SÉCURITÉ DES CONTENEURS

SCANNING, RUNTIME, HARDENING

SCANNING
Détecter les vulnérabilités avant l'exécution

RUNTIME
Protéger les conteneurs en temps réel

HARDENING
Renforcer les images, les configurations et les environnements

BONNES PRATIQUES
Guides et recommandations actionnables

OUTILS & LABS
Outils open source et laboratoires pratiques

POLITIQUES
Contrôles et politiques de sécurité efficaces

INTÉGRATION DEVSECOPS
Sécurité intégrée dans le cycle CI/CD

CONFORMITÉ
CIS Benchmark, NIST, ISO 27001 et plus

Ayi NEDJIMI

PENSER COMME UN ATTAQUANT. AGIR COMME UN PROFESSIONNEL.

SÉCURITÉ DES CONTENEURS : SCANNING, RUNTIME, HARDENING
Ayi NEDJIMI

Contents

Sécurité des Conteneurs : Scanning, Runtime, Hardening

La conteneurisation a profondément transformé le développement et le déploiement logiciel, avec Docker comptant aujourd'hui plus de 13 millions d'utilisateurs actifs et Kubernetes devenu le standard de facto pour l'orchestration de conteneurs en production. Cette adoption massive s'accompagne d'une surface d'attaque considérablement élargie : images vulnérables provenant de registres publics non maîtrisés, mauvaises configurations de runtime, privilèges excessifs accordés aux conteneurs, chaînes d'approvisionnement logicielle compromises, et prolifération de secrets dans les layers d'images. Les incidents de sécurité liés aux conteneurs se multiplient : en 2023, l'attaque SolarWinds-like ciblant des images Docker Hub malveillantes a touché plus de 17 000 projets, et les ransomwares ciblant Kubernetes se sont multipliés avec des vecteurs d'entrée exploitant des APIs non authentifiées (CVE-2023-2728, CVE-2024-21626). Ce guide technique couvre l'intégralité du cycle de vie de la sécurité des conteneurs : du scanning d'images à la sécurité runtime, en passant par le hardening des configurations, la gestion de la supply chain et la conformité Kubernetes Pod Security Standards.

Points clés : La sécurité des conteneurs s'opère à quatre niveaux distincts qui doivent tous être adressés : (1) la sécurité de l'image (scanning, provenance, signing), (2) la sécurité du registre (contrôle d'accès, scanning continu), (3) la sécurité du runtime (confinement, détection d'anomalies), et (4) la sécurité de l'orchestrateur (RBAC Kubernetes, Pod Security). Négliger l'un de ces niveaux crée des angles morts exploitables.

1. Anatomie d'une image Docker et surface d'attaque

Une image Docker est constituée de couches (layers) immuables empilées les unes sur les autres. Chaque instruction Dockerfile crée un nouveau layer. Cette architecture a des implications de sécurité directes : les secrets accidentellement inclus dans un layer intermédiaire restent accessibles même s'ils sont supprimés dans un layer ultérieur.

```
# Inspecter les layers d'une image avec dive
# https://github.com/wagoodman/dive
dive nginx:latest

# Alternative : docker history
# ... (extrait — voir documentation officielle)
```

2. Trivy : scanner de vulnérabilités de référence

Trivy est le scanner de sécurité open-source le plus complet pour les conteneurs, développé par Aqua Security. Il analyse les vulnérabilités OS, les dépendances applicatives, les mauvaises configurations et les secrets.

```
# Installation Trivy
curl -sfL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh | sh -s -- -b
  ↪  /usr/local/bin v0.50.0

# Scan basique d'une image
trivy image nginx:latest
# ... (extrait — voir documentation officielle)
```

3. Grype : alternative performante pour le scanning

Grype, développé par Anchore, offre une base de données de vulnérabilités différente de Trivy (Grype utilise principalement NVD + GitHub Advisory Database + plusieurs sources spécifiques aux écosystèmes). L'utilisation combinée des deux outils améliore la couverture.

```
# Installation Grype
curl -sSfL https://raw.githubusercontent.com/anchore/grype/main/install.sh | sh -s -- -b
 ↪  /usr/local/bin

# Scan basique
grype nginx:latest
# ... (extrait — voir documentation officielle)
```

4. Snyk Container : scanning SaaS avec remédiation

Snyk Container se distingue par sa capacité à suggérer des images de base alternatives avec moins de vulnérabilités et à prioriser les CVEs exploitables en contexte réel.

```
# Installation Snyk CLI
npm install -g snyk

# Authentification
snyk auth
# ... (extrait — voir documentation officielle)
```

5. Docker Bench for Security : hardening de l'hôte

Docker Bench for Security est un script d'audit automatique qui vérifie des dizaines de bonnes pratiques de sécurité Docker selon les recommandations CIS Docker Benchmark.

```
# Exécution de Docker Bench
git clone https://github.com/docker/docker-bench-security.git
cd docker-bench-security
sudo bash docker-bench-security.sh

# ... (extrait — voir documentation officielle)
```

