployer des applications. Une minorité de ressources ont une portée globale (ex. `Nodes`, `ClusterRoles`), mais la plupart appartiennent à un `namespace` (espace de noms). Le `namespace kube-system` est utilisé par les ressources appartenant à la machinerie interne du cluster.

Les principales ressources sont les suivantes :

Pod (cosse de pois, ou groupe de baleines en anglais) ensemble constitué d'un ou plusieurs conteneurs, s'exécutant sur la même machine, partageant le même *namespace* réseau, et ayant parfois des **Volumes** (répertoires) partagés entre eux. C'est l'unité de déploiement minimale. Les **Pods** ne sont presque jamais déclarés directement, on déclare plutôt des ressources comme **Job**, **CronJob**, **DaemonSet** ou **Deployment**, qui géreront le démarrage et l'arrêt des **Pods**.

Job, CronJob ressources spécifiant qu'un **Pod** doit être lancé à la demande, une fois seulement (**Job**) ou périodiquement (**CronJob**).

Deployment ressource spécifiant qu'un certain nombre d'instances d'un même **Pod** doivent s'exécuter.

DaemonSet ressource indiquant qu'une unique instance d'un **Pod** doit s'exécuter sur chaque **Node**, quel que soit le nombre de **Nodes**.

NetworkPolicy règles de pare-feu à appliquer entre **Pods**, ou entre **Pod** et services externes.

Role, RoleBinding ressources utilisées pour décrire les permissions fines (*authorization*) : les **Roles** décrivent un ensemble d'actions autorisées dans un **namespace**, et les **RoleBindings** lient un utilisateur, un groupe ou un compte de service à un **Role**.

ClusterRole, ClusterRoleBinding même chose que pour **Role** et **RoleBinding**, mais définissant des permissions sur le cluster entier, et non pas sur un seul **namespace**.

PersistentVolume espace de stockage persistent (non détruit lors de l'arrêt d'un **Pod**) mis à disposition du cluster, que les **Pods** pourront utiliser en déclarant des **PersistentVolumeClaim**.

ConfigMap stockage de tables de configuration clef-valeur, qui peuvent être mises à disposition des conteneurs sous forme de variables d'environnement, d'arguments de ligne de commande, ou d'un répertoire monté dans le conteneur dans lequel chaque valeur est stockée dans un fichier dont le nom est la clef.

Secret stockage de secrets de petite taille, comme une clef privée, un mot de passe ou un jeton d'authentification. Ils sont mis à disposition des conteneurs sous forme de fichiers.

Service décrit comment une application (constituée par un ensemble de **Pods**) est rendue accessible par le réseau. Concrètement, les flux réseau reçus par le cluster et à destination d'une IP « de service », d'un port donné, ou d'un *reverse proxy* (ex. fourni par l'opérateur de cloud) seront redirigés vers un port donné des **Pods** de l'application.

L'API de Kubernetes peut également être étendue par des « Opérateurs » (c'est un patron de conception), qui définissent de nouvelles ressources (`CustomResourceDefinition`) et apportent la machinerie capable de traiter ces extensions. Il existe par exemple un opérateur « MySQL » permettant de déclarer facilement la volonté d'utiliser un serveur de base de données.

On peut consulter la liste complète des ressources définies sur un cluster à l'aide de la commande `kubectl api-resources`.

2.2 Constitution de la machinerie de Kubernetes

Le cœur du cluster est l'`API Server` qui permet la consultation et la modification des ressources, présentées plus haut, qui constituent l'état du cluster. Cette API est utilisée par les administrateurs du cluster, par la machinerie du cluster, et également par les applications déployées sur le cluster.

Les données sont stockées par l'`API Server` sur un système de stockage clef-valeur distribué appelé `etcd`. Par défaut, les données stockées dans `etcd` ne sont pas chiffrées, y compris les `Secrets`. Une fonctionnalité (pour l'instant beta) permet d'activer divers mécanismes de chiffrement, y compris l'utilisation d'un HSM éventuellement fourni par l'opérateur de cloud.

D'autres composants servent à amener dans le cluster l'état décrit par les ressources définies, et ajoutent à ces ressources des informations sur l'état réel :

`scheduler` affecte les `Pods` aux `Nodes`.

`kubelet` s'exécute sur chaque `Node` du cluster, et assure que les `Pods` programmés par le `scheduler` s'y exécutent bien.

