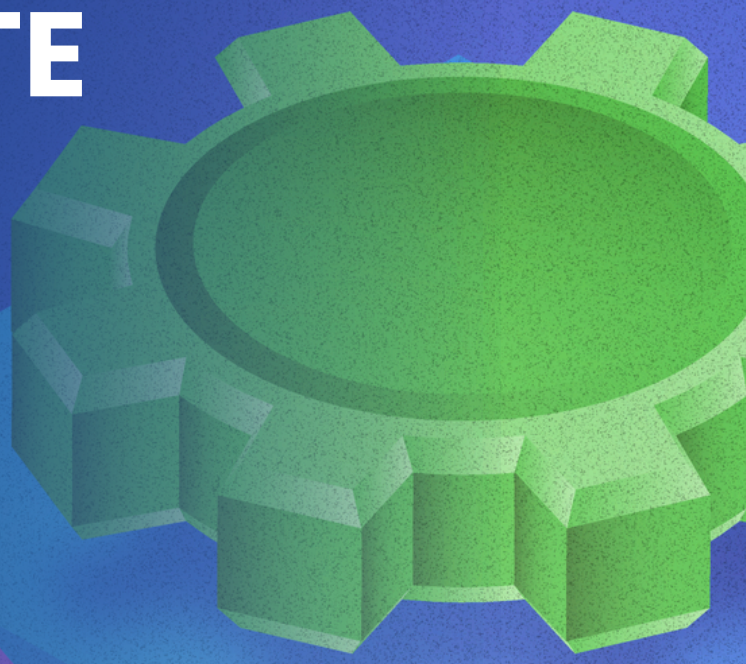
activestate

THE COMPLETE GUIDE TO CONTAINER SECURITY

Actionable steps to keep your images risk-free, secure, and compliant.



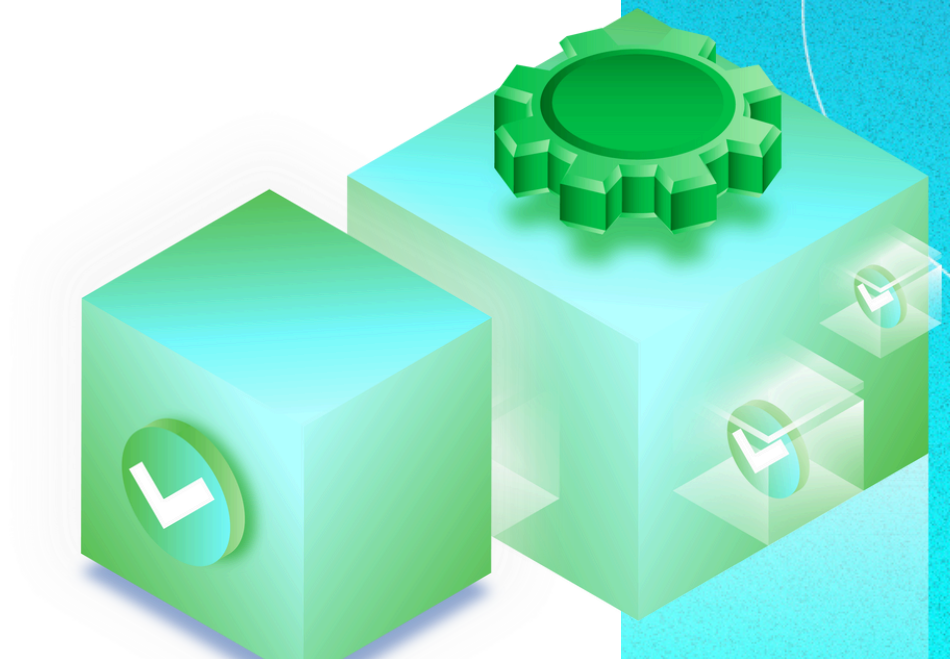
PART 02:

A Step-by-Step Approach to Container Security

By Eric Gallagher

Table of Contents

Missed Part 1 of this series? - Read it Here



01 Securing Container Images from the Start

Containers don't magically become secure just because you're using them. Security has to be built into the image from the first line of the Dockerfile to the final deployment tag.

This chapter will walk through actionable strategies for hardening container images. We'll focus on the build phase, where the majority of container security decisions are made and where the wrong choices can embed vulnerabilities, misconfigurations, and secrets into every environment you deploy to.

