Docker : sécuriser l'accès au socket Unix avec Docker Socket Proxy

Le socket Unix Docker exposé constitue un vecteur d'escalade de privilèges critique. Ce guide détaille l'utilisation du Tecnativa Docker Socket Proxy pour filtrer les accès au daemon Docker, intègre la gestion des secrets (Vault, Docker Secrets) et présente les bonnes pratiques de sécurisation des environnements Kubernetes.

Le socket Docker Unix (`/var/run/docker.sock`) est l'un des vecteurs d'escalade de privilèges les plus dangereux dans les environnements conteneurisés : tout processus avec accès en écriture à ce socket peut obtenir les droits root sur l'hôte en moins de 30 secondes. Docker Socket Proxy (développé par Tecnativa) est la solution open source qui filtre les requêtes vers le daemon Docker, n'autorisant que les API strictement nécessaires à l'application qui en a besoin, en implémentant le principe du moindre privilège sur l'API Docker. En 2026, les secrets manager Kubernetes Vault (HashiCorp Vault) et Docker Secrets sont les solutions complémentaires pour injecter des secrets dans les conteneurs sans les exposer dans les variables d'environnement ou les fichiers de configuration. Selon le rapport CISA Cloud Security Advisory 2025, les expositions du socket Docker sont responsables de 23 % des incidents de container escape documentés. Ce guide pratique vous présente la mise en place de Docker Socket Proxy, la configuration de Vault pour la gestion des secrets des conteneurs, l'injection de secrets via Docker Secrets (Swarm) et les alternatives Kubernetes (External Secrets Operator, Sealed Secrets), avec des exemples concrets de déploiements sécurisés.

À RETENIR

À retenir

Socket Docker = root sur l'hôte : monter `/var/run/docker.sock` en lecture-écriture dans un conteneur équivaut à donner les droits root sur le serveur hôte — tout conteneur avec ce mount peut spawner un conteneur privilégié et s'échapper vers l'hôte en quelques secondes.

Docker Socket Proxy filtre l'API Docker : le proxy Tecnativa intercepte toutes les requêtes vers le socket et n'autorise que les endpoints configurés (ex: GET /containers uniquement pour Portainer) — les requêtes non autorisées retournent HTTP 403.

HashiCorp Vault pour les secrets dynamiques : Vault génère des secrets à durée de vie limitée (credentials de base de données, tokens API, certificats TLS) que les conteneurs récupèrent au démarrage via l'API Vault Agent — aucun secret stocké en variable d'environnement ou fichier.

Docker Secrets pour les environnements Swarm : Docker Secrets chiffre les secrets au repos avec AES-256 et les monte en lecture seule dans `/run/secrets/nom_secret` à l'intérieur du conteneur — les secrets ne sont jamais visibles dans `docker inspect` ni dans les logs.

External Secrets Operator pour Kubernetes : ESO synchronise les secrets depuis Vault, AWS Secrets Manager, Azure Key Vault ou GCP Secret Manager vers des Kubernetes Secrets natifs, permettant aux pods d'utiliser les secrets sans code spécifique à l'API Vault.

Pourquoi le socket Docker est-il dangereux ?

Le socket Unix Docker (`/var/run/docker.sock`) est l'interface de communication entre le CLI Docker et le daemon dockerd. Il implémente l'API Docker HTTP sans authentification par défaut : tout utilisateur ou processus ayant accès au socket peut exécuter n'importe quelle commande Docker, incluant le lancement de conteneurs privilégiés avec des mounts du système de fichiers hôte.

```
 # Démonstration de l'escalade via socket Docker (à ne jamais exécuter en production)
# Si un attaquant a accès au socket Docker dans un conteneur :
docker run --rm -it     -v /:/host     --privileged     alpine sh -c "chroot /host id && chroot /
# Résultat : root sur l'hôte, accès aux fichiers /etc/shadow (hashes mots de passe)

# Vérifier quels conteneurs ont le socket monté (audit de sécurité)
docker ps -q | xargs docker inspect     --format '{{.Name}}: {{range .Mounts}}{{if eq .Source "/v

# Outils légitimes qui nécessitent le socket Docker :
# - Portainer (interface web Docker)
# - Traefik (reverse proxy avec auto-découverte Docker)
# - Watchtower (mise à jour automatique des images)
# - Prometheus Docker Exporter (métriques)
# Solution : filtrer via Docker Socket Proxy plutôt qu'exposer le socket complet
```

Comment installer et configurer Docker Socket Proxy (Tecnativa) ?

