Breaking Bad Containers

Practical steps to secure your Container, Kubernetes workloads

Fredrik Steen

Fredrik Steen $> whoami

Director of Software Engineering

Varnish Software

Mail: fredrik.steen@varnish-software.com

LinkedIn: fredriksteen

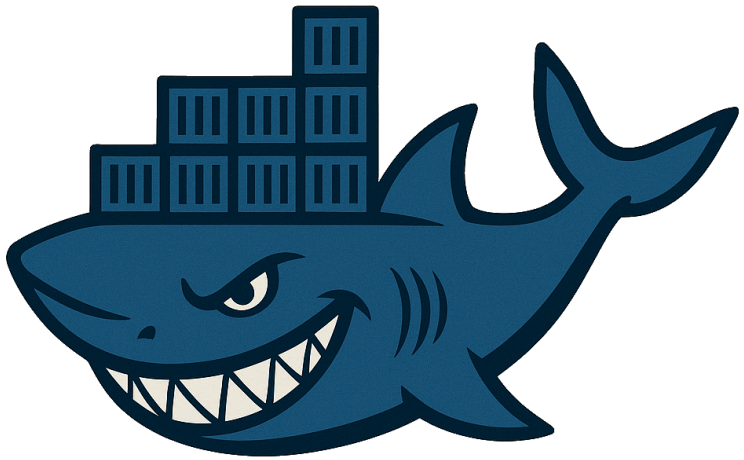
GitHub: stone

"Blog": https://tty.se/

Linux and Open Source user/developer since the mid 90s, Psytrance-DJ, Beekeeper, Electronics, Microbiology and Beer enthusiast.

Containers and Current Threats

- Of 6,292 images from 1,573 repos, ~61% had vulnerabilities in every tag (ICSME 2025)
- Significant portion built on outdated base images (Haque & Babar, 2021)
- **87% of production images still contain high or critical vulnerabilities** (Sysdig 2024)



Attack windows are shrinking to mere minutes

Attack windows are shrinking to mere minutes

- **AKS clusters face probing attempts within 18 minutes of deployment**
- **EKS clusters are targeted within 28 minutes**.

Supply chain attacks are rising sharply

- 2025-09-08: Hijacked 18 npm packages with 2B weekly downloads, planting malware to steal crypto by redirecting wallet transactions.
- **16,279 malicious packages discovered across npm, PyPI, Maven Central in Q2 2025** (Sonatype)
- The recent XZ Utils backdoor (CVE-2024-3094) demonstrated how "close" the entire Linux ecosystem came to widespread compromise.

IngressNightmare

(CVE-2025-1097, CVE-2025-1098..)

Disclosed in March 2025 affected vulnerable (nginx) admission controllers.

The most critical flaw allowed unauthenticated remote code execution (RCE) leading to complete cluster takeover, demonstrating how core Kubernetes components can become critical attack vectors.

This demands action and practical security implementations.

With some simple steps, you can significantly improve the security of your cluster and containerized applications.

It all start with a... `Dockerfile`

A simple `Dockerfile`

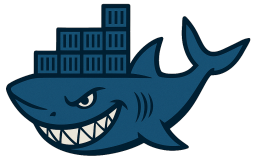
```
FROM ubuntu:latest
RUN apt-get update && apt-get install -y golang ca-certificates
USER root
WORKDIR /app
COPY . /app
RUN go build -o /app/app
ENTRYPOINT ["/app/app"]
EXPOSE 80
```

Let's analyze this simple Dockerfile...

A simple `Dockerfile` - what's wrong with it?

This simple Dockerfile illustrates common pitfalls:

- Unpinned builds (`:latest`) they change; CVE surface shifts daily.
- Running as `root` inside the container (CVE-2024-21626: container escape).
- `COPY` Pulls in everything from `.` (git history, tests, secrets..)
- Large Attack surface (unnecessary tools in final image).
- Ubuntu is a fat runtime, You can ship a Go app as a single binary.