Why Image Hardening Matters

Every time you build a container image, you're packaging an entire execution environment, including the OS-level tools and libraries, language runtimes and interpreters, app dependencies, Config files, and startup behavior. The challenge? Most developers are focused on the **"will it run?"** problem, not the **"is it secure?"** problem. This makes the image build phase one of the most critical points of control in your entire software supply chain.

Use Minimal Base Images

Why it matters:

The larger the image, the larger the attack surface. Base images often include unused utilities, shells, compilers, and interpreters, many of which have known vulnerabilities or can be exploited during runtime.

What to do:

- Use **slim** or **distroless** images (`python:3.11-slim`, `gcr.io/distroless/python3`)
- Avoid `ubuntu:latest`, `debian:latest`, or any bloated base images unless necessary.

Even official base images can be outdated. Always scan them before use and verify how recently they were updated.

Distroless Images:

A distroless image is a container image with a minimal list of applications that also shares the host Linux kernel. Distroless container images Don't include a package manager, don't include a shell, and don't include a web client (such as curl or wget)

Run as a Non-Root User

By default, containers run as `root`. If an attacker breaks out of the container or if the container mounts sensitive host volumes, they now have root access on the host.

What to do:

- Create a non-root user during the build:
  - RUN useradd -m appuser
  - USER appuser
- Ensure your container image and orchestrator (e.g., Kubernetes) enforce this policy
- Explicitly drop unneeded capabilities and deny privilege escalation

Avoid Installing Unnecessary Packages

It's tempting to install utilities like `curl`, `vim`, or `git` during development. But leaving them in production containers can give attackers tools to move laterally, exfiltrate data, or escalate access.

What to do:

- Install only what you need, and remove what you don't
- Use multi-stage builds to separate build tools from the final runtime
- Strip out package managers (`apt`, `apk`, etc.) if not needed after install

Don't Bake Secrets into the Image

Developers sometimes hardcode secrets into config files or environment variables in Dockerfiles. Those secrets are then committed to Git, pushed to registries, and deployed to prod.

What to do:

- Use external secrets managers
- Never store API keys, database passwords, or SSH keys in the image
- Audit Dockerfiles and build scripts for hardcoded credentials

Lock Down Dependency Installation

Installing packages via `pip`, `npm`, or `apt` without version pinning or verification is a recipe for supply chain compromise.

What to do:

- Pin exact versions in `requirements.txt`, `package.json`, etc.
- Use hashes or checksums to validate downloads (e.g., `pip install --require-hashes`)
- Use tools like `poetry`, `pipenv`, `npm audit`, or `snyk` for additional safety
- Install from internal mirrors when possible to avoid relying on public registries

Scan Images at Build Time

Don't wait until an image is deployed to discover vulnerabilities. Scan it as part of the build pipeline.

What to check for:

- CVEs in OS packages
- Vulnerable app dependencies
- Secrets or credentials
- Known malware signatures

Fail builds when critical or high-severity vulnerabilities are flagged, not later in staging or prod.

Use Multi-Stage Builds

Multi-stage builds let you build your application in one image (with compilers, debuggers, etc. and copy only the necessary output to a clean, final image. This eliminates build-time bloat, sensitive files, and extraneous tools from the final image.

Sign and Verify Your Images

Anyone can upload an image with the same name and tag to a public registry. Without verification, you have no assurance it's from a trusted source.

What to do:

- Use Sigstore Cosign or Notary v2 to sign images at build time
- Configure your CI/CD pipeline to verify image signatures before deployment
- Prefer registries that support signature enforcement

This is foundational for software supply chain integrity.

Keep Images Up to Date

Once a container image is built, it's often forgotten. But the base image or app dependencies may accumulate known vulnerabilities over time.

What to do:

- Rebuild and rescan images regularly—even if your app code hasn't changed
- Subscribe to CVE feeds or use automated update tools (e.g., Renovate, Dependabot)
- Use image digests (`sha256:...`) instead of tags like `latest` for consistency

Takeaway: Build It Right, or Fix It Later (At a Higher Cost)

Bad Practice	Secure Alternative
Use bloated base image (ubuntu:latest)	Use minimal base (alpine, distroless)
Run as root	Create and run as non-root user
Install extra packages/tools	Strip everything non-essential
Store secrets in Dockerfile	Use external secrets management
Pull packages from public registries blindly	Pin versions, scan, and mirror trusted deps
Build once, deploy forever	Rebuild and rescan regularly

The earlier you catch security issues, the cheaper and easier they are to fix. Harden your container images before they ever leave your CI pipeline.