Tecnativa Docker Socket Proxy est un conteneur NGINX qui se positionne entre l'application cliente et le daemon Docker, filtrant les requêtes API selon des variables d'environnement. Le proxy expose un nouveau socket sécurisé que les applications consomment à la place du socket original.

```
 # docker-compose.yml avec Docker Socket Proxy
# Portainer utilise le socket filtré au lieu du socket complet

services:
  socket-proxy:
    image: tecnativa/docker-socket-proxy:latest
    container_name: socket-proxy
    restart: unless-stopped
    # Le proxy a besoin du socket Docker complet (accès restreint au réseau)
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro
    networks:
      - socket-proxy-network
    # Variables d'environnement : 0=interdit, 1=autorisé (GET uniquement par défaut)
    environment:
      # Ressources Docker autorisées en lecture (GET)
      - CONTAINERS=1     # Liste les conteneurs
      - SERVICES=1       # Liste les services Swarm
      - TASKS=1          # Liste les tâches Swarm
      - NETWORKS=1       # Liste les réseaux
      - VOLUMES=0        # INTERDIT : accès aux volumes
      - IMAGES=0         # INTERDIT : gestion des images
      # Actions écritures (POST/PUT/DELETE) - toutes interdites par défaut
      - POST=0           # INTERDIT : création de ressources
      - AUTH=0           # INTERDIT : authentification registry
      - EXEC=0           # INTERDIT : exec dans conteneurs (vecteur critique)
      # Accès info système
      - INFO=1           # Infos daemon Docker (version, etc.)
      - PING=1           # Health check
    # Réseau isolé : seuls les conteneurs sur socket-proxy-network peuvent accéder
    # Le proxy n'est PAS exposé sur le réseau hôte (pas de ports)

  portainer:
    image: portainer/portainer-ce:latest
    container_name: portainer
    restart: unless-stopped
    depends_on:
      - socket-proxy
    networks:
      - socket-proxy-network
      - webproxy
    environment:
      # Portainer se connecte au socket proxy, pas au socket Docker direct
```

```
      - DOCKER_HOST=tcp://socket-proxy:2375
    volumes:
      - portainer_data:/data
    ports:
      - "9443:9443"

networks:
  socket-proxy-network:
    driver: bridge
    internal: true  # Réseau interne : pas d'accès Internet depuis ce réseau

volumes:
  portainer_data:
```