Setuid root binaries - a classic privilege escalation vector

```
# Mount host /tmp into container
docker run -it --rm -v /tmp:/host-tmp ubuntu:24.04 bash

# Inside container as root:
cp /bin/bash /host-tmp/evil-bash
chmod 4755 /host-tmp/evil-bash  # Set setuid bit

# Exit container, then on host as regular user:
/tmp/evil-bash -p  # Now you have root shell on host!
```

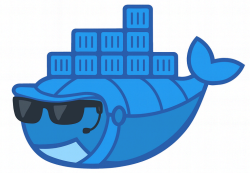
How to build secure container images (1/2)

```
FROM ubuntu:24.04 AS builder
ARG TARGETOS=linux
ARG TARGETARCH=amd64
ENV DEBIAN_FRONTEND=noninteractive
WORKDIR /src
COPY . /src
RUN set -eux; \
  apt-get update; \
  apt-get install -y --no-install-recommends ca-certificates golang; \
  rm -rf /var/lib/apt/lists/*; \
  CGO_ENABLED=0 GOOS=$TARGETOS GOARCH=$TARGETARCH go build -trimpath -ldflags="-s -w" -o /src/app

FROM gcr.io/distroless/static:nonroot AS runtime
WORKDIR /app
COPY --from=builder --chown=nonroot:nonroot /src/app /app/app
COPY --from=builder /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs/
USER nonroot:nonroot
EXPOSE 8080/tcp
ENTRYPOINT ["/app/app"]
```

How to build secure container images (2/2)

- Use a specific, up-to-date base image, ensure known CVE status
- Switch to a non-root user to limit privileges
- Use multi-stage builds, keep runtime clean
- Minimized attack surface: no shell, no package manager, only your binary (if possible)
- Use a .dockerignore file to exclude files (e.g., `.git`, `.env`, `*.key`, `id_rsa*`)
- Our example became a 7.7 MB final image where 5.1 MB is our static Go binary



Know your vulnerabilities - Trivy

Trivy has emerged as the de facto standard for container vulnerability scanning

```
$ docker run --rm -v /tmp/cache:/root/.cache/ aquasec/trivy:0.65.0 image python:2 --severity CRITICAL
$ trivy image $(docker ps --format "table {{.Image}}" | tail -n +2)
$ trivy k8s --include-namespaces kube-system --report summary
```

- Trivy scans for OS and language-specific vulnerabilities, misconfigurations, secrets, and generates SBOMs
- Integrates with CI/CD pipelines for automated scanning on build and deploy
- Use Trivy's severity filtering to prioritize remediation efforts on critical vulnerabilities

node:latest (debian 12.11)

Total: 26 (CRITICAL: 26)

Library	Vulnerability	Severity	Status	Installed Version	Fixed Version	Title
imagemagick	CVE-2025-53014	CRITICAL	affected	8:6.9.11.60+dfsg-1.6+deb12u3		ImageMagick: ImageMagick Heap Buffer Overflow https://avd.aquasec.com/nvd/cve-2025-53014
imagemagick-6-common						
imagemagick-6.q16						
libaom3	CVE-2023-6879			3.6.0-1+deb12u1		aom: heap-buffer-overflow on frame size change https://avd.aquasec.com/nvd/cve-2023-6879
libmagickcore-6-arch-config	CVE-2025-53014			8:6.9.11.60+dfsg-1.6+deb12u3		ImageMagick: ImageMagick Heap Buffer Overflow https://avd.aquasec.com/nvd/cve-2025-53014
libmagickcore-6-headers						
libmagickcore-6.q16-6						
libmagickcore-6.q16-6-extra						
libmagickcore-6.q16-dev						
libmagickcore-dev						
libmagickwand-6-headers						
libmagickwand-6.q16-6						
libmagickwand-6.q16-dev						
libmagickwand-dev						
libopenexr-3-1-30	CVE-2023-5841			3.1.5-5		OpenEXR: Heap Overflow in Scanline Deep Data Parsing https://avd.aquasec.com/nvd/cve-2023-5841
libopenexr-dev						
libsqlite3-0	CVE-2025-6965			3.40.1-2+deb12u1		sqlite: Integer Truncation in SQLite https://avd.aquasec.com/nvd/cve-2025-6965
	CVE-2025-7458					sqlite: SQLite integer overflow https://avd.aquasec.com/nvd/cve-2025-7458
libsqlite3-dev	CVE-2025-6965					sqlite: Integer Truncation in SQLite https://avd.aquasec.com/nvd/cve-2025-6965

