activestate

THE COMPLETE GUIDE TO CONTAINER SECURITY

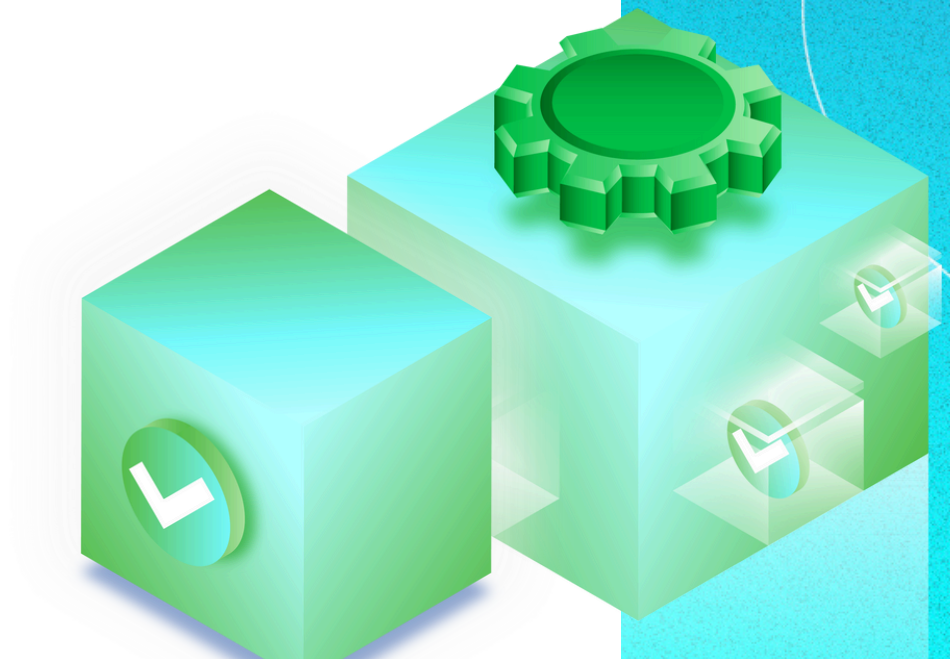
Actionable steps to keep your images risk-free, secure, and compliant.

PART 01:

Why You Need a Strategy For Secure Containers

By Eric Gallagher

Table of Contents



01 A Brief History of Containers

The Rise of Containers

The concept of isolating computing environments isn't new. Over the past several decades, we've seen everything from segmenting resources on mainframe systems to the rise of virtualization, cloud computing, and ultimately containers.

Unlike the resource-intensive VMs before them, containers offered a variety of benefits that sent their popularity into the stratosphere. They're lightweight, fast to spin up, and highly portable. Three qualities that align perfectly with the speed and scale of modern software development.

While Docker sparked the container revolution, by 2015, the ecosystem had exploded. Public registries were growing, CI/CD pipelines were retooled, and orchestration tools such as Kubernetes turned containers into a platform for building cloud-native apps.

Shifting from VMs to Containers

VMs	Containers
• Heavyweight, full OS • Start in minutes • High resource usage	• Lightweight, Share OS • Start in seconds • Low resource usage

Part of this rapid growth was due to containers' natural fit for DevOps practices. Containers didn't just complement DevOps; they helped accelerate and operationalize it at scale. On the operations side, containers enable:

- Predictable deployments
- Easier rollback and versioning
- Better utilization of compute resources

On the development side, they enable:

- Faster feedback loops
- Environment parity
- Dependency encapsulation

Of course, these benefits come with a catch. The same traits that make containers so attractive also make them easier to misuse and more complicated to secure. Public images pulled without vetting can include outdated libraries or malware. CI/CD pipelines can accidentally bake in secrets. Containers running as root can escalate privileges. Teams often fail to update images regularly, leaving known vulnerabilities exposed in production.

02 What's in a Software Container?

Containers may feel like magic boxes that run applications, but beneath the surface, they're composed of well-defined layers and components. Understanding the components of a container and its movement through the software lifecycle is the foundation for securing it.