6. Images distroless : réduire la surface d'attaque

Les images distroless, maintenues par Google, ne contiennent que le runtime de l'application et ses dépendances directes — sans shell, sans gestionnaire de paquets, sans binaires système. Cette approche réduit drastiquement la surface d'attaque.

```
# Dockerfile multi-stage avec image distroless finale
# Stage 1 : Build
FROM golang:1.22-alpine AS builder
```

```
WORKDIR /app
# ... (extrait — voir documentation officielle)
```

7. Conteneurs rootless : isolation renforcée

Les conteneurs rootless permettent d'exécuter le daemon Docker ou Podman sans privilèges root sur l'hôte. En cas d'évasion du conteneur (*container escape*), l'attaquant obtient les droits d'un utilisateur non-privilégié sur l'hôte, et non root.

```
# Configuration de Docker en mode rootless
# Prérequis : uid_map et gid_map dans /proc/self/uid_map
dockerd-rootless-setuptool.sh install

# Vérification
# ... (extrait — voir documentation officielle)
```

8. Seccomp : filtrage des appels système

Seccomp (Secure Computing Mode) permet de restreindre les appels système (*syscalls*) disponibles pour un conteneur. Un conteneur standard Docker dispose de ~300 syscalls sur les ~400 disponibles dans le noyau Linux — seccomp permet de réduire ce nombre aux seuls syscalls nécessaires à l'application.

```
// Profil seccomp personnalisé pour un serveur web (nginx)
// /etc/docker/seccomp/nginx-profile.json
{
  "defaultAction": "SCMP_ACT_ERRNO",
  "architectures": ["SCMP_ARCH_X86_64", "SCMP_ARCH_AARCH64"],
# ... (extrait — voir documentation officielle)
```

```
# Appliquer le profil seccomp
docker run --security-opt seccomp=/etc/docker/seccomp/nginx-profile.json nginx:latest

# Identifier les syscalls nécessaires avec strace
strace -c -f nginx -g 2>&1 | head -30
# ... (extrait — voir documentation officielle)
```

9. AppArmor : profils MAC pour conteneurs

AppArmor (Application Armor) est un module de sécurité Linux (LSM) qui implémente le Mandatory Access Control (MAC) via des profils définissant les ressources accessibles à chaque programme.

```
# Vérifier qu'AppArmor est actif
aa-status | head -10

# Profil AppArmor pour un conteneur Docker nginx
cat > /etc/apparmor.d/docker-nginx << 'EOF'
# ... (extrait — voir documentation officielle)
```

10. Falco : détection d'anomalies runtime

Falco est le standard de facto pour la détection d'anomalies runtime dans les environnements Kubernetes et Docker. Développé par Sysdig et maintenu par la CNCF, il utilise eBPF (ou des modules noyau) pour intercepter les appels système et les événements Kubernetes.

```
# Installation Falco avec Helm sur Kubernetes
helm repo add falcosecurity https://falcosecurity.github.io/charts
helm repo update

helm install falco falcosecurity/falco \\
# ... (extrait — voir documentation officielle)
```

```
# /etc/falco/rules.d/custom-rules.yaml

# Règle 1 : Détecter un shell dans un conteneur (conteneurs distroless ne devraient jamais avoir de
 ↪  shell)
- rule: Shell spawned in container
  desc: Un shell a été lancé dans un conteneur — possible intrusion
# ... (extrait — voir documentation officielle)
```

11. OPA/Gatekeeper : politiques Kubernetes as Code

OPA Gatekeeper (Open Policy Agent) est un contrôleur d'admission Kubernetes qui évalue les politiques Rego avant de permettre la création ou la modification de ressources. Il constitue la couche de prévention, tandis que Falco est la couche de détection.

```
# Installation OPA Gatekeeper
kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/gatekeeper/release-
 ↪  3.14/deploy/gatekeeper.yaml

# Attendre que Gatekeeper soit prêt
kubectl wait --for=condition=Ready pods --all -n gatekeeper-system --timeout=120s
```

```
---
# ConstraintTemplate : interdire les conteneurs root
apiVersion: templates.gatekeeper.sh/v1
kind: ConstraintTemplate
metadata:
# ... (extrait — voir documentation officielle)
```

12. Kubernetes Pod Security Standards

Les Pod Security Standards (PSS) ont remplacé Pod Security Policy (PSP, déprécié en 1.21, supprimé en 1.25) dans Kubernetes. Ils définissent trois niveaux de sécurité appliqués via l'admission controller *Pod Security Admission*.