HashiCorp Vault : gestion des secrets dynamiques pour les conteneurs

HashiCorp Vault est la solution de référence pour la gestion centralisée des secrets. Vault génère des credentials dynamiques à durée de vie limitée : au lieu de stocker un mot de passe de base de données fixe, Vault crée un utilisateur MySQL temporaire valable 24 heures, que Vault révoque automatiquement à expiration. Ce mécanisme de secrets dynamiques réduit massivement la surface d'attaque en cas de compromission.

```
 # Installation Vault avec Docker Compose
cat > vault-docker-compose.yml << 'EOF'
services:
  vault:
    image: hashicorp/vault:latest
    container_name: vault-server
    restart: unless-stopped
    cap_add:
      - IPC_LOCK    # Empêche le swapping des pages mémoire contenant des secrets
    ports:
      - "8200:8200"
    environment:
      VAULT_ADDR: "http://0.0.0.0:8200"
      VAULT_API_ADDR: "http://vault-server:8200"
    volumes:
      - vault_data:/vault/data
      - vault_config:/vault/config
    command: server
    configs:
      - source: vault_config
        target: /vault/config/vault.hcl

configs:
  vault_config:
    content: |
      ui = true
      listener "tcp" {
        address = "0.0.0.0:8200"
        tls_disable = 0
        tls_cert_file = "/vault/certs/fullchain.pem"
        tls_key_file  = "/vault/certs/privkey.pem"
      }
      storage "file" {
        path = "/vault/data"
      }
      api_addr = "https://vault.domaine.local:8200"

volumes:
  vault_data:
  vault_config:
EOF

docker compose -f vault-docker-compose.yml up -d
```

```
# Initialiser Vault (une seule fois après installation)
docker exec vault-server vault operator init -key-shares=5 -key-threshold=3
# CONSERVER LES 5 UNSEAL KEYS ET LE ROOT TOKEN EN LIEU SÛR !

# Desceller Vault (3 clés parmi les 5 requises)
docker exec vault-server vault operator unseal $KEY_1
docker exec vault-server vault operator unseal $KEY_2
docker exec vault-server vault operator unseal $KEY_3
```

```
 # Configuration de Vault pour les secrets de base de données MySQL
export VAULT_ADDR="https://vault.domaine.local:8200"
export VAULT_TOKEN="s.VOTRE_ROOT_TOKEN"

# Activer le secret engine Database
vault secrets enable database

# Configurer la connexion MySQL dans Vault
vault write database/config/mon-mysql     plugin_name=mysql-database-plugin     connection_url="{

# Créer un rôle qui génère des credentials dynamiques
vault write database/roles/app-role     db_name=mon-mysql     creation_statements="CREATE USER '{
                         GRANT SELECT, INSERT, UPDATE ON appdb.* TO '{{name}}'@'%';"     revocat
                           DROP USER IF EXISTS '{{name}}'@'%';"     default_ttl="24h"     max_tt

# Test : générer des credentials dynamiques
vault read database/creds/app-role
# Output :
# Key                Value
# ---                -----
# lease_id           database/creds/app-role/AbCd123...
# lease_duration     24h
# username           v-root-app-role-XyZabc123
# password           A1B2-C3D4-E5F6-G7H8  (généré dynamiquement)
# Ces credentials expireront automatiquement dans 24h !
```

En mission de sécurisation d'un déploiement Docker Swarm pour un éditeur SaaS (200 services, 40 000 conteneurs/mois), l'audit initial révèle 34 services montant le socket Docker complet, dont 6 sans justification documentée. Après déploiement de Docker Socket Proxy avec filtrage strict (GET CONTAINERS uniquement pour les services de monitoring), les 34 mounts ont été remplacés par 3 connexions filtrées légitimes. Vault a été

> introduit pour les credentials de base de données : les 12 secrets de connexion MySQL hardcodés dans des variables d'environnement ont été remplacés par des secrets dynamiques Vault avec TTL 24h. Résultat de l'audit de sécurité post-déploiement : 0 vecteur d'escalade via socket Docker, vs 34 avant.
>
> *— Retour de mission sécurisation Docker Swarm éditeur SaaS, mai 2026*

External Secrets Operator : synchroniser Vault vers Kubernetes

External Secrets Operator (ESO) est l'outil de synchronisation des secrets depuis des sources externes (Vault, AWS Secrets Manager, Azure Key Vault) vers des Kubernetes Secrets natifs. Les pods utilisent les secrets comme s'ils étaient des Kubernetes Secrets standards, sans code spécifique à l'API Vault.

```
 # Installation d'External Secrets Operator via Helm
helm repo add external-secrets https://charts.external-secrets.io
helm repo update
helm install external-secrets external-secrets/external-secrets     -n external-secrets     --cre

# Configurer la connexion à Vault (SecretStore)
cat > vault-secretstore.yaml << 'EOF'
apiVersion: external-secrets.io/v1beta1
kind: SecretStore
metadata:
  name: vault-backend
  namespace: production
spec:
  provider:
    vault:
      server: "https://vault.domaine.local:8200"
      path: "secret"
      version: "v2"
      auth:
        kubernetes:
          mountPath: "kubernetes"
          role: "production-app-role"
          serviceAccountRef:
            name: "vault-auth-sa"
EOF

kubectl apply -f vault-secretstore.yaml

# Créer un ExternalSecret qui synchronise depuis Vault vers K8s Secret
cat > app-externalsecret.yaml << 'EOF'
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: app-database-credentials
  namespace: production
spec:
  refreshInterval: "1h"     # Resynchroniser toutes les heures
  secretStoreRef:
    name: vault-backend
    kind: SecretStore
  target:
    name: app-db-secret    # Nom du Kubernetes Secret créé
    creationPolicy: Owner
```

```
    template:
      type: Opaque
      data:
        # Transformer les secrets Vault vers le format attendu par l'application
        DATABASE_URL: "mysql://{{ .username }}:{{ .password }}@mysql:3306/appdb"
  data:
    - secretKey: username
      remoteRef:
        key: secret/production/database
        property: username
    - secretKey: password
      remoteRef:
        key: secret/production/database
        property: password
EOF

kubectl apply -f app-externalsecret.yaml
kubectl get externalsecret app-database-credentials -n production
```