`proxy` assure l'acheminement du trafic destiné à une IP de `Service` vers les bons `Pods`.

`overlay network` (optionnel, apporté par un projet tiers) assure l'acheminement du trafic réseau entre `Nodes`, l'application des `NetworkPolicies`, et éventuellement le chiffrement du trafic.

3 Sécurisation d'un cluster

Cette partie traite de la sécurité du cluster en lui-même. Il s'agit de limiter la surface d'attaque exposée, et en particulier d'empêcher que la compromission par un attaquant d'une application hébergée par le cluster

n'aie des effets sur l'ensemble des autres applications, ou sur la machinerie du cluster.

3.1 Mode d'installation d'un cluster Kubernetes

Selon les besoins, il est possible de choisir de s'impliquer plus ou moins dans la gestion du cluster. On peut par exemple confier la gestion du matériel, l'installation et la maintenance du cluster y compris systèmes de stockage et *overlay network* à un tiers (ex. Google Kubernetes Engine (GKE) [14], Amazon Elastic Kubernetes Service (EKS) [10]), et n'utiliser que l'API Kubernetes. La sécurité du cluster sera dans ce cas la responsabilité du tiers, et il ne sera probablement pas possible d'auditer la machinerie du cluster en elle-même.

À l'opposé, on peut également choisir de prendre en charge tout ou partie de ces tâches. De nombreuses options sont disponibles pour faciliter cela, dont par exemple :

— Minikube [25] fournit un cluster sous forme de VM unique, pour le développement ;
— Kubespray [24] est un installateur basé sur Ansible [2], qui prend en charge l'installation sur plusieurs `Nodes`, les mises à jour, de nombreux `overlay networks` et applications utiles.

Une liste plus complète d'options est présentée dans la documentation de Kubernetes [29].

Il est nécessaire de bien identifier quelles fonctionnalités de sécurité sont prises en charge par l'hébergeur, par le système d'installation, et celles qui restent à la charge des administrateurs du cluster :

— configuration et mise à jour du système d'exploitation (voir 3.2) ;
— mise en place et sécurisation de l'accès au système de stockage exposé par Kubernetes sous forme de `PersistentVolume` ;
— mises à jour des composants du cluster : les versions mineures sont publiées environ tous les trois mois, et les versions dites *patch* le sont toutes les 3 à 4 semaines.

3.2 Durcissement du système d'exploitation

Les machines (Linux en général) sur lesquelles le cluster est installé doivent être raisonnablement durcies. Voir par exemple les recommandations de l'ANSSI [27] à ce sujet.

Une attention particulière devrait être apportée aux points suivants :

— recensement des personnes ayant des droits effectifs d'administrateurs sur les machines (ex. opérateur cloud, de l'infrastructure

de virtualisation, administrateurs, agents éventuellement imposés installés sur les machines, etc.)
— inventaire et application des correctifs : quels sont les composants logiciels installés ? Qui est responsable de quels composants ? À quelle fréquence ? Une veille sur les vulnérabilités est-elle organisée ?
— pare-feu sur chacun des nœuds du cluster : en particulier, restreindre l'accès aux ports exposés par la machinerie Kubernetes.

3.3 Gestion des secrets

Il est important de recenser les secrets utilisés par le cluster. On peut citer notamment :

— les secrets utilisés par le cluster, stockés principalement dans `etcd` ;
— les secrets d'authentification entre différents composants de Kubernetes (ex. clefs utilisées pour les communications TLS entre `API Server` et `etcd`) ;
— si le chiffrement *at rest* de `etcd` est configuré pour utiliser un mécanisme de chiffrement local [30] (`aescbc`, `secretbox`, etc.), alors les fichiers dans lesquels ces secrets sont stockés doivent être recensés (typiquement `/etc/kubernetes/secrets.conf`) ;
— si un système de gestion de clef (*Key Management Service* (KMS), voir [32]) est utilisé, alors les mécanismes de communication et d'authentification avec ce système doivent être étudiés et recensés. Ces services sont typiquement fournis par les hébergeurs cloud ;
— si une application de gestion des secrets est mise à disposition des applications hébergées (ex. HashiCorp Vault [28], fréquemment utilisé), alors les secrets et les mécanismes de communication entre le cluster et ce système doivent être recensés.