Niveau	Description	Usage typique	Restrictions clés
Privileged	Aucune restriction	Outils système (CNI, CSI)	Aucune
Baseline	Restrictions minimales	Applications génériques	Pas de privileged, pas de hostPath sensible
Restricted	Hardening maximal	Applications critiques	Non-root, no capabilities, seccomp RuntimeDefault

```
# Appliquer le niveau Restricted sur un namespace
kubectl label namespace production \\
  pod-security.kubernetes.io/enforce=restricted \\
  pod-security.kubernetes.io/enforce-version=v1.28 \\
  pod-security.kubernetes.io/warn=restricted \\
# ... (extrait — voir documentation officielle)
```

```
# Pod Security Context complet — niveau Restricted compatible
apiVersion: v1
kind: Pod
metadata:
  name: secure-pod
# ... (extrait — voir documentation officielle)
```

13. Image signing avec Cosign et Notary

Cosign, développé par Sigstore/Linux Foundation, est devenu le standard pour la signature cryptographique d'images de conteneurs. Il s'intègre nativement avec les registres OCI et permet une vérification transparente via le journal de transparence Rekor.

```
# Installation Cosign
curl -O -L "https://github.com/sigstore/cosign/releases/latest/download/cosign-linux-amd64"
sudo install cosign-linux-amd64 /usr/local/bin/cosign

# Génération d'une paire de clés pour la signature
# ... (extrait — voir documentation officielle)
```

14. SBOM : Software Bill of Materials pour les conteneurs

Un SBOM (Software Bill of Materials) est un inventaire exhaustif de tous les composants logiciels d'une image de conteneur, incluant leurs versions, licences et vulnérabilités connues. Il est devenu obligatoire pour les logiciels vendus au gouvernement américain (Executive Order 14028, mai 2021) et recommandé par l'ANSSI.

```
# Générer un SBOM avec Syft (par Anchore)
pip install syft  # ou via package manager

# Format SPDX (standard ISO/IEC 5962:2021)
syft nginx:latest -o spdx-json > nginx-sbom.spdx.json
# ... (extrait — voir documentation officielle)
```

15. Registres privés : sécurisation et contrôle d'accès

```
# Configuration Harbor (registre privé enterprise)
# Harbor = registre OCI avec scanning Trivy intégré, RBAC, audit logs

# Installation avec Helm
helm repo add harbor https://helm.goharbor.io
# ... (extrait — voir documentation officielle)
```

16. Audit de la supply chain : détecter les images malveillantes

```
# Détecter les backdoors dans les images Docker Hub
# Analyser les layers avec whaler
go install github.com/P3GLEG/Whaler@latest
whaler nginx:latest

# ... (extrait — voir documentation officielle)
```

Supply chain sécurité : Ne jamais utiliser d'images Docker Hub non officielles comme image de base de production. Préférer les images officielles (docker.io/library/) ou les images de l'éditeur vérifiées. Épingler les images par digest SHA256 plutôt que par tag (le tag "latest" peut être modifié). Exemple : `nginx@sha256:abc123...` au lieu de `nginx:latest`.

17. Kubernetes RBAC : principe du moindre privilège

```
# RBAC minimal pour un service applicatif
---
apiVersion: v1
kind: ServiceAccount
metadata:
# ... (extrait — voir documentation officielle)
```

```
# Audit des permissions Kubernetes avec kubectl-who-can
kubectl-who-can create pods --namespace production
kubectl-who-can exec pods --namespace production

# Outil rakkess : matrice de permissions RBAC
# ... (extrait — voir documentation officielle)
```

18. Network Policies Kubernetes : microsegmentation

```
# Network Policy : deny-all par défaut, puis autorisation sélective
---
# Bloquer TOUT le trafic entrant et sortant par défaut
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
# ... (extrait — voir documentation officielle)
```

19. Monitoring et observabilité de la sécurité

```
# Stack de monitoring sécurité pour conteneurs
# Falco + Falcosidekick + Prometheus + Grafana

# Métriques Falco exposées pour Prometheus
# falco_events_total{priority="CRITICAL", rule="Shell spawned in container"}
# ... (extrait — voir documentation officielle)
```

20. Comparaison des outils de sécurité conteneurs

Outil	Type	Licence	Intégration CI/CD	Runtime	Kubernetes
Trivy	Scanner image	Apache 2.0	Excellente	Non	Oui (plugin)
Grype	Scanner image	Apache 2.0	Bonne	Non	Non
Snyk Container	Scanner SaaS	Freemium	Excellente	Non	Oui
Falco	Détection runtime	Apache 2.0	Non	Oui (eBPF)	Oui (natif)
OPA/Gatekeeper	Admission control	Apache 2.0	Non	Non	Oui (natif)
Kyverno	Admission control	Apache 2.0	Non	Non	Oui (natif)
Cosign	Image signing	Apache 2.0	Excellente	Non	Via Kyverno
Docker Bench	Audit hôte	Apache 2.0	Bonne	Non	Non
Aqua Security	Suite complète	Commerciale	Excellente	Oui	Oui
Sysdig Secure	Suite complète	Commerciale	Excellente	Oui	Oui

Architecture de sécurité recommandée : Couche 1 — Scanner d'image (Trivy) en CI/CD avec seuil CRITICAL bloquant. Couche 2 — Admission control (OPA Gatekeeper ou Kyverno) pour les politiques Kubernetes. Couche 3 — Runtime detection (Falco avec eBPF) pour les anomalies en production. Couche 4 — Image signing (Cosign) et SBOM pour la traçabilité supply chain. Cette défense en profondeur couvre le cycle de vie complet de la sécurité des conteneurs.