02 SBOMs, Provenance, and Dependency Hygiene

Containers are often treated like sealed boxes: you build them, tag them, push them to a registry, and hope for the best. But what's actually inside those boxes?
What packages? What licenses? What vulnerabilities? Who built it and when?
If you don't have a clear record of what goes into your containers and where each piece comes from, then your software supply chain is open to exploitation.

This chapter covers three foundational practices for container transparency and trust

- **Software Bill of Materials (SBOMs)**
- **Provenance tracking**
- **Dependency hygiene**

Together, they help teams identify, audit, and control the third-party code and tools embedded in containers before attackers do.

What Is an SBOM (Software Bill of Materials)?

An SBOM is like an ingredient label for your software. It lists:

- All components and dependencies
- Their versions
- Their origins (e.g., PyPI, npm)
- Licensing information
- Known vulnerabilities (if available)

In the container world, SBOMs can describe everything from the OS packages in your base image to the transitive libraries pulled in by your application code.

Why SBOMs Matter

- Regulatory mandates (e.g., U.S. Executive Order 14028) now require SBOMs for federal contractors
- They accelerate incident response (e.g., during Log4Shell or XZ backdoor events)
- They're critical for vulnerability scanning, license compliance, and third-party risk management

Without an SBOM, you're left asking, *"Are we exposed?"* Without a clear answer.

How to Generate SBOMs for Containers

Most modern container security tools can generate SBOMs as part of the build or scan process.

Popular Tools:

- **Syft** – Generates SBOMs in SPDX, CycloneDX, or JSON formats
- **Trivy** – Combines scanning and SBOM generation
- **Docker Scout** – Provides component insights in CI
- **Grype** – Can consume SBOMs for vulnerability detection

Tools for Provenance and Trust

Sigstore (Cosign + Rekor + Fulcio)

- **Cosign** signs container images
- **Rekor** stores immutable build logs
- **Fulcio** issues short-lived signing certificates based on identity

Sigstore enables a fully open-source, verifiable trust model for software artifacts.

In-Toto and SLSA (Supply-chain Levels for Software Artifacts)

- **In-Toto** defines cryptographic policies for what's allowed in a build
- **SLSA** defines maturity levels for software supply chain integrity

Adopting these frameworks can drastically reduce your exposure to compromised build systems, rogue contributors, and tampered binaries.

Dependency Hygiene: What It Is and Why It's Critical

Dependency hygiene is the practice of intentionally managing, auditing, and minimizing the third-party code your application depends on.

Why It Matters:

- Every dependency introduces potential vulnerabilities, license risk, and bloat
- Most software exploits today involve known vulnerabilities in third-party packages
- Transitive dependencies (indirect imports) are often untracked and unmonitored

Symptoms of Poor Hygiene:

- Floating versions (`>=, ^, latest`)
- Hundreds of indirect packages installed without review
- Abandoned or unmaintained dependencies
- Unknown or incompatible software licenses

Improving Dependency Hygiene in Containers

Pin Everything

Use fixed versions (e.g., `==2.1.3`) in `requirements.txt, package-lock.json`, etc.
This creates repeatable builds and predictable risk.

Audit Dependencies Regularly

Use tools like:

- `pip-audit`
- `npm audit`
- OWASP Dependency-Check
- GitHub Dependabot

Scan both direct and transitive dependencies remove what you don't need

- Don't include dev/test tools in production containers
- Prune unused packages and legacy libraries
- Use `multi-stage builds` to discard build-time dependencies

Use Trusted Sources

Install from:

- Vetted internal mirrors
- Official, verified repositories
- Git commits with verifiable signatures

Avoid pulling directly from GitHub branches or user-contributed packages without scrutiny.

Monitor for License Conflicts

Track:

- GPL, AGPL, and other "viral" licenses
- Commercial license restrictions
- License incompatibility across packages

This is critical for legal compliance in regulated or commercial software.

