AYI NEDJIMI Consultants

# Sécurité des conteneurs : hardening Docker et Kubernetes

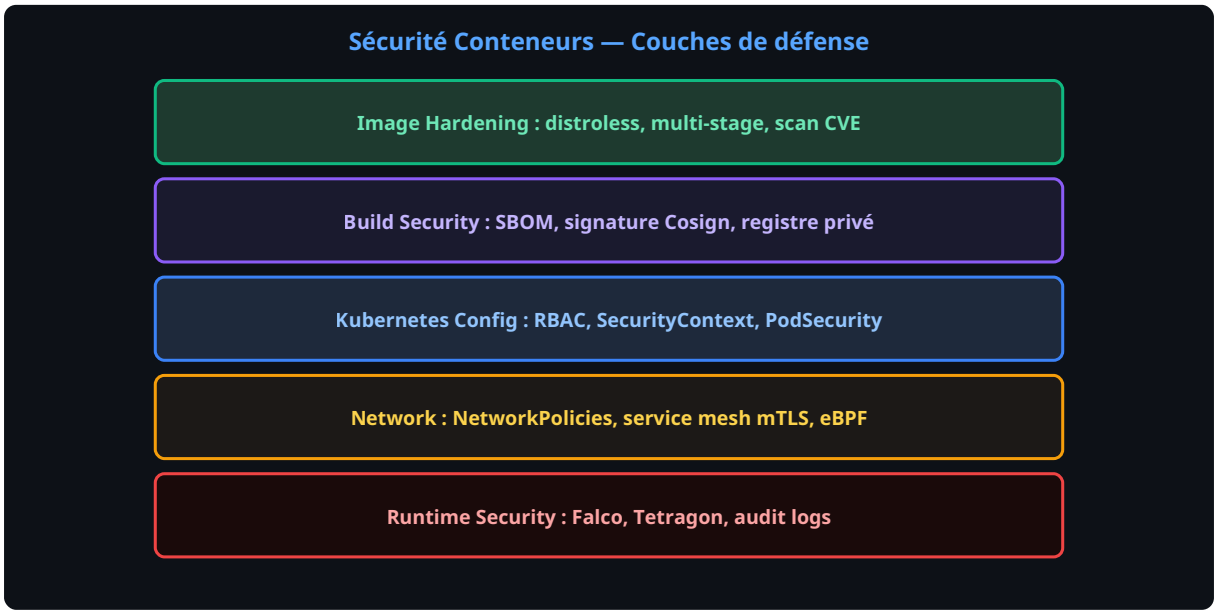
Catégorie : DevSecOps  Lecture : 4 min  Publié le : 12/03/2026  Auteur : Ayi NEDJIMI

*Guide complet pour durcir vos conteneurs Docker et clusters Kubernetes en production. Images minimales, RBAC, network policies et runtime security.*

Vos conteneurs tournent en production avec des droits root, des images basées sur Ubuntu avec 400 paquets inutiles, et un cluster Kubernetes où chaque pod communique librement avec tous les autres ? Vous n'êtes pas seul dans cette situation. D'après les analyses de Sysdig sur leur base de clients, 76% des conteneurs en production tournent en root, et 90% des clusters n'appliquent aucune network policy. Le passage aux conteneurs apporte de l'agilité, mais sans durcissement, vous déplacez juste vos vulnérabilités dans un environnement plus dynamique et plus difficile à surveiller. Ce guide couvre les deux dimensions du problème : le hardening des images Docker en amont, et la sécurisation du cluster Kubernetes en production. De la construction d'images minimales avec Chainguard au déploiement de Falco pour la détection runtime, vous trouverez des configurations concrètes et testées pour chaque recommandation pratique.

## Points clés à retenir

- Utilisez des images **distroless** ou **Chainguard** pour réduire la surface d'attaque de 90%
- Le **SecurityContext** Kubernetes (runAsNonRoot, readOnlyRootFilesystem, drop ALL) bloque la majorité des exploits
- Les **network policies** sont votre micro-segmentation : sans elles, un pod compromis accède à tout le cluster
- **Falco** en runtime détecte les comportements anormaux que le scanning statique ne voit pas



## Construire des images Docker minimales

La première règle du hardening Docker : moins il y a de composants dans l'image, moins il y a de surface d'attaque. Une image basée sur `ubuntu:24.04` contient environ 100 paquets et 120+ CVE connues à un instant T. Une image **distroless** de Google ou `chainguard/static` en contient zéro. La technique du multi-stage build est votre alliée :

```
FROM golang:1.22-alpine AS builder
WORKDIR /app
COPY . .
RUN CGO_ENABLED=0 go build -o /app/server

FROM cgr.dev/chainguard/static:latest
COPY --from=builder /app/server /server
USER nonroot
ENTRYPOINT ["/server"]
```

L'image finale ne contient que votre binaire. Pas de shell, pas de package manager, pas de curl — un attaquant qui obtient une RCE ne peut quasiment rien exploiter. Scannez systématiquement vos images avec **Trivy** ou **Grype** avant le push sur le registry. Intégrez ce scan dans votre pipeline CI/CD sécurisé comme gate bloquante.