FAQ — Questions fréquentes sur la sécurité des conteneurs

Quelle est la différence entre un scan de vulnérabilités statique et la sécurité runtime ?

Le scan statique analyse l'image avant exécution pour détecter les packages vulnérables, les mauvaises configurations et les secrets. La sécurité runtime surveille le comportement réel du conteneur en production via eBPF (Falco) pour détecter des activités anormales comme l'exécution d'un shell, des connexions réseau inattendues ou des accès à des fichiers sensibles. Ces deux approches sont complémentaires : le scan statique est la prévention, le runtime est la détection.

Les images distroless sont-elles vraiment impossibles à déboguer en production ?

Le débogage des conteneurs distroless requiert des approches alternatives. Kubernetes dispose de la fonctionnalité **ephemeral containers** (stable depuis 1.25) : `kubectl debug -it pod/mypod --image=busybox --target=myapp` injecte un conteneur de débogage temporaire dans le même namespace de processus, permettant l'inspection sans modifier l'image de production. `kubectl exec` reste possible sur un pod distroless si le processus principal accepte les signaux.

Comment gérer les secrets dans les conteneurs Kubernetes sans les exposer dans les variables d'environnement ?

Les variables d'environnement Kubernetes Secret sont visibles via `kubectl exec -- env` et dans les logs d'événements. Les alternatives plus sécurisées sont : (1) **HashiCorp Vault** avec l'agent Vault sidecar qui injecte les secrets en fichiers montés, (2) **AWS Secrets Manager / Azure Key Vault** via le CSI Secrets Store Driver, (3) **Sealed Secrets** (Bitnami) pour chiffrer les secrets dans Git, et (4) **External Secrets Operator** pour synchroniser depuis des coffres-forts externes. Le montage en fichier (volume) est toujours préférable aux variables d'environnement pour les secrets sensibles.

Quelle est la différence entre Kyverno et OPA Gatekeeper pour l'admission control Kubernetes ?

OPA Gatekeeper utilise le langage Rego (puissant mais avec une courbe d'apprentissage élevée) et offre une grande flexibilité pour les politiques complexes. Kyverno est spécifiquement conçu pour Kubernetes et utilise des manifests YAML natifs (plus accessible), avec des fonctionnalités supplémentaires comme la génération automatique de ressources et la mutation de manifests. Pour les équipes débutant avec les politiques K8s, Kyverno est plus accessible. Pour des politiques très complexes avec logique conditionnelle avancée, OPA/Gatekeeper est plus puissant.

Comment sécuriser les images construites dans les pipelines CI/CD contre les attaques sur la chaîne d'approvisionnement ?

La protection de la supply chain repose sur : (1) épingler TOUTES les images de base par digest SHA256 dans les Dockerfiles, (2) vérifier les signatures des images de base avec Cosign avant le build, (3) générer et attester un SBOM pour chaque image produite, (4) signer les images finales avec des clés liées à l'identité OIDC du pipeline CI/CD (Sigstore keyless signing), (5) utiliser SLSA Build Level 3 (builds hermétiques, provenance vérifiable), et (6) scanner les dépendances applicatives (npm, pip, go modules) avec Dependabot ou Renovate.

Falco peut-il détecter une évasion de conteneur (container escape) en temps réel ?

Falco peut détecter les *indicateurs* d'une évasion de conteneur, notamment : l'écriture dans des répertoires hôte montés anormalement, des accès à `/proc/1/` ou `/proc/*/root` depuis un conteneur, l'exécution de binaires hôte depuis un namespace de conteneur, et des changements dans les capabilities Linux.

Cependant, une évasion réussie et silencieuse (sans syscalls inhabituels) peut passer inaperçue. Les règles Falco doivent être complétées par un monitoring des kernel audit logs et une surveillance réseau au niveau hôte.

Quels sont les prérequis techniques pour activer les conteneurs rootless en production ?

Les conteneurs rootless nécessitent : (1) noyau Linux >= 5.11 (pour eBPF rootless complet) avec `CONFIG_CGROUPS_V2` activé, (2) `newuidmap` et `newgidmap` installés (package `uidmap`), (3) plages d'UIDs/GIDs configurées dans `/etc/subuid` et `/etc/subgid`, (4) `net.ipv4.ip_unprivileged_port_start=0` si des ports < 1024 sont nécessaires. Les limitations incluent : certains plugins CNI non supportés, performances légèrement inférieures pour les I/O réseau, et incompatibilité avec certains drivers de stockage.

Comment implémenter le scanning de conteneurs dans une pipeline GitLab CI ?