SBOMs + Provenance + Hygiene = Trusted Containers

Practice	Focus	Benefit
SBOM	What's in the image	Auditability, CVE tracking
Provenance	Where it came from	Build integrity, trust validation
Dependency Hygiene	How well it's maintained	Fewer bugs, risks, and surprises

When done right, they form a self-reinforcing chain of custody for your software—from source to prod.

Takeaway: Transparency Is the First Step to Trust

Blindly trusting what's inside your containers is no longer an option. If you're serious about security, visibility must come first.

03 Secure Builds and CI/CD Pipelines

The CI/CD pipeline is where containers are born and often where security gets left behind. Modern DevOps workflows prioritize speed: build fast, ship often, fail forward. But without security controls baked into the CI/CD process, teams unintentionally automate the introduction of vulnerabilities, secrets, and unverified code into production environments.

In this chapter, we'll show you how to shift container security left, embedding controls directly into the build and deployment stages. The goal: move fast without breaking trust.

Why Securing the Pipeline Matters

Every artifact that flows through your CI/CD pipeline eventually ends up in production. If attackers compromise the pipeline or if unsafe components are allowed through unchecked, they don't need to break into your servers. They're already inside the supply chain.

Notable Supply Chain Attacks:

- **SolarWinds (2020):** Attackers inserted malicious code during build time
- **Codecov (2021):** CI scripts leaked credentials, allowing attackers to modify code and gain access to private repos
- **CircleCI (2022):** Compromised credentials used to exfiltrate data via poisoned builds

Principles of Secure CI/CD

Principle	What It Means
Shift Left	Move security checks earlier in the development lifecycle
Fail Fast	Block risky builds before they reach staging or prod
Immutable Builds	Never mutate existing artifacts—rebuild and redeploy
Least Privilege	Grant only the minimal access necessary at each stage
Trust Nothing by Default	All code, dependencies, and artifacts must be verified

Locking Down the Build Process

Use Dedicated, Isolated Build Runners

- Avoid shared runners or self-hosted agents with open internet access
- Use short-lived runners that terminate after each build
- Isolate secrets and sensitive data from build logs

Use Infrastructure as Code (IaC)

- Version control everything: build configs, pipelines, deployment manifests
- Use tools like Terraform, Pulumi, or Helm for repeatable, auditable environments

Pin Tools and Dependencies

- Use locked versions of build tools and language packages
- Avoid installing from external sources during builds (e.g., `curl https://... | bash`)

Embed Security into the Pipeline

Static Code Analysis (SAST)

- Run linters and vulnerability scanners on application code
- Check for hardcoded secrets, insecure function usage, or policy violations

Dependency Scanning (SCA)

- Scan for vulnerable packages during build time using
  - Snyk
  - Trivy
  - GitHub Dependabot
  - OWASP Dependency-Check

Image Scanning

- Automatically scan every built image for:
  - CVEs in OS packages
  - Embedded secrets
  - Unapproved base images
  - License violations

SBOM Generation

- Generate a Software Bill of Materials during every build
- Store SBOMs alongside artifacts and sign them when possible

Secrets Management in CI/CD

Secrets often leak through CI logs, misconfigured environments, or hardcoded values.

Best Practices:

- Use dedicated secrets managers (e.g., Vault, AWS Secrets Manager, Doppler)
- Never store secrets in Git or environment files
- Mask secrets in CI logs
- Rotate secrets regularly and revoke unused ones

Pro tip: Scan your repos regularly with tools like `git-secrets`, TruffleHog, or Gitleaks to catch leaked credentials early.

Enforce Policies Automatically

Use policy-as-code tools to define and enforce rules across your pipeline.

Tools:

- **Open Policy Agent (OPA) / Conftest** Define guardrails for Dockerfiles, Kubernetes manifests, and Terraform configs
- **Kyverno** – Kubernetes-native policy engine for enforcing security controls
- **Checkov / tfsec** – Static analysis for infrastructure-as-code

Common Policies:

- Block containers that:
  - Run as root
  - Include vulnerable packages
  - Lack SBOMs or signatures
  - Use banned base images
- Enforce image provenance and signature verification before deployment

Policies remove subjectivity—if a build violates security requirements, it fails.