In this chapter, we'll break down the anatomy of a container, demystify how containers are built and run, and introduce the components of containerized infrastructure. If you're new to containers, this is your blueprint. If you're already working with them, this is your checklist for where to embed security.

What Is a Container?

At its core, a container is a lightweight, portable executable package that includes:

- An application or service
- All its dependencies (e.g., libraries, configuration files, binaries)
- Instructions to run the application

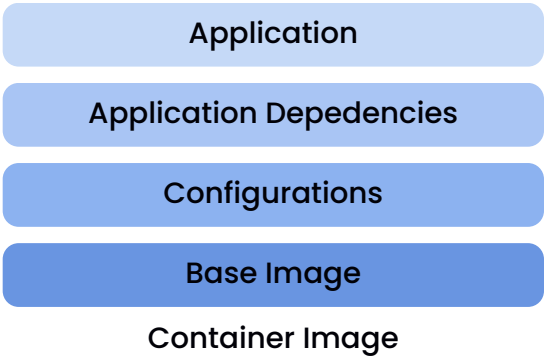
But containers don't virtualize hardware like VMs. Instead, they leverage the host's operating system kernel and isolate processes using Linux features like:

- **Namespaces** – isolate processes, network interfaces, and file systems
- **cgroups (control groups)** – limit resource usage (CPU, memory, I/O)
- **Union file systems** – layer multiple filesystems for image efficiency

These features mean multiple containers can share the same kernel but still run independently and securely, at least in theory. That shared-kernel model introduces performance gains and security trade-offs.

The Layers of a Container Image

A container image is built in **layers**, each representing a step in the build process (typically defined in a `Dockerfile`). Understanding this layered structure is crucial for both efficiency and security.



Example Dockerfile Layers in a Python Web App Container:

1. **Base Image**
   `FROM python:3.11-slim`
   Starts with a Python base, which itself may be based on Debian, Alpine, or an alternative minimal OS.

2. **System Packages**
   `RUN apt-get update && apt-get install -y curl build-essential`
   Adds OS-level tools and compilers—can introduce CVEs if not managed carefully.

3. **Application Code**
   `COPY . /app`
   Injects your app into the image. Secrets or sensitive files often leak here.

4. **Python Dependencies**
   `RUN pip install -r requirements.txt`
   Pulls open-source libraries from PyPI—potentially vulnerable or malicious.

5. **Entrypoint**
   `CMD ["gunicorn", "main:app"]`
   Defines how the container runs when started.

Each layer is cached and can be reused across builds. But every layer is also an **attack surface**, and vulnerabilities in any layer can compromise the entire container.

The Container Lifecycle: From Dev to Prod

Containers don't live in isolation. They're part of a broader software delivery lifecycle that stretches from the developer's laptop to production. Here's how it typically flows:

Step 1: Development

- Developers write code and define a `Dockerfile`
- Build and test containers locally using `docker build`
- Common risks: unchecked dependencies, secrets in image, outdated base images

Step 2: CI/CD Integration

- Container builds triggered automatically on code push
- Static scanning, unit testing, linting
- Image gets tagged and pushed to a registry

Step 3: Registry Storage

- Public (e.g., Docker Hub) or private (e.g., Harbor, ECR, Artifactory) image storage
- Access-controlled, versioned, and potentially signed
- Common risks: unverified images, lack of immutability, outdated tags

Step 4: Deployment

- Images deployed into dev/staging/prod via Kubernetes, ECS, Nomad, etc.
- Helm charts or manifests define replicas, networking, storage, secrets
- Risks here include misconfigured security contexts, lack of runtime policies

Step 5: Runtime

- Containers execute as isolated processes managed by an orchestrator
- Runtime tools monitor performance, logs, and behavior
- Common risks: privilege escalation, container escapes, insecure network exposure

Each phase introduces different threat vectors—and each is an opportunity to embed security checks.