Lorsqu'un secret chiffré est recensé, il convient de bien identifier de quelle manière le cluster peut accéder en clair à ce secret ; cela permet d'identifier de nouveaux secrets qui auraient pu échapper au recensement, puis tirer la pelote jusqu'au bout...

Une fois les secrets recensés, il convient de s'assurer que ceux-ci seront accessibles uniquement aux applications hébergées par le cluster qui en ont besoin (c'est rare !).

Il convient de filtrer par des `NetworkPolicies` l'accès réseau à `etcd`, et aux autres interfaces d'administration du cluster auquel les applications hébergées n'ont pas besoin d'accéder.

Les accès à `etcd` par les `API Servers` devraient être protégés par authentification mutuelle TLS.

Il est également nécessaire de s'assurer que les secrets (ex. clefs secrètes utilisées pour TLS) et *sockets* de contrôle (ex. `/var/run/docker.sock`) stockés sur les `Nodes` ne sont pas rendus accessibles par le montage de fichiers locaux, soit en interdisant ce type de montage via une `PodSecurityPolicy` (voir 3.7), soit par une revue manuelle ou automatique des ressources déclarées par les applications hébergées.

3.4 Configuration de `API Server`

La configuration par défaut du composant `API Server` peut être durcie en production.

Authentification anonyme Par défaut, les connexions anonymes ne sont pas interdites sur les composants `kubelet` et `API Server`. L'utilisateur `system:anonymous` peut effectuer des requêtes, si le système d'autorisation les accepte. C'est le cas avec la valeur par défaut `AlwaysAllow`, rencontrée occasionnellement dans des clusters construits à la main (les installateurs comme kubespray incitent fortement à utiliser un autre modèle). Il est souhaitable de configurer ces composants pour que les authentifications anonymes soient refusées, en les exécutant avec le paramètre `--anonymous-auth=false`.

Permissions de `kubelet` Par défaut, `API Server` limite trop peu les modifications de ressources pouvant être effectuées par les agents `kubelet` qui s'exécutent sur chacun des `Nodes`. Il est souhaitable d'activer sur l'`API Server` la fonctionnalité `NodeRestriction` [33], permettant d'éviter que l'instance de `kubelet` qui tourne sur le `Node` « A » ne puisse modifier des informations (objets `Node`, `Pod`) concernant un `Node` « B ».

3.5 Activer la protection *seccomp* par défaut

Docker active par défaut une protection *seccomp* pour tous les conteneurs, qui limite les appels système que les applications peuvent utiliser, et limite ainsi la surface d'attaque du noyau Linux exposée aux applications. Il s'agit d'une mesure de défense en profondeur. Kubernetes est souvent configuré pour utiliser Docker pour gérer les conteneurs, mais *seccomp* n'est pas activé par défaut ; pour l'activer, il faut ajouter des annotations dans la définition des `Pods` spécifiant le profil *seccomp* à utiliser, pour l'ensemble du `Pod` ou pour un conteneur spécifique comme présenté sur le listing 1.