Signed Builds and Provenance Verification

To secure your pipeline against tampering and impersonation, you need cryptographic proof of who built it, when it was built, and what inputs were used.

Tools to Use:

- **Sigstore Cosign** – Sign and verify container images
- **Rekor** – Immutable transparency log for build metadata
- **SLSA (Supply-chain Levels for Software Artifacts)** – Defines build integrity levels

By signing and verifying images and SBOMs, you ensure every artifact is traceable and trustworthy.

Continuous Monitoring and Feedback Loops

Even after securing the pipeline, ongoing feedback is critical:

- **Alert on new vulnerabilities** found in previously safe images
- **Rebuild and redeploy** containers when base image CVEs are disclosed
- **Track drift** between source, image, and deployment environments

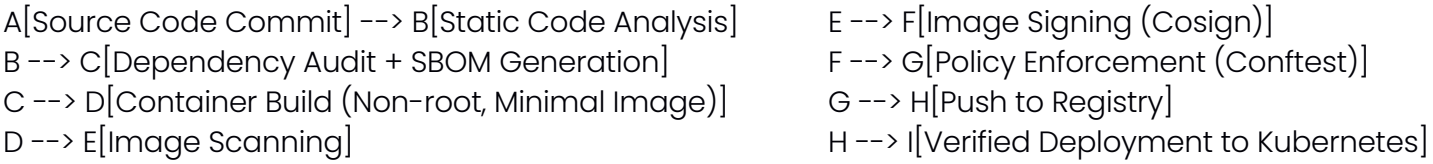
Integrate tools like:

- **Docker Scout, Trivy,** or **Grype** for image monitoring
- **Kubernetes admission controllers** to enforce deployment-time policies
- **Dashboards and Slack alerts** for build failures or security warnings

The goal is to make security feedback visible, fast, and actionable.

Example: A Secure CI/CD Pipeline

Every stage includes checks and balances. Failures block progress, and signed metadata provides an audit trail.



Takeaway:
Security Should Be a Gate, Not a Speed Bump

Weak Build Practice	Secure Alternative
Pulling from public sources mid-build	Use vetted, mirrored dependencies
Running builds on shared agents	Use ephemeral, isolated runners
Skipping image scans	Automate scanning in CI
Hardcoded secrets in configs	Inject secrets securely at runtime
Manually reviewing security post-deploy	Fail the build when risks are detected

When done right, security accelerates delivery by catching problems early, ensuring clean releases, and building trust across the stack.

04 Runtime Security and Container Hygiene

Once a container is running, your controls don't stop; they evolve. The runtime environment is where real-world behavior emerges: network traffic, process execution, file access, and inter-container communication. If you're not watching closely, attackers can exploit blind spots, escalate privileges, and persist without detection.

This chapter focuses on what happens after deployment, and how to continuously monitor, restrict, and respond to suspicious behavior in live environments.

The Case for Runtime Controls

Even perfectly built containers are exposed at runtime due to:

- Host kernel vulnerabilities
- Overly permissive container permissions
- Insecure networking between services
- Unexpected (or malicious) process behavior
- Human error in deployment configuration

Runtime security is about **detecting and responding to issues that slip past your shift-left efforts**, because something always does.

Least Privilege and Capability Restriction

Containers often run with more privileges than necessary. This violates the principle of least privilege and dramatically expands the blast radius of a compromise.

What to Do:

Drop all capabilities by default using:

```
securityContext:
  capabilities:
    drop:
      - ALL
```

What to Do:

- Explicitly add only the minimal set needed (e.g., NET_BIND_SERVICE for web apps)
- Set `allowPrivilegeEscalation: false` and `readOnlyRootFilesystem: true`
- Avoid `privileged: true` containers unless absolutely necessary (they get near-full host access)
- Use Linux Security Modules (LSMs) like **AppArmor, SELinux,** or **Seccomp** to sandbox and limit syscalls

These settings can often be applied declaratively in Kubernetes via PodSecurityContexts or enforced with admission controllers.

Control Network Exposure

Containers are often overly exposed internally and externally.

What to Do:

- Use **Kubernetes Network Policies** or **CNI plugins** (e.g., Calico) to restrict traffic between pods
- Limit ingress/egress with firewall rules or service mesh configuration
- Avoid exposing ports like SSH, Redis, or internal APIs to the public internet
- Apply **zero trust networking** principles: no implicit trust between services, even within the same cluster

By default, many orchestrators assume open communication between pods. You must define what's not allowed.