Orchestration: Managing Containers at Scale

Running a few containers on a laptop is easy. Running thousands across clusters is not. Initially developed by Google and open-sourced in 2014, Kubernetes has become the de facto tool for container orchestration. It handles scheduling, scaling, and managing container lifecycles across clusters of machines. Kubernetes helped organizations move from running containers on single hosts to managing fleets of services distributed across hybrid cloud environments.

Kubernetes is powerful, but complex! Misconfigurations are common, and default settings often prioritize usability over security.

The Hidden Layers: Infrastructure, Storage, and Networking

Containers don't operate in a vacuum. Behind every container is a stack of underlying infrastructure:

- **Container Runtime:** Docker, containerd, CRI-O
- **Operating System:** Linux distro (e.g., Ubuntu, Alpine, Red Hat)
- **Cloud Platform:** AWS, Azure, GCP, or on-prem infrastructure

- **Networking Stack:** CNI plugins like Calico, Flannel, or Cilium
- **Storage:** Mounted volumes, persistent volumes (PVs), or object stores

Every part of this stack introduces potential misconfigurations, outdated components, or insufficient access controls. Container security must be holistic, not just focused on the image itself.

Who Owns What in a Containerized World?

In traditional infrastructure, security responsibility was clear. Operations secured servers, developers wrote code, and security teams reviewed configs and access. In a containerized world, lines blur. Developers write code and Dockerfiles, Operations manages orchestrators and pipelines for builds and deployments, and Security must plug into all of it.

This shared responsibility model requires alignment between teams. Otherwise, gaps form and attackers find them first.

Takeaways: Mapping the Stack to the Attack Surface

Layer	Risk Category	Example Threat
Base image	Vulnerabilities, outdated OS	CVEs in Ubuntu, Alpine, etc.
Application code	Secrets, logic bugs	Hardcoded API keys, RCE flaws
Open-source dependencies	Malicious or outdated packages	Typosquatting, vulnerable libraries
CI/CD pipeline	Supply chain attacks	Compromised build tools or scripts
Registry	Tampering, unverified sources	Unsigned images, lack of RBAC
Kubernetes config	Privilege escalation, exposure	Containers running as root, open ports
Runtime	Lateral movement, persistence	Container escape, rootkit persistence

03 Containers & The Illusion of Safety

Containers are fast, efficient, and portable, which is precisely why they've become so popular. But with their rise came a false sense of security. Many teams believe that simply by containerizing an application, they've inherently made it safer.

In this chapter, we break down where this "illusion of safety" comes from and what makes containers vulnerable in real-world environments.

The "Containers Are Isolated" Myth

Containers are sometimes described as "lightweight VMs," a term that's not just misleading but dangerous. Unlike VMs, containers share the host OS kernel and run as regular processes on the host, allowing them to interact with host resources more easily than assumed. These features mean that a vulnerability inside a container can impact the host system and other containers if not properly isolated.

Consider:

- If a container runs as root and the host lacks proper security controls, that container has the same privileges as root on the host.
- Containers can escape through kernel exploits, misconfigured namespaces, or privileged modes.

Containers are *process isolation*, not hardware isolation. They require deliberate hardening both at build time and at runtime.

Security by Default? Not Quite.

Many developers also assume that containers are secure by default. Yet, many images found on public repositories such as Docker Hub, even official images, have their own set of challenges. Containers often run as root, base images usually contain outdated packages, and unwanted capabilities may be enabled, giving unnecessary access to low-level system calls. Security in containers is configurable, not guaranteed.

The Shared Kernel Problem

Containers are lightweight because they share the same Linux kernel on a given host. However, this also means:

- Any container can exploit a vulnerability in the kernel.
- Compromised containers may affect other workloads or the host itself.
- Kernel updates require coordination and sometimes reboots, meaning many production clusters run outdated, vulnerable kernels for long periods.

The shared kernel is the Achilles' heel of container isolation. It's why some enterprises isolate sensitive workloads on dedicated nodes or opt for microVMs (like Firecracker) to strike a better balance between speed and isolation.