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    # utiliser le profil "hardened.json" pour
    # tous les conteneurs du Pod
    seccomp.security.alpha.kubernetes.io/pod: >
        "localhost/hardened.json"
    # utiliser le profil seccomp par défaut fourni
    # par docker pour le conteneur nommé "conteneur1"
    container.security.alpha.kubernetes.io/conteneur1: >
        "runtime/default"
  ...
```

Listing 1. Sélection de profils *seccomp* pour un `Pod` ou un conteneur

3.6 Accès aux services Kubernetes par les applications

Les services Kubernetes (ex. DNS, `API Server`, etc.) sont déployés dans le `namespace` réservé `kube-system`. Il est souhaitable de déployer des `NetworkPolicies` interdisant par défaut le trafic réseau vers `kube-system` provenant des applications hébergées dans d'autres `namespaces`, et écrire des `NetworkPolicies` pour les rares applications ayant besoin d'accéder à ces services.

Cela permet de réduire la surface d'attaque exposée par le cluster aux applications. Cette mesure de défense en profondeur aurait par exemple été utile pour empêcher l'exploitation d'une vulnérabilité découverte en 2018 [26].

3.7 Politique de sécurité à l'échelle du cluster avec `PodSecurityPolicy`

Le mécanisme de `PodSecurityPolicy` permet aux administrateurs d'un cluster d'imposer des politiques de sécurité sur l'ensemble des `Pods` exécutés sur le cluster. Ces politiques vont restreindre les ressources qu'il sera possible de définir : par exemple, on peut interdire la définition de `Pods` *privilégiés*, qui peuvent notamment accéder sans restrictions aux périphériques de l'hôte.

Ce mécanisme est particulièrement intéressant lorsque les utilisateurs du cluster (c'est-à-dire les personnes qui peuvent définir ou modifier une partie des ressources) ne sont pas nécessairement les administrateurs du cluster.

Ces politiques de sécurité permettent par exemple :

— d'interdire aux conteneurs d'accéder aux *namespaces* Linux de l'hôte ;
— d'interdire l'utilisation du compte `root` dans les conteneurs ;
— de contrôler les *capabilities* Linux pouvant être exigées dans la définition d'un `Pod` ;
— de choisir le profil *seccomp* appliqué par défaut ;
— etc.

L'application de ces politiques peut éviter bien des problèmes de configuration, permettant parfois de compromettre les `Nodes` à partir d'un conteneur contrôlé par un attaquant, par exemple suite à l'exploitation d'une vulnérabilité dans une application.

On pourra trouver la liste complète des politiques supportées dans la documentation sur ces politiques [31].

3.8 Revue des rôles définis

Il convient de réduire autant que possible les permissions accordées aux utilisateurs et applications interagissant avec la machinerie du cluster (`API Server` principalement). Pour cela, il faut effectuer une revue des mécanismes d'authentification configurés, et des permissions configurées.

Authentification Les utilisateurs d'un cluster Kubernetes sont répartis en deux catégories : les comptes de service (`ServiceAccount`) gérés par Kubernetes, et les utilisateurs et groupes gérés à l'extérieur du cluster.

Pour les comptes de service, l'authentification est basée sur des jetons, qui sont mis à disposition dans les conteneurs selon leur configuration, dans le dossier `/run/secrets/kubernetes.io/serviceaccount/token`. Un jeton par défaut est automatiquement monté (pour le désactiver, voir 4.2). On peut configurer un `Pod` pour utiliser un autre jeton avec la configuration présentée sur le listing 2.

```
apiVersion: v1
kind: Pod
...
spec:
  containers:
  - image: debian:buster
    name: mon-conteneur
    serviceAccountName: mon-compte-de-service
```

Listing 2. Utilisation d'un `ServiceAccount` différent de `default`

Pour les utilisateurs et groupes, plusieurs mécanismes existent, dont l'utilisation de certificats client X509 contenant le nom d'utilisateur et de

groupe. Le composant `API Server` acceptera les certificats signés par une CA qui lui est passée par le paramètre de lancement `--client-ca-file`.

Autorisation L'accès aux APIs exposées par `API Server` est contrôlé par une politique RBAC (Role-Based Access Control). Il est possible de définir des rôles (ressources `Role`, `ClusterRole`), autorisés à accéder à une liste de chemins (*endpoints*) et d'actions possible sur chaque chemin (*verbs* HTTP : `get`, `list`, `watch`, `update`, `delete`, etc.).

Les rôles sont affectés à des `ServiceAccounts`, des utilisateurs ou des groupes par la déclaration de `RoleBinding` ou `ClusterRoleBinding`.

Pour vérifier les permissions configurées, il faut donc examiner ces ressources, en utilisant par exemple les commandes présentées sur le listing 3.