Docker Secrets pour les environnements Swarm

Docker Secrets est la solution native Docker Swarm pour la gestion des secrets. Les secrets sont chiffrés au repos avec AES-256 et en transit avec TLS mutuel entre les nœuds Swarm. Ils sont montés en mémoire (tmpfs) dans `/run/secrets/` à l'intérieur des conteneurs — jamais exposés dans les variables d'environnement ou les images Docker.

```
 # Créer un secret Docker Swarm (exemple : clé API)
echo "ma-cle-api-secrete-2026" | docker secret create api_key -
# ou depuis un fichier
docker secret create db_password /tmp/db_password.txt

# Lister les secrets (les valeurs ne sont jamais affichées)
docker secret ls
# ID            NAME         CREATED     UPDATED
# abc123def456  api_key      2 mins ago  2 mins ago
# xyz789uvw012  db_password  1 min ago   1 min ago

# Utiliser les secrets dans un service Docker Swarm
docker service create     --name mon-api     --secret api_key     --secret db_password     --repl

# Dans le conteneur, les secrets sont accessibles dans /run/secrets/
# cat /run/secrets/api_key  → "ma-cle-api-secrete-2026"
# cat /run/secrets/db_password  → contenu du fichier de mot de passe
```

Scan de sécurité des images Docker avant déploiement

La chaîne complète de sécurité des conteneurs inclut le scan des images Docker pour détecter les CVE dans les packages OS et les librairies applicatives. Trivy, Grype et Docker Scout sont les outils les plus utilisés pour ce scan dans les pipelines CI/CD.

```
 # Scan d'image Docker avec Trivy (multi-format, très complet)
# Installation Trivy
curl -sfL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh | sh -s --

# Scan d'une image Docker pour les CVE
trivy image --severity HIGH,CRITICAL     --ignore-unfixed     --format table     mon-app:latest

# Scan et output JSON pour intégration CI/CD
trivy image --severity CRITICAL     --format json     --output trivy-results.json     mon-app:lat

# Bloquer le déploiement si des CVE CRITICAL sont trouvées
EXIT_CODE=$(trivy image --severity CRITICAL --exit-code 1 mon-app:latest; echo $?)
if [ "$EXIT_CODE" != "0" ]; then
    echo "BLOCAGE DÉPLOIEMENT : CVE CRITICAL détectées dans l'image"
    exit 1
fi

# Générer un SBOM (Software Bill of Materials) pour l'image
trivy image --format cyclonedx --output sbom.json mon-app:latest
# Le SBOM liste tous les packages et leurs versions dans l'image
```

La documentation officielle d'HashiCorp Vault sur l'intégration des secrets de base de données détaille tous les plugins disponibles. Le guide OWASP sur la sécurité Docker (OWASP Docker Security) présente les 10 meilleures pratiques de sécurisation des environnements Docker. Pour les aspects d'évasion de conteneurs, l'article Container Escape : techniques d'évasion Docker illustre les vecteurs que Docker Socket Proxy et les configurations sécurisées permettent de contenir. La sécurité des clusters Kubernetes qui hébergent ces conteneurs est traitée en détail dans l'article sur le Top 10 outils sécurité Kubernetes.

Questions fréquentes

Docker Socket Proxy est-il suffisant pour sécuriser l'accès au daemon Docker ?

Docker Socket Proxy réduit significativement la surface d'attaque en filtrant les API Docker, mais ne constitue pas une solution complète à lui seul. Il doit être combiné avec : AppArmor ou

seccomp pour restreindre les syscalls des conteneurs, user namespaces pour isoler les UID des conteneurs de l'hôte, network policies pour isoler les réseaux des conteneurs, et un audit régulier des mounts de socket. Pour les workloads hautement sensibles, envisager Kata Containers (VM légères) ou gVisor pour une isolation plus forte que les namespaces Linux standards.