## SecurityContext Kubernetes : la configuration ignorée par 80% des clusters

Le **SecurityContext** est le mécanisme natif de Kubernetes pour restreindre les privilèges d'un pod. Voici la configuration minimale pour la production :

```
securityContext:
  runAsNonRoot: true
  runAsUser: 65534
  readOnlyRootFilesystem: true
  allowPrivilegeEscalation: false
  capabilities:
    drop: ["ALL"]
  seccompProfile:
    type: RuntimeDefault
```

`runAsNonRoot` empêche le conteneur de tourner en root. `readOnlyRootFilesystem` bloque l'écriture sur le filesystem sauf les volumes montés explicitement. `drop ALL` supprime toutes les capabilities Linux. Pour appliquer ces contraintes à l'échelle du cluster, utilisez les Pod Security Standards ou mieux, des politiques Policy as Code avec Kyverno qui offrent plus de flexibilité.

## Network policies : la micro-segmentation des conteneurs

Par défaut, Kubernetes autorise toute communication entre pods. Un pod compromis dans le namespace frontend peut interroger directement votre base de données dans le namespace backend. Les **NetworkPolicies** corrigent cette faille architecturale :

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: backend-db-only
  namespace: backend
spec:
  podSelector:
    matchLabels:
      app: postgres
  policyTypes: ["Ingress"]
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: api-server
      ports:
        - port: 5432
```

Commencez par une politique `default-deny` dans chaque namespace, puis ouvrez uniquement les flux nécessaires. C'est le principe du moindre privilège appliqué au réseau. Pour les clusters cloud, notre guide sur la posture de sécurité cloud complète cette approche.

## Détection runtime avec Falco et Tetragon

**Falco** (CNCF) surveille les appels système de vos conteneurs en temps réel grâce à eBPF. Il détecte les comportements anormaux : exécution d'un shell dans un conteneur, lecture de fichiers sensibles, connexion réseau inattendue, modification de binaires système. **Tetragon** (Cilium) va plus loin en permettant non seulement la détection mais aussi le blocage en temps réel.

Les deux outils s'intègrent avec votre stack d'observabilité : alertes vers Prometheus/Alertmanager, logs vers votre SIEM. En cas d'incident, les données Falco ou Tetragon alimentent directement votre playbook de réponse aux incidents.

## Registre privé et admission control

Ne déployez jamais d'images provenant de registres publics non vérifiés. Mettez en place un registre privé — **Harbor** est l'option open-source la plus complète. Il scanne automatiquement chaque image avec Trivy et peut bloquer le pull des images avec des CVE critiques. Côté Kubernetes, un admission controller valide chaque demande de création de pod.

Selon les recommandations du guide ANSSI sur la sécurité des conteneurs Docker, l'utilisation d'un registre privé et la signature des images constituent des mesures de base indispensables. Le CIS Kubernetes Benchmark fournit des centaines de contrôles automatisables avec **kube-bench** d'Aqua Security.

| Mesure de hardening | Impact sécurité | Complexité | Priorité |
|---|---|---|---|
| Images distroless / Chainguard | Réduit 90% des CVE image | Moyenne | Haute |
| SecurityContext restrictif | Bloque privilege escalation | Faible | Critique |
| NetworkPolicies default-deny | Micro-segmentation réseau | Moyenne | Haute |
| Falco / Tetragon runtime | Détection temps réel | Moyenne | Haute |
| Harbor + admission control | Supply chain image | Élevée | Haute |
| kube-bench audit CIS | Conformité benchmark | Faible | Moyenne |

**Sources et références :** OWASP DevSecOps · NIST

## Questions fréquentes sur la sécurité des conteneurs

### Les conteneurs sont-ils plus sécurisés que les machines virtuelles ?

Non, par défaut ils le sont moins. Les conteneurs partagent le kernel de l'hôte — une vulnérabilité d'évasion donne accès à l'ensemble de l'hôte. Les VM bénéficient d'une isolation hardware via l'hyperviseur. Le hardening décrit dans cet article (seccomp, capabilities, user namespaces) réduit considérablement l'écart, mais l'isolation d'une VM reste supérieure.

### Faut-il scanner les images à chaque déploiement ou seulement au build ?

Les deux. Au build pour bloquer les images vulnérables avant le registry. En continu sur le registry pour détecter les nouvelles CVE publiées après le build. Harbor automatise ce rescanning. Une image saine au build peut devenir critique 2 semaines plus tard.

### Quel est l'impact performance de Falco en production ?

Avec le driver eBPF recommandé, Falco consomme environ 1-3% de CPU et 200-300 Mo de RAM par noeud. C'est négligeable pour la plupart des clusters. Pour les clusters à très haute charge, Tetragon avec son architecture eBPF native est encore plus léger.

**Ayi NEDJIMI Consultants** — Expert cybersécurité offensive & intelligence artificielle

ayinedjimi-consultants.fr · ayi@ayinedjimi-consultants.fr