```
$ kubectl get serviceaccounts --all-namespaces -o yaml
$ kubectl get clusterroles --all-namespaces -o yaml
$ kubectl get roles --all-namespaces -o yaml
$ kubectl get clusterrolebindings --all-namespaces -o yaml
$ kubectl get rolebindings --all-namespaces -o yaml
```

Listing 3. Énumération des permissions définies

Certains rôles ont des privilèges élevés ; ils peuvent avoir été créés par des applications tierces installées sur le cluster, qui disposent elles-mêmes de privilèges élevés.

Certains groupes sont automatiquement attribués aux utilisateurs :

— Le groupe `system:authenticated` est attribué aux utilisateurs authentifiés ;

— Le groupe `system:unauthenticated` est attribué aux utilisateurs non authentifiés.

Exemple concret Par exemple, l'application `Helm` [20] est un « gestionnaire de packages » pour Kubernetes, qui s'exécute typiquement sur un cluster, et permet d'y installer d'autres applications à partir de *templates* (appelés *charts*) mis à disposition par des tiers (de manière similaire aux rôles Ansible partagés sur Ansible Galaxy [11]). `Helm` dispose donc des droits permettant d'installer de nouvelles ressources et manipuler les ressources existantes, et donc au final d'exécuter du code arbitraire sur le cluster.

L'existence d'une instance de `Helm` sur un cluster, ou d'une application ayant des privilèges similaires, apparaîtra lors d'une revue des comptes et privilèges définis. On pourra ensuite étudier de plus près l'application `Helm`,

pour déterminer sous quelles conditions un attaquant pourrait ordonner le déploiement d'une application malveillante. En version 2, `Helm` disposait de son propre système d'authentification et autorisation, dont la configuration doit être revue afin de s'assurer qu'un attaquant ne pourra pas utiliser `Helm` pour prendre le contrôle du cluster Kubernetes.

De manière générale, les systèmes d'intégration et déploiement continu (CI/CD) parfois installés sur les *clusters* disposent également souvent de droits importants.

4 Sécurisation des applications hébergées sur le cluster

Cette partie traite de la sécurisation des applications hébergées par un cluster, et des fonctionnalités de sécurité proposées par Kubernetes aux applications.

4.1 Filtrage réseau avec les NetworkPolicies

Il est possible de définir des `NetworkPolicies`, décrivant au niveau transport (ports TCP, UDP) les flux réseau autorisés. Les sources et destinations peuvent être spécifiées par leur adresse IP (ex. pour les services externes), ou par les *labels* affectés aux `Pods`. Il est ainsi possible de déclarer l'intention « le trafic réseau sur le port TCP/3306 doit être autorisé des `Pods` portant le label `uses-mysql: true` (ex. un serveur web) vers les `Pods` portant le label `k8s-app: mysql` », sans devoir préciser d'adresse IP, ou se soucier du nombre ou de la localisation de ces `Pods` dans le cluster. Pour mettre en œuvre cet exemple, on peut par exemple écrire la `NetworkPolicy` présentée sur le listing 4.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: api-allow-mysql-ingress
spec:
  podSelector:         # Cette politique s'applique à tous
    matchLabels:       # les Pods ayant le label
      k8s-app: mysql   # k8s-app: mysql (ex. serveur MySQL)
  ingress:  # cette politique ne concerne
  - ports:  # que les flux entrants
    - port: 3306
      protocol: TCP  # optionnel
    from:                    # le trafic provenant
    - podSelector:           # des Pods ayant le label
        matchLabels:         # uses-mysql: true sera
          uses-mysql: true   # autorisé par cette politique
```

Listing 4. Définition de `NetworkPolicy` pour un serveur web

Si on a choisi d'interdire tous les flux sortants qui ne sont pas explicitement autorisés, il faudra également appliquer une politique de filtrage des flux sortants (type « Egress ») similaire.

Un dépôt github [13] propose une collection d'exemples de cas d'usages bien documentés.

L'application de ces `NetworkPolicies` n'est pas assurée directement par Kubernetes, mais en général par l'`overlay network` utilisé. Il faut s'assurer qu'un `overlay network` est bien utilisé, et qu'il supporte les `NetworkPolicies`. C'est le cas dans la plupart des environnements *cloud*, ou par exemple avec les `overlay networks` open source *calico* [3] ou *cilium* [4], mais pas *flannel* [16].

4.2 Comptes de service montés dans les `Pods`

Par défaut, Kubernetes crée un jeton nommé `default` pour chaque `namespace`, et le rend accessible via le système de fichiers dans chaque conteneur appartenant à un `Pod` qui appartient à ce `namespace`. Ce jeton donne l'accès aux APIs autorisées pour les `Roles` et `ClusterRoles` applicables au jeton. Un `RoleBinding` donne le rôle `system:basic-user` à tous les utilisateurs authentifiés, dont `default`. Ce rôle dispose de très peu de permissions.

Cependant, ne pas monter automatiquement le jeton en utilisant l'option `automountServiceAccountToken: false` permet de réduire la surface d'attaque du cluster exposée aux applications qui n'ont pas besoin d'interagir avec `API Server`.