Monitor Runtime Behavior

Runtime monitoring detects behaviors that deviate from expected patterns—whether caused by attacks, misconfigurations, or bugs.

Tools to Use:

- **Falco** (CNCF): Monitors container syscalls in real time and alerts on suspicious behavior
- **Sysdig Secure, Datadog, Aqua Security, Prisma Cloud,** or **Wiz:** Full-featured runtime security platforms
- **eBPF-based tools** (like Cilium Tetragon): Lightweight kernel-level monitoring

Common Detection Rules:

- Unexpected file writes to `/etc`, `/bin`, or `/root`
- Shells spawning inside containers (`sh`, `bash`)
- DNS tunneling or outbound connections to known bad IPs
- New binaries appearing in the container after startup
- High CPU or memory usage anomalies (e.g., crypto mining)

Monitoring is not just about catching attackers; it's about enforcing expected behavior.

Implement Runtime Admission Controls

Admission controllers validate and mutate workloads before the Kubernetes API server accepts them.

Use them to:

- Reject containers that run as root
- Enforce the use of signed images
- Block use of specific base images
- Require labels, annotations, or compliance tags

These controls act as a "runtime firewall" at the deployment layer, keeping bad workloads from ever starting.

Reduce the Attack Surface with Runtime Hardening

Run Minimal Containers

- Fewer packages = fewer vulnerabilities
- Distroless or scratch images prevent shell access and reduce exploitability

Use Read-Only File Systems

- Prevents attackers from writing scripts, binaries, or persistence mechanisms

Avoid Host Mounts

- Don't mount. `/var/run/docker.sock` or `/etc` from the host
- Avoid writing logs or temp data directly to host volumes

Set Resource Limits

- Define `cpu`, `memory`, and `ephemeral-storage` requests/limits
- Prevents DoS from runaway processes

Incident Response in a Containerized World

Incident response must evolve with containers. When a compromise occurs:

- Identify the container ID and image tag
- Pull the SBOM to identify impacted packages
- Trace image provenance and build history
- Correlate activity with logs from your runtime monitoring tools
- Terminate and redeploy from a clean, verified image

Containers are ephemeral by design, so your IR playbook should be too:

- Avoid patching live containers
- Rebuild and redeploy from secure sources
- Automate quarantine of compromised pods using orchestration or network segmentation

Continuous Compliance and Reporting

In regulated environments, runtime hygiene isn't just best practice; it's mandatory.

What to Audit:

- Running containers vs. allowed images
- Security context configurations
- Container-to-container traffic logs
- Image signatures and SBOM presence
- Runtime alerts and anomaly trends

Tools for Compliance:

- **OpenSCAP** and **CIS Benchmarks**
- **Kube-bench, Kube-hunter**
- **Compliance dashboards** in Prisma Cloud, Aqua, and Wiz

Keep records of: What ran, who deployed it, how long it ran, and what it accessed

Takeaway:
A Secure Build Is Useless If Runtime Is a Free-For-All

Runtime Risk	Mitigation Strategy
Privilege escalation	Drop capabilities, avoid root
Container breakout	Use AppArmor/Seccomp, restrict host mounts
Network pivoting	Apply NetworkPolicies, enforce least-access
Malware injection	Read-only filesystems, runtime monitoring
Undetected persistence	Ephemeral containers, immutable infrastructure

Up Next:
Scale and futureproof your container security strategy

Now that we've secured builds and runtime, in Part 3 we'll zoom in and look at how to enforce these practices at scale using governance frameworks, compliance requirements, and policy-as-code.

If you enjoyed this eBook and want bite-sized updates on all things related to software supply chain security, please subscribe to my newsletter - "Securing the Backbone,"

[Read Part 3 Now]

activestate

Skip the Checklists With ActiveState Secure Containers

If you want to skip the container security checklists and start realizing the benefits of fully secured containers today. ActiveState is here to help.

ActiveState Secure Containers provides fully customizable, regularly updated, low-to-no CVE container images for any use case. No matter what your application specifications are, our team will ensure that every layer, from the OS to application dependencies, is fully secured and ready for deployment.

Drop us a line today, and try your first build for free!