Misconfigurations: The Real Attack Surface

According to multiple industry studies, misconfiguration is the #1 cause of security incidents in containerized environments. Some of the most common misconfigurations include:

- Containers running in privileged mode
- Lack of resource limits, leading to DoS vulnerabilities
- Exposing container ports directly to the internet
- Mounting the host file system into a container (`/var/run/docker.sock`)
- Weak or missing network policies, allowing lateral movement

Misconfigurations aren't just risky, they're exploitable. And they often slip through the cracks because security and DevOps teams operate on different assumptions and priorities.

The Inheritance Problem: Trusting What You Pull

Container images often start from base images pulled from public registries. Many teams assume that "official" means safe, but even images from verified sources can contain:

- Outdated software with known CVEs
- Bloated packages that introduce unnecessary risk
- A lack of clarity and provenance, or understanding who published the image

Each layer of a container image inherits whatever's beneath it. So if your base image contains `openssl` with a critical CVE, your application does, even if you never directly used it. Without tools to analyze and scan these layers, you're essentially building on unverified infrastructure.

Runtime Risks: What Happens After the Container Starts

Even if a container is built securely, it can still become a liability once deployed. Common runtime risks include:

- Container escapes using kernel exploits
- Lack of syscall filtering, allowing dangerous system calls
- Log injection or abuse of shared volumes
- Sensitive data exfiltration due to a lack of egress controls

Worse still, many environments lack real-time monitoring for container behavior. Once a container is running, it's often invisible to traditional security tools, making runtime observability and anomaly detection tools critical to closing the gap.

Overconfidence in Orchestration Platforms

Many teams believe Kubernetes will "handle security for them." While Kubernetes does offer some security primitives (e.g., role-based access control, network policies, pod security contexts), it's typically complex to configure securely, insecure by default in many areas, and often misused or bypassed by over-permissive roles and weak controls.

Security in Kubernetes is not plug-and-play. Without defined policies, continuous monitoring, and hardened configurations, it's easy to introduce new vulnerabilities at the orchestration level.

Real-World Incidents:

Tesla (2018): Attackers gained access to an open Kubernetes dashboard, deployed malicious containers, and used them for crypto mining.

Alpine Linux Base Image (2019): A widely-used version had an empty root password, opening the door for privilege escalation if misused.

Docker Hub Breach (2019): Credentials for ~190,000 accounts were compromised, impacting image integrity and developer trust.

Takeaway:
Containers Are Not Secure by Default

Containers can be secure, but only with intention, visibility, and discipline.

Myth	Reality
Containers are isolated	Only process-level isolation—shared kernel creates risk
"Official" images are safe	Many contain outdated packages or unknown provenance
Orchestration handles security	Kubernetes is powerful, but not secure out of the box
Runtime risks are minimal	Container escapes, privilege escalation, and data exfiltration happen in the wild

04 The Trouble With Public Container Registries

Public container registries are a critical part of the open-source ecosystem. But they're also one of the most overlooked and ungoverned parts of the modern software supply chain. This chapter explores how these registries work, the risks of unquestioningly trusting them, and what you can do to pull safely.

What Is a Container Registry?

A container registry is a storage and distribution system for container images. Registries serve as centralized hubs where developers and teams can push built container images, version and tag them, and pull images into their build or runtime environments.

Types of Registries

- **Common Public Registries**
  - Docker Hub
  - GitHub Container Registry
- **Private Registries**
  - Amazon ECR
  - Google Artifact Registry
  - Harbor (self-hosted)
  - JFrog Artifactory

Because anyone can publish images, public registries democratize container development. But they also create a massive risk.

The Popularity (and Risk) of Docker Hub

Docker Hub is the default registry used by Docker. It hosts millions of images, a mix of "official," "verified publisher," and "community" content, and a large portion of the internet's containerized applications. But popularity doesn't equal safety.

Docker Hub Red Flags

- No enforced security scanning for many images
- No requirement to patch or maintain images
- No validation of image provenance
- High prevalence of outdated and vulnerable packages

Anyone can upload an image called `nginx:latest`.
Unless it's pulled from the verified `library/nginx` repo, you might be running *anything*.