Trivy Scan of `node:latest` (2025-09-01)

Immediate action plan for security improvement

(Yes, going nuts with ai generated images..)

Kubernetes

Mature ecosystem with many security tools and best practices available to make your container workloads secure.

How?

Level 1 - focus on foundational controls.

- Enable Pod Security Standards at the **Restricted level**
  - Privileged, Baseline, **Restricted** 🏆
- Deploy Falco for runtime monitoring (kernel-level syscall inspection)
- Implement default-deny network policies.
- Secure your cluster (API) access using firewall/VPN and use strong auth.

This will provide immediate protection against common attack vectors.

And insigts

Level 2 - Enhanced security with policy enforcement

- Admission control policies using OPA Gatekeeper
  - More flexible - Custom policies beyond standard security profiles
  - Cluster-wide enforcement - Not just namespace-level
  - Policy as Code - Version controlled, reviewable policies
- Mandatory image scanning and signing workflows of images (Cosign/Trivy)
- Least-privilege RBAC implementation.

A bit more effort, but will significantly enhance the security posture by enforcing security.

Boss Level - Security excellence with SLSA Level 2+ compliance

- Zero-trust network architecture (e.g., service mesh with mTLS)
- Advanced threat detection and automated response capabilities (e.g., Falco + SOAR)
- Comprehensive supply chain security attestation
- Train teams on security best practices and incident response
- Regular security audits and penetration testing

Exccellent security, but requires significant organizational commitment and resources.


In cybersecurity, you don't have to outrun the bear, you just have to be faster then the person next to you.
KEEP CALM

Kubernetes

Examples of how to harden your workloads

Admission control with OPA Gatekeeper

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sAllowedRepos
spec:
  parameters:
    repos:
    - "registry.company.com/"
```

- Enforce image policy on pod `spec` at admission time
- This example will block images from untrusted registries

PodSecurity Standards

```yaml
kind: Namespace
metadata:
  name: my-namespace
  labels:
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/warn: restricted
```

- Kubernetes built-in policies, set levels of security restrictions for Pods.
- API server will reject Pod creation/updates that violate the defined profile.

Pod securityContext - restricted policy

```
securityContext:
  runAsNonRoot: true
  allowPrivilegeEscalation: false
  readOnlyRootFilesystem: true
  capabilities:
    drop: ["ALL"]
```

- No root Containers
- No privilege escalation
- Read-only filesystem
- Drop all capabilities

Network Policies - deny by default

```
kind: NetworkPolicy
spec:
  podSelector: {}
  policyTypes: ["Ingress","Egress"]
  ingress:
    - from:
      - podSelector: { matchLabels: { role: "backend" } }
```

- Default deny all ingress and egress traffic
- Explicitly allow traffic only from/to trusted pods or namespaces

What about regular ol-Docker?

```
# Capability dropping and no-new-privileges
docker run --cap-drop=ALL --user 1000:1000 myapp:1.2.3
docker run --security-opt no-new-privileges myapp:1.2.3
# Network isolation
docker network create --internal restricted-net
docker run --network=restricted-net myapp
```

Q&A

- Thank you for your attention!
- Questions?