GitLab intègre nativement Container Scanning (basé sur Trivy) depuis la version 15.0. Il suffit d'inclure le template dans le `.gitlab-ci.yml` : `include: template: Security/Container-Scanning.gitlab-ci.yml` et de définir `CONTAINER_SCANNING_DISABLED: "false"`. Les résultats apparaissent dans l'onglet Security du pipeline et dans le Vulnerability Report du projet. Pour un contrôle plus fin, l'implémentation manuelle avec `trivy image --format template --template @contrib/gitlab.tpl` permet d'exporter en format DAST compatible GitLab.

Pour aller plus loin sur la sécurité Kubernetes, consultez notre article sur le hardening RBAC Kubernetes. La sécurité des conteneurs s'inscrit dans une démarche DevSecOps plus large couverte dans notre guide sur l'intégration de la sécurité dans les pipelines CI/CD. Pour la surveillance des menaces runtime, notre analyse de eBPF pour la sécurité Linux approfondit les mécanismes sous-jacents de Falco. La gestion des secrets en production est détaillée dans notre article sur HashiCorp Vault avec Kubernetes. Enfin, pour la conformité réglementaire des environnements conteneurisés, voir la conformité cloud ISO 27001.

Références externes : OWASP Docker Top 10 et NIST SP 800-190 Application Container Security Guide.

21. cgroups v2 et resource limits pour la sécurité

Les cgroups (Control Groups) version 2, devenus le standard depuis Linux 5.2 et adoptés par Docker et Kubernetes, permettent non seulement de limiter les ressources mais aussi de renforcer la sécurité des conteneurs en empêchant les attaques par déni de service interne et l'évasion par fork bomb.

```
# Vérifier que cgroups v2 est utilisé
mount | grep cgroup
# cgroup2 on /sys/fs/cgroup type cgroup2 (rw, nosuid, nodev, noexec, relatime)

# Docker : activer cgroups v2 (Linux 5.2+)
# ... (extrait — voir documentation officielle)
```

22. Kubernetes Admission Controllers personnalisés

Les Admission Controllers Kubernetes s'intercalent dans le chemin d'authentification pour valider ou muter les ressources avant leur persistance dans etcd. Au-delà d'OPA Gatekeeper et Kyverno, il est possible de développer des webhooks d'admission personnalisés.

```
// Webhook d'admission personnalisé en Go
// Valide que toutes les images proviennent d'un registre approuvé

package main

# ... (extrait — voir documentation officielle)
```

23. Analyse forensique d'un conteneur compromis

```
# Procédure forensique pour un conteneur suspect en production

# 1. Ne PAS supprimer le conteneur — conserver les preuves
# Identifier le conteneur suspect
docker ps -a | grep suspicious-pod
# ... (extrait — voir documentation officielle)
```

24. Sécurisation d'un cluster Kubernetes en production

```
# Audit de sécurité Kubernetes avec kube-bench (CIS Benchmark)
docker run --rm --pid=host --userns=host --net=host \\
  -v /etc:/etc:ro \\
  -v /var:/var:ro \\
  -v /usr/lib:/usr/lib:ro \\
# ... (extrait — voir documentation officielle)
```

```
apiVersion: v1
kind: Pod
metadata:
  name: kube-apiserver
  namespace: kube-system
# ... (extrait — voir documentation officielle)
```

```
# /etc/kubernetes/audit-policy.yaml — Politique d'audit Kubernetes
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
# Loguer toutes les opérations sur les secrets (niveau Metadata = pas de contenu)
# ... (extrait — voir documentation officielle)
```

25. Stratégie de mise à jour des images en production

```
#!/bin/bash
# Script de mise à jour sécurisée des images de conteneurs en production
# Intègre : scan Trivy, vérification de signature Cosign, déploiement progressif

IMAGE_NAME="$1"
# ... (extrait — voir documentation officielle)
```

Pipeline DevSecOps conteneurs complet : (1) Pre-commit : Hadolint (linting Dockerfile) + gitleaks (secrets). (2) CI Build : Trivy + Grype (scan image), Checkov (IaC). (3) CI Sign : Cosign (signature) + Syft (génération SBOM). (4) Registry : Harbor avec scanning automatique + blocage si CRITICAL. (5) Pre-deploy : Kyverno (vérification signature) + OPA (politiques). (6) Deploy : déploiement progressif avec métriques. (7) Runtime : Falco (détection anomalies) + Wazuh agent (monitoring). Cette chaîne constitue l'état de l'art de la sécurité des conteneurs en 2026.