4.3 Gestion des images docker

Les applications hébergées par un cluster Kubernetes sont exécutées à partir d'images au format standardisé, développé initialement par Docker (voir [15]). Ces images sont typiquement construites à partir d'une recette appelée `Dockerfile` (voir [9]), dont un exemple est présenté sur le listing 5.

```
# Cette nouvelle image est construite
# à partir de l'image publique debian:buster
FROM debian:buster
# Créer le répertoire /install et travailler dedans
WORKDIR /install
# Installer le package wget
RUN apt-get update && apt-get install wget -y
# Télécharger le binaire du webshell gotty
RUN wget 'https://github.com/yudai/gotty/releases/download/'\
'v1.0.1/gotty_linux_amd64.tar.gz'
# Extraire le binaire
```

```
RUN tar -xf gotty_linux_amd64.tar.gz
# Définir la commande à executer
CMD /install/gotty /bin/bash
```

Listing 5. Exemple de `Dockerfile`

Cette image installe le *webshell* Gotty [17], une variante du traditionnel bind shell `nc -lnvp 8080 -e /bin/bash`. Elle est fort pratique pendant une mission d'audit, lorsqu'une erreur de configuration dans un cluster Kubernetes cible permet de créer un `Pod` arbitraire : cela facilite la reconnaissance du terrain.

La construction de ces images et leur acheminement vers le cluster doivent être sécurisés, pour éviter qu'un attaquant ne puisse télécharger des images (le stockage de secrets dans les images est déconseillé, mais est parfois rencontré en pratique), ou alors modifier une image et ainsi exécuter son propre code au lieu du code légitime sur le cluster qui utilisera son image malveillante.

Construction et vérification des images Les images sont quasi-systématiquement construites à partir d'images de base publiques. Il est préférable de partir d'images de confiance, qui n'embarquent pas de composants logiciels obsolètes ou vulnérables. Par exemples, les images marquées *Official Image* sur le dépôt public Docker Hub [6].

Il est également souhaitable de mettre en place dans son *pipeline* d'intégration continue une analyse automatique des vulnérabilités présentes dans les image Docker que l'on construit, par exemple avec les outils open source Clair [5] ou Anchore Engine [1].

Une fois construites, il est également fortement recommandé de signer les images en utilisant par exemple Docker Notary [7], pour éviter qu'elles ne soient modifiées par un attaquant.

Acheminement et stockage des images L'acheminement des images Docker est souvent un point faible dans les infrastructures Kubernetes : il est très courant de trouver un dépôt d'images Docker Registry [8] avec une configuration par défaut, hébergé sur l'infrastructure de production, configuré pour transférer les images sans chiffrement (pas de TLS !), sans gestion des permissions d'écriture. Les images sont également rarement signées, ce qui n'arrange rien.

Un attaquant ayant un accès réseau à ce dépôt d'images (parfois exposé directement, parfois via un rebond suite à une erreur de configuration dans le cluster) peut alors s'en donner à cœur joie, et remplacer les images par

des versions backdoorées par ses soins, et attendre qu'elles soient utilisées pour prendre le contrôle de l'application voire du cluster.

Pour éviter cela, il est crucial de renforcer l'acheminement des images.

— Le dépôt d'images doit avoir une gestion fine des permissions, en particulier dans le cas où le cluster et le dépôt ont des utilisateurs ayant des permissions différentes (ex. plusieurs équipes, *multi-tenant*). En particulier, les permissions d'écriture doivent être restreintes aux seuls systèmes légitimes pour ces opérations : systèmes d'intégration continue (CI) et administrateurs. En aucun cas une application « lambda » déployée sur le cluster ne doit avoir les droits de faire cette opération.
— L'utilisation de TLS pour les communications avec le dépôt d'images doit être rendue obligatoire.
— Si le cluster est utilisé par plusieurs clients, il est possible de protéger les opérations en lecture sur le dépôt par des secrets différents pour chaque client. Dans la définition des `Pods`, il faudra référencer ce secret via l'option de configuration `imagePullSecrets`).
— Il convient de configurer le cluster pour n'autoriser que les images signées par des utilisateurs de confiance, et de protéger correctement les clefs de signature.

L'utilisation du dépôt d'image open source Harbor [19] permet de régler une partie des problèmes, grâce au support de permissions fines, et à la possibilité de rejeter les images non signées.