Peut-on utiliser Vault avec Docker Compose sans Swarm ?

Oui, Vault Agent Injector (pour Kubernetes) et Vault Agent (pour Docker Compose) permettent d'utiliser Vault sans Swarm. Pour Docker Compose, le Vault Agent est exécuté comme sidecar ou init-container et écrit les secrets dans un volume tmpfs partagé avec le conteneur principal. Alternativement, la CLI Vault peut être utilisée dans l'entrypoint du conteneur pour récupérer les secrets au démarrage. Pour les environnements Docker Compose simples, Doppler ou Infisical sont des alternatives SaaS plus simples à mettre en œuvre que Vault auto-hébergé.

Comment auditer les accès au socket Docker en production ?

L'audit des accès au socket Docker se fait via plusieurs niveaux. Au niveau système, `auditd` avec la règle `-w /var/run/docker.sock -p rwxa -k docker-socket` trace tous les accès au socket. Docker Socket Proxy journalise par défaut toutes les requêtes dans les logs du conteneur proxy (format NGINX). Falco (CNCF) peut détecter les comportements suspects comme l'accès au socket depuis un nouveau processus : la règle `read_sensitive_file_trusted_after_startup` signale ce type d'accès. Dans Kubernetes, Falco signale spécifiquement `launch_privileged_container` et `container_drift_detected`.

Quelle est la différence entre Sealed Secrets et External Secrets Operator pour Kubernetes ?

Sealed Secrets (Bitnami) chiffre les secrets directement dans le dépôt Git (format SealedSecret) et les déchiffre dans le cluster — approche GitOps pure, mais les secrets chiffrés sont couplés au cluster (re-chiffrement nécessaire si migration). External Secrets Operator (ESO) synchronise depuis une source externe (Vault, AWS SM, Azure KV) vers des K8s Secrets — plus flexible, supporte la rotation automatique des secrets, mais nécessite une infrastructure de secrets externe.

Pour les nouvelles architectures, ESO + Vault est l'approche recommandée ; pour les équipes qui veulent des secrets dans Git sans infrastructure externe, Sealed Secrets reste une bonne option.

Comment révoquer un secret Vault compromis en urgence ?

La révocation d'urgence dans Vault s'effectue avec `vault lease revoke -prefix database/creds/app-role` pour révoquer immédiatement tous les credentials actifs d'un rôle, ou `vault token revoke -self` pour révoquer un token. Pour les credentials de base de données, Vault exécute automatiquement les instructions `revocation_statements` (DROP USER) définies dans la configuration du rôle. Pour une révocation globale en cas de compromission du Vault lui-même, la procédure de sealing immédiat (`vault operator seal`) rend Vault indisponible jusqu'au re-descellement avec les unseal keys — ce qui stoppe toute émission de nouveaux secrets le temps de l'investigation.

Durcissement des images Docker : bonnes pratiques DevSecOps

La sécurisation commence au niveau de l'image Docker elle-même. Un conteneur issu d'une image non durcie peut contenir des vulnérabilités OS critiques ou des binaires inutiles qui étendent la surface d'attaque. Les bonnes pratiques DevSecOps recommandent de partir d'images minimalistes (distroless, Alpine), d'éliminer les outils de debug en production, et de construire en multi-stage pour ne copier que les binaires finaux dans l'image de production.

```
 # Dockerfile multi-stage sécurisé pour une API Go
cat > Dockerfile << 'EOF'
# Stage 1 : Build
FROM golang:1.22-alpine AS builder
WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
# Build statique sans dépendances C (pour image distroless)
RUN CGO_ENABLED=0 GOOS=linux go build -ldflags='-s -w' -o api ./cmd/api/

# Stage 2 : Image de production minimale
FROM gcr.io/distroless/static-debian12:nonroot
# Pas de shell, pas d'outils OS, pas de gestionnaire de paquets
# Seul le binaire compilé est présent
COPY --from=builder /app/api /api
# Exécuter en tant qu'utilisateur non-root (UID 65532 = nonroot)
USER 65532:65532
EXPOSE 8080
ENTRYPOINT ["/api"]
EOF

# Build et scan immédiat de l'image
docker build -t mon-api:latest .
trivy image --severity HIGH,CRITICAL mon-api:latest
# Résultat attendu avec distroless : 0 CVE (pas de packages OS vulnérables)

# Vérifier la taille de l'image (distroless vs alpine vs debian)
docker images mon-api:latest --format '{{.Size}}'
```