26. Comparaison des runtimes de conteneurs

Runtime	Isolation	Performance	Sécurité	Support K8s	Use case
runc	Namespaces Linux	Excellente	Standard	Natif (OCI)	Production général
gVisor (runsc)	Sandbox userspace	Bonne	Très haute	Via RuntimeClass	Code non fiable
Kata Containers	VM légère (KVM/QEMU)	Moyenne	Maximale	Via RuntimeClass	Multi-tenant strict
Firecracker	microVM	Très bonne	Maximale	Via Flintlock	FaaS, Lambda
Podman	Namespaces Linux	Excellente	Haute (rootless)	Via CRI-O	Workstations, CI/CD

```
# Kubernetes RuntimeClass pour gVisor
# Prérequis : gVisor installé sur les nodes (runsc)
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
# ... (extrait — voir documentation officielle)
```

27. Surveillance des dépendances et mise à jour automatique

```
# Renovate Bot — Mise à jour automatique des dépendances et images Docker
# .github/renovate.json

{
  "$schema": "https://docs.renovatebot.com/renovate-schema.json",
# ... (extrait — voir documentation officielle)
```

28. Sécurité des images multi-architecture

Avec la généralisation des architectures ARM dans les environnements cloud (AWS Graviton, Apple Silicon pour le développement) et IoT, la sécurité des images multi-architecture est devenue une préoccupation concrète. Les images multi-architecture, distribuées via des Docker manifests lists, peuvent présenter des profils de vulnérabilités différents selon l'architecture cible, ce qui complique les workflows de scanning.

Un aspect souvent négligé est que le scanning d'une image multi-architecture se fait par défaut sur l'architecture de la machine qui effectue le scan. Si les pipelines CI/CD tournent sur des machines x86_64 mais que les images sont déployées sur des nodes ARM64 en production (Amazon Graviton, par exemple), les vulnérabilités spécifiques aux binaires ARM peuvent passer inaperçues. La solution consiste à forcer explicitement l'architecture lors du scanning avec Trivy via l'option `--platform linux/arm64`, ou à utiliser des runners CI/CD natifs ARM pour les builds et scans destinés aux nodes ARM.

Les images distroless elles-mêmes existent en variantes multi-architecture pour toutes les plateformes majeures (amd64, arm64, arm/v7, s390x, ppc64le), ce qui facilite l'adoption sans compromettre la couverture architecturale. Google maintient des SBOMs distincts pour chaque architecture dans le cadre de son programme de transparence des artefacts logiciels (SLSA Level 3 pour les images distroless officielles).

La construction d'images multi-architecture sécurisées avec Docker Buildx nécessite une stratégie de build cohérente. Le cross-compilation est préférable à l'émulation QEMU pour les performances, mais requiert que les dépendances soient disponibles pour l'architecture cible. Dans un contexte de sécurité de la supply chain, il est important de s'assurer que les images de base utilisées pour chaque architecture proviennent du même éditeur de confiance et sont épinglées par digest spécifique à l'architecture (chaque architecture a son propre digest dans un manifest list).

La gestion des secrets spécifiques aux architectures est un autre point d'attention. Certaines implémentations cryptographiques sont optimisées par architecture (AES-NI sur x86, AES extensions sur ARM). Les conteneurs doivent utiliser les bibliothèques adaptées à leur architecture cible pour bénéficier des accélérations matérielles, notamment pour les charges de travail cryptographiques intensives comme les terminaisons TLS ou les opérations de hachage en masse.

29. Service Mesh et sécurité mTLS entre microservices

Dans les architectures microservices Kubernetes, les communications entre services constituent une surface d'attaque significative souvent sous-estimée. Le chiffrement et l'authentification mutuels entre services via mTLS (mutual TLS) est la solution recommandée, et les service meshes comme Istio ou Linkerd automatisent ce processus de manière transparente pour les applications.

Un service mesh implémente mTLS en injectant un proxy sidecar (Envoy pour Istio, linkerd-proxy pour Linkerd) dans chaque pod. Ces proxies interceptent tout le trafic réseau entrant et sortant du pod et établissent des connexions TLS mutuellement authentifiées avec les autres sidecars, en utilisant des certificats de courte durée émis automatiquement par un Certificate Authority interne au mesh. Cette approche présente l'avantage de ne nécessiter aucune modification du code applicatif et de fournir une authentification service-to-service basée sur les identités SPIFFE (Secure Production Identity Framework For Everyone) plutôt que sur des adresses IP potentiellement usurpées.

La configuration de politiques d'autorisation dans Istio permet de définir précisément quels services peuvent communiquer avec quels autres services, et avec quelles méthodes HTTP et chemins. Cette granularité va bien au-delà des NetworkPolicies Kubernetes qui opèrent au niveau IP/port, en ajoutant une couche d'autorisation au niveau L7. Par exemple, une politique Istio peut autoriser le service `payment-service` à appeler uniquement le chemin `POST /api/payments` du service `bank-connector`, bloquant tout autre appel entre ces deux services même s'ils sont dans le même namespace.

L'observabilité fournie par les service meshes est également précieuse pour la sécurité. Istio génère automatiquement des métriques (Prometheus), des logs d'accès et des traces distribuées (Jaeger) pour toutes les communications inter-services, créant un journal d'audit exhaustif de toutes les interactions entre microservices. Cette visibilité est fondamentale pour la détection d'anomalies comportementales (un service qui commence soudainement à appeler des services qu'il n'appelait pas habituellement peut indiquer une compromission) et pour la réponse aux incidents (reconstituer le chemin d'une attaque de lateral movement à travers les microservices).