4.4 Sécurité des communications réseau

Souvent, les applications déployées utilisent des communications non chiffrées ; elles sont accessibles depuis l'extérieur du cluster via des *reverse proxys* assurant le chiffrement, qui peuvent prendre plusieurs formes :

— un *reverse proxy* géré par le fournisseur cloud (mis en place suite à la création d'une ressource `Ingress`).
— un conteneur déployé dans chaque `Pod` de l'application, contenant un *reverse proxy* comme nginx écoutant en HTTPS, et retransmettant les requêtes vers le conteneur de l'application en HTTP. Ce *reverse proxy* nginx pourra alors être rendu accessible par la définition d'un `Service`.

Il convient de recenser les flux réseau en clair, les chemins de passage de ces flux, et de s'assurer qu'ils sont « de confiance » pour l'application considérée.

Une revue des ressources de type `Ingress`, `Service`, `Endpoints` et `EndpointSlices` permettra de vérifier quels services réseau sont accessibles depuis l'extérieur du cluster.

4.5 Gestion des logs

Comme sur une infrastructure classique, il est souvent important de collecter et centraliser les logs de fonctionnement des applications hébergées par le cluster. Plusieurs techniques sont possibles pour collecter les logs de chacun des conteneurs :

- Pour les applications capables d'écrire leur logs sur les sorties `stdout` et `stderr`, il n'y a rien à faire : Kubernetes collectera automatiquement ces logs.
- Si l'application est capable d'écrire des logs vers un fichier local, il est alors possible d'adjoindre dans ses `Pods` un conteneur supplémentaire (souvent appelé « sidecar »), qui lira les fichiers écrits dans un répertoire partagé entre le conteneur de l'application et celui du « sidecar » et les écrira sur ses propres sorties `stdout` et `stderr`.
- Certaines applications ne jurent que par `syslog`, et refusent d'écrire leurs logs ailleurs que dans un *socket* UNIX situé dans `/dev/log`, emplacement codé en dur dans sa `libc`. Malheureusement, il n'est pas possible pour un conteneur non root d'écouter sur ce *socket*... il faut alors ruser et patcher `libc.so.6` avec `sed^w^w` en recompilant la `libc` pour qu'elle utilise un autre chemin (ex. `/shared/logsocket`) partagé entre le conteneur faisant tourner l'application récalcitrante et le conteneur faisant tourner un serveur syslog.

Une fois les logs collectés par Kubernetes, ils sont accessibles dans le dossier `/var/log/containers/` au format JSON. Votre solution préférée de centralisation de logs pourra alors les envoyer vers un serveur de stockage, par exemple en lançant un `DaemonSet` (c'est-à-dire un `Pod` sur chaque `Node`) faisant tourner un agent de collecte `rsyslog`, `syslog-ng`, ou `logstash`.

4.6 Quotas et limitations de ressources

Sur un cluster Kubernetes, en général, de nombreuses applications s'exécutent en parallèle. Pour s'assurer qu'une application trop gourmande ou dysfonctionnelle n'utilise trop de mémoire ou de temps CPU, il est possible de configurer des limites sur chaque conteneur. Sous Linux, le

mécanisme de `cgroups` (*control groups*) assurera le respect de ces limites. Par exemple, pour empêcher une application d'utiliser plus de 2 Gio de RAM et plus de 2 cœurs CPU, on pourra utiliser la configuration présentée sur le listing 6.

```
apiVersion: v1
kind: Pod
...
spec:
  containers:
  - image: debian:buster
    name: mon-conteneur
    resources:
      limits:
        memory: 2Gi
        cpu: 5
```

Listing 6. Configuration d'une limite de mémoire et CPU sur un `Pod`

4.7 Affectation `Pods`/`Nodes`

Pour des raisons de performance ou de sécurité, il est possible d'exprimer des contraintes que le `scheduler` devra respecter :

- Exécuter certains `Pods` sur une liste restreinte de `Nodes`, par exemple pour exécuter la machinerie Kubernetes sur des `Nodes` dédiés ;
- Colocaliser certains `Pods` sur le même `Node` pour des raisons de performance (ex. un serveur web et son cache applicatif) ;
- Imposer que deux groupes de `Pods` s'exécutent sur des `Nodes` différents, pour éviter les attaques par canaux cachés sur le CPU dans le cas où les deux groupes de `Pods` sont contrôlés par des clients différents (*multi-tenancy*).