Runtime Security avec Falco : détecter les comportements anormaux des conteneurs

Falco (CNCF) est le standard de facto pour la détection des comportements anormaux en temps réel dans les environnements Docker et Kubernetes. Il analyse les appels système (syscalls) de chaque conteneur via eBPF et alerte sur les comportements suspects : écriture dans `/etc`, spawn

d'un shell dans un conteneur applicatif, lecture de fichiers de secrets, connexion réseau vers des IPs non autorisées.

```
 # Installation de Falco via Docker
docker run -d \
    --name falco \
    --privileged \
    -v /var/run/docker.sock:/host/var/run/docker.sock \
    -v /dev:/host/dev \
    -v /proc:/host/proc:ro \
    -v /boot:/host/boot:ro \
    -v /lib/modules:/host/lib/modules:ro \
    -v /usr:/host/usr:ro \
    falcosecurity/falco:latest

# Règle Falco personnalisée : détecter l'accès au socket Docker
cat > /etc/falco/custom_rules.yaml << 'EOF'
- rule: Docker Socket Access by Unexpected Container
  desc: Détecte l'accès au socket Docker depuis un conteneur non autorisé
  condition: >
    evt.type = open and
    fd.name = /var/run/docker.sock and
    container.name != socket-proxy and
    container.name != portainer and
    container.name != traefik
  output: >
    Accès socket Docker non autorisé
    (container=%container.name pid=%proc.pid cmd=%proc.cmdline)
  priority: CRITICAL
  tags: [docker, container-escape]
EOF

# Monitorer les alertes Falco en temps réel
docker logs -f falco 2>&1 | grep -E 'CRITICAL|WARNING'
```

La sécurisation complète des environnements Docker et Kubernetes repose sur plusieurs couches défensives complémentaires : Docker Socket Proxy pour le filtrage de l'API Docker, Vault pour les secrets dynamiques, Trivy pour le scan d'images, Falco pour la détection runtime, et les Network Policies pour l'isolation réseau. Pour comprendre les vecteurs d'attaque contre lesquels ces mécanismes protègent, l'article sur les techniques d'évasion Docker et containerd est essentiel. La sécurisation des secrets Vault en production s'inscrit dans la politique globale

anti-secrets sprawl. L'intégration de tous ces outils dans un cluster Kubernetes sécurisé est détaillée dans l'article sur le Kubernetes offensif (RBAC abuse) qui présente les vecteurs d'attaque à contre-mesurer.

La sécurisation du socket Docker et la gestion des secrets avec Vault constituent les deux piliers fondamentaux d'une infrastructure conteneurisée sécurisée en 2026. À mesure que les organisations adoptent des architectures microservices et des pipelines DevSecOps plus matures, ces bonnes pratiques deviennent des prérequis non négociables pour satisfaire les audits de conformité ISO 27001 (contrôle A.12.6 — gestion des vulnérabilités techniques) et les exigences NIS 2. La référence incontournable sur la sécurité des conteneurs reste le OWASP Docker Security Project, qui liste les 10 vulnérabilités Docker les plus critiques et les contre-mesures associées. Pour les environnements Kubernetes, le benchmark CIS Kubernetes Benchmark définit les niveaux de durcissement Level 1 et Level 2. L'ensemble des techniques présentées — Docker Socket Proxy, Vault, Docker Secrets, External Secrets Operator, Trivy, Falco — forment une stratégie de défense en profondeur qui réduit drastiquement les risques de container escape et d'exfiltration de secrets.

La convergence entre la sécurisation des conteneurs (Docker Socket Proxy, Vault, Falco) et les pratiques DevSecOps modernes (shift-left security, SCA, SAST) représente la meilleure stratégie pour réduire les risques dans les architectures cloud-native. Implémenter ces contrôles progressivement — en commençant par Docker Socket Proxy et le scan d'images Trivy, puis en ajoutant Vault et Falco — permet d'améliorer la posture de sécurité sans bloquer les équipes de développement. La sécurité des conteneurs s'inscrit dans une démarche DevSecOps continue, pas dans un projet one-shot.