30. Sécurité des registres de conteneurs en entreprise

La sécurisation du registre de conteneurs est un maillon souvent négligé de la chaîne de sécurité. Un registre compromis peut permettre à un attaquant d'injecter du code malveillant dans toutes les images distribuées aux équipes de développement et aux environnements de production. Les bonnes pratiques de sécurité pour les registres d'entreprise couvrent plusieurs dimensions.

Le contrôle d'accès au registre doit suivre le principe du moindre privilège. Les pipelines CI/CD n'ont besoin que de droits de push (écriture) sur des espaces de noms spécifiques correspondant aux projets qu'ils gèrent. Les développeurs individuels ne devraient avoir que des droits de lecture en production. Seuls les comptes de service dédiés aux pipelines de déploiement (ArgoCD, Flux) ont besoin de droits de pull. Les comptes administrateurs doivent être protégés par MFA et leur utilisation auditée.

La politique de rétention des images est une considération à la fois de sécurité et de gouvernance. Les images vulnérables ou obsolètes stockées dans le registre représentent un risque si elles sont accidentellement redéployées. Une politique automatisée doit supprimer les images non taguées après un délai court (7 jours), archiver ou supprimer les versions plus anciennes que la politique de rétention (par exemple, garder les 5 dernières versions de chaque tag stable), et bloquer les pulls d'images ayant des vulnérabilités CRITICAL non résolues depuis plus de 30 jours.

La séparation des registres par environnement est une autre bonne pratique. Les images en développement (non validées) ne doivent pas être accessibles depuis les environnements de production. Un pipeline de promotion formalisé (développement → staging → production) avec un registre distinct pour chaque environnement, une signature Cosign requise pour passer du staging à la production, et une validation manuelle optionnelle pour les changements majeurs, garantit qu'aucune image non validée n'atteint la production.

La surveillance continue des images en production via des scanners comme Trivy Operator (qui scanne les images déployées dans Kubernetes et crée des ressources VulnerabilityReport) ou les solutions commerciales comme Aqua Security ou Sysdig permet de détecter de nouvelles CVEs dans des images déjà déployées, sans attendre le prochain build. Quand une CVE critique est publiée et affecte une image en

production, le système d'alerte doit notifier l'équipe responsable en quelques heures pour permettre une remédiation rapide.

31. Sécurité des Helm Charts et des packages Kubernetes

Les Helm Charts sont devenus le standard de packaging des applications Kubernetes, mais leur sécurité est souvent négligée. Un Chart mal configuré peut déployer des workloads avec des privilèges excessifs, des images non signées, ou des configurations réseau permissives qui annulent tous les efforts de hardening réalisés au niveau des pods individuels.

La sécurisation des Helm Charts commence par l'audit des valeurs par défaut (`values.yaml`). Les Charts publics (Helm Hub, Artifact Hub) sont créés pour la facilité d'utilisation plutôt que pour la sécurité maximale, ce qui se traduit par des valeurs par défaut permissives : `securityContext` absent, `hostNetwork: false` non enforced, `resources` sans limites. Avant de déployer un Chart externe en production, un audit systématique de ses valeurs et templates est indispensable.

L'outil `helm-audit` et le plugin `helm-checkov` permettent d'analyser automatiquement les Charts pour détecter les mauvaises configurations. Trivy supporte également le scanning des Charts via `trivy config ./mychart/`. Pour les organisations utilisant Helm comme standard de déploiement, la mise en place d'une bibliothèque de Charts internes basés sur des standards de sécurité définis (securityContext obligatoire, resource limits par défaut, networkPolicy générée automatiquement) est une approche plus efficace que l'audit au cas par cas.

La signature des Charts avec le mécanisme de provenance Helm (`helm package --sign` avec une clé GPG) permet de vérifier l'intégrité et la provenance d'un Chart avant son déploiement. Cette fonctionnalité, disponible depuis Helm 2 mais peu utilisée en pratique, prend de l'importance dans le contexte de la supply chain sécurité où la compromission d'un repository Helm pourrait permettre l'injection de Charts malveillants.

La gestion des secrets dans les Helm Charts est un autre point critique. La pratique de stocker des secrets en clair dans les `values.yaml` ou les templates Helm est malheureusement courante. Les alternatives sécurisées incluent : l'utilisation du plugin `helm-secrets` (qui intègre SOPS pour le chiffrement des fichiers de valeurs sensibles dans Git), l'injection des secrets via External Secrets Operator depuis un vault externe au moment du déploiement, ou la référence à des Kubernetes Secrets existants (créés par un pipeline séparé) plutôt que leur création directe dans les templates Helm.

32. Conteneurs et conformité PCI-DSS, HIPAA, SOC 2

Les environnements conteneurisés soumis à des obligations réglementaires spécifiques (PCI-DSS pour les traitements de paiement, HIPAA pour les données de santé américaines, SOC 2 pour les prestataires de services cloud) nécessitent des contrôles supplémentaires au-delà du hardening technique général. La mise en conformité dans ces contextes demande une compréhension fine des exigences réglementaires et de leur traduction dans les environnements Kubernetes.