5 Conclusion

L'utilisation de Kubernetes dans une infrastructure apporte une complexité importante, qui augmente avec le nombre de composants de l'écosystème *cloud-native* [12] utilisés.

Une fois ces composants bien compris et sécurisés, Kubernetes apporte également des fonctionnalités de sécurité très intéressantes, comme l'utilisation d'une infrastructure immutable, de règles *seccomp*, et la déclaration simple de règles de pare-feu qui seront appliquées sur l'ensemble du cluster.

Il est possible de réduire encore la surface d'attaque exposée, en remplaçant le *runtime* utilisant des conteneurs par des alternatives :

— `gVisor` [18], sandbox qui ré-implémente en userland (et en Go) une bonne partie de l'interface du noyau ;
— `kata` [21], qui remplace les conteneurs par des machines virtuelles.

Il existe également un nombre grandissant d'outils d'analyse automatique de la sécurité d'un cluster, comme `kubesec` [23] qui analyse les fichiers YAML de déclaration des ressources, ou encore `kube-bench` [22] qui vérifie la configuration du cluster pour identifier les bonnes pratiques de sécurité qui pourraient être mises en œuvre.

Références

1. Anchore engine. `https://github.com/anchore/anchore-engine`.
2. Ansible. `https://www.ansible.org/`.
3. Calico. `https://www.projectcalico.org/`.
4. Cilium. `https://cilium.io/`.
5. Clair, *Vulnerability Static Analysis for Containers*. `https://github.com/quay/clair`.
6. Docker hub. `https://hub.docker.com/`.
7. Docker notary : signature d'images docker. `https://docs.docker.com/notary/getting_started/`.
8. Docker registry : dépôt d'images docker. `https://docs.docker.com/registry/`.
9. Documentation du format `Dockerfile`. `https://docs.docker.com/engine/reference/builder/`.
10. *Amazon Elastic Kubernetes Service (EKS)*. `https://aws.amazon.com/fr/eks/`.
11. *Ansible galaxy*. `https://galaxy.ansible.com/`.
12. *CNCF Cloud Native Interactive Landscape*. `https://landscape.cncf.io/`.
13. *Example Kubernetes Network Policies*. `https://github.com/ahmetb/kubernetes-network-policy-recipes`.
14. *Google Kubernetes Engine (GKE)*. `https://cloud.google.com/kubernetes-engine`.
15. *Open Container Initiative* : spécification d'un format d'images standard et de *runtimes*. `https://www.opencontainers.org/`.
16. Flannel. `https://github.com/coreos/flannel`.
17. Gotty, webshell écrit en golang. `https://github.com/yudai/gotty`.
18. gVisor : sandbox réimplémentant en Go une bonne partie de l'API noyau Linux. `https://github.com/google/gvisor`.
19. Harbor. `https://goharbor.io/`.
20. *`Helm`, the package manager for Kubernetes*. `https://helm.sh/`.
21. Kata containers : *runtime* alternatif utilisant des machines virtuelles au lieu de conteneurs. `https://katacontainers.io/`.
22. Kube-bench : vérification automatique de l'application de bonnes pratiques de sécurité sur un cluster. `https://github.com/aquasecurity/kube-bench`.

23. Kubesec : analyse de fichiers `YAML` déclarant les ressources utilisées sur un cluster. `https://kubesec.io/`.
24. Kubespray. `https://github.com/kubernetes-sigs/kubespray`.
25. Minikube. `https://minikube.sigs.k8s.io/docs/`.
26. Vulnérabilité CVE-2018-1002105. `https://github.com/kubernetes/kubernetes/issues/71411`.
27. ANSSI. Recommandations de sécurité relatives à un système GNU/Linux. `https://www.ssi.gouv.fr/uploads/2016/01/linux_configuration-fr-v1.2.pdf`.
28. HashiCorp. Vault. `https://www.vaultproject.io/`.
29. Documentation Kubernetes. Choix de solution pour l'installation (local, hébergé...).
30. Documentation Kubernetes. *Encrypting Secret Data at Rest*.
31. Documentation Kubernetes. *Pod Security Policies*.
32. Documentation Kubernetes. *Using a KMS provider for data encryption*.
33. Documentation Kubernetes. Utilisation de `NodeRestriction` pour restreindre les droits de l'agent local (`kubelet`).