PCI-DSS v4.0 (applicable depuis mars 2024) impose, pour les environnements de traitement des données de paiement conteneurisés : l'isolation réseau stricte entre les pods du CDE (Cardholder Data Environment)

et les autres pods via des NetworkPolicies (satisfait aux exigences de segmentation 1.3.x), le logging et la surveillance de toutes les activités dans le CDE avec rétention de 12 mois (Falco + Wazuh couvrent les exigences 10.x), la gestion des patches dans les 30 jours pour les vulnérabilités critiques (workflow Trivy + Renovate), et la gestion des identités et des accès avec MFA pour tous les accès aux systèmes CDE (satisfait par l'RBAC Kubernetes + Entra ID/OIDC avec MFA).

HIPAA (Health Insurance Portability and Accountability Act) pour les données de santé américaines exige des contrôles d'accès (AC), d'audit (AU) et d'intégrité (SI) qui se traduisent concrètement dans les environnements Kubernetes par : le chiffrement au repos des Persistent Volumes contenant des PHI (Protected Health Information) via les fonctionnalités de chiffrement des PVC AWS EBS/GCP PD, le chiffrement en transit via mTLS (Istio/Linkerd), la gestion des clés de chiffrement via un KMS external (AWS KMS, HashiCorp Vault), et la piste d'audit complète de tous les accès aux données PHI (Kubernetes Audit Logs + Falco).

SOC 2 Type II, la certification de sécurité la plus demandée pour les prestataires SaaS, évalue la conception ET l'efficacité opérationnelle des contrôles sur une période de 6 à 12 mois. Pour les environnements conteneurisés, les contrôles SOC 2 les plus souvent évalués sont : la gestion du changement (git flow, code review, déploiement automatisé via CI/CD avec approbations), la surveillance continue (dashboards Grafana + alertes PagerDuty pour les métriques de sécurité), la gestion des accès (revue trimestrielle des accès Kubernetes RBAC), et la réponse aux incidents (runbooks documentés, exercices de tabletop annuels). La collecte de preuves pour SOC 2 est facilitée par les logs immuables des pipelines CI/CD et des systèmes d'audit Kubernetes.

33. Benchmarking de performance : impact des contrôles de sécurité

L'application de contrôles de sécurité sur les conteneurs a un impact mesurable sur les performances, qu'il est important de quantifier pour dimensionner correctement les ressources et arbitrer les choix de sécurité en connaissance de cause. L'impact est souvent surestimé par les équipes de développement et sous-estimé par les équipes de sécurité, d'où l'importance de mesures objectives.

Le profil seccomp avec l'option `RuntimeDefault` (le profil par défaut de Docker/Kubernetes) introduit une latence inférieure à 2% sur la plupart des charges de travail applicatives selon les benchmarks Aqua Security 2024, car il ne filtre que les syscalls rarement utilisés. Un profil seccomp personnalisé très restrictif peut réduire cette latence à zéro ou même légèrement l'améliorer en réduisant les vérifications kernel. L'activation d'AppArmor avec un profil adapté au workload introduit une latence similaire, inférieure à 3% selon les tests Ubuntu 22.04.

Les conteneurs rootless avec Docker présentent une légère pénalité de performance (5-10%) pour les opérations d'I/O réseau intensives en raison de la couche supplémentaire de traduction des espaces de noms d'utilisateurs. Pour les applications web standard, cet impact est négligeable. Pour les applications à très haute performance réseau (streaming, base de données), l'évaluation cas par cas est recommandée. Podman rootless présente généralement de meilleures performances que Docker rootless pour les opérations réseau grâce à son implémentation différente des user namespaces.

Les runtimes d'isolation renforcée ont un impact plus significatif. gVisor (runsc) introduit une pénalité de 10-30% sur les opérations I/O système en raison de son interception de tous les syscalls dans son sandbox

userspace. Kata Containers avec une microVM QEMU présente une pénalité similaire (10-25%) avec un surcoût de démarrage de 0.5-1 seconde par pod. Ces pénalités sont acceptables pour les workloads traitant du code non fiable (exécution de code utilisateur, microservices exposés directement à l'internet) mais trop importantes pour les workloads latency-sensitive. Le choix du runtime doit donc être guidé par le profil de risque de chaque type de workload.

Conclusion et recommandations

La maîtrise de ces techniques et outils est indispensable pour tout professionnel de la cybersécurité en 2026. L'évolution constante des menaces exige une veille permanente et une mise à jour régulière des compétences. Pour aller plus loin, consultez nos articles techniques ou contactez notre équipe pour un accompagnement sur mesure adapté à votre contexte.

À retenir : La sécurité est un processus continu, pas un état. Chaque audit, chaque test et chaque analyse contribue à renforcer la posture de défense de l'organisation face aux menaces actuelles et futures.