Case Study: Attacks via Public Registries

Attackers know public registries are a distribution goldmine. Here are real-world examples of how they've exploited that:

Typosquatting Attacks

Bad actors upload images with names similar to popular ones. These malicious images often contain crypto miners, reverse shells, or data exfiltration payloads.

- `python3.11-silm` instead of `python3.11-sli`
- `alpine3.17` vs. `alpine:3.17`

Imagine a developer wants to install the well-known Python library for HTTP requests:

"pip install requests"

But they accidentally type: "pip install request"

That small typo (dropping the "s") leads to a completely different package. In the real world, attackers have uploaded similarly named packages like:

- `request`
- `requsets`
- `reqeusts`

These malicious packages often mimic the real library's functionality just enough to seem legit but contain hidden code that exfiltrates environment variables, steals API keys, or installs backdoors. In many cases, these fake packages use obfuscated code to delay detection and blend into production environments where the name is rarely rechecked.

Typosquatting on PyPI is real, active, and dangerous. To avoid issues, teams should:

- Use dependency locking (`requirements.txt, pip-tools`)
- Scan dependencies regularly for known malicious packages
- Only install from known-good sources and repositories

Malicious Dependency Injection

In some cases, attackers create base images that appear legitimate but contain preloaded, backdoored dependencies. Some examples include: modified `pip` or `npm` to install extra malicious packages, pre-baked SSH keys or hardcoded credentials, and tools to scrape environment variables at runtime.

Recent Incidents:

- **2021:** Aqua Security found over 4,000 images on Docker Hub with malware, secrets, or crypto miners.
- **2023:** Dozens of PyTorch and Node.js devs downloaded images containing covert data-stealing tools via typosquatted containers.

These aren't edge cases; they're part of a growing trend of supply chain attacks through poisoned containers.

Trust, But Rarely Verify: The Developer Blind Spot

One of the most dangerous practices in modern DevOps occurs when pulling containers directly from Docker Hub into production. While it is fast and convenient, unless you scan the image, verify its source, track its version, and monitor for updates, you've just introduced an untrusted, unmonitored binary blob into your environment. In any other software process, this would be unthinkable. But with containers, it's often business as usual. Public registries offer no guarantees, only options. If your team doesn't use those options proactively, you're operating on faith.

Security Shortcomings of Public Registries

Security Concern	Why It's a Problem
No mandatory scanning	Images with known CVEs often remain live for months or years
No signed images by default	You can't prove who built it or if it was tampered with
No update notifications	Outdated images silently accumulate vulnerabilities
No required maintainer identity	Attackers can impersonate trusted sources
Tag confusion (`latest`)	Tags like `latest` can change content unexpectedly

Best Practices for Pulling from Public Registries

Pulling from public sources isn't inherently bad; it just requires rigor. If you must pull, do it safely. Use only verified publishers (such as ActiveState) or official images.

- **Pin image versions**
  Use exact tags like `nginx:1.25.2-alpine`, not `nginx:latest`.
- **Scan every image**
  Use tools like: Trivy, Grype, Snyk, and Docker Scout
- **Avoid overly large base images**
  Choose minimal images (e.g., `alpine, distroless`) to reduce attack surface.
- **Limit runtime privileges**
  Ensure containers do not run as root or with unnecessary capabilities.
- **Mirror and lock down trusted images**
  Pull and store trusted images in an internal registry where you control access and updates.
- **Track provenance and generate SBOMs**
  Know what's in your images and who put it there.

Building a "Zero-Trust" Container Pipeline

Instead of relying on the trustworthiness of public registries, flip the model. Assume every image is untrustworthy until proven safe.

- **Ingest only through vetted CI pipelines**
- **Scan at build and deploy time**
- **Sign and verify images with tools like Cosign**
- **Enforce policies via OPA or Kyverno**
- **Use hardened base images from trusted sources**

Takeaway:
Convenience Shouldn't Trump Security

Security doesn't mean you can't move fast. But it does mean you can't cut corners. Public registries are a powerful tool, but also an attractive attack vector. Without verification, they are one of the easiest ways for attackers to infiltrate your software supply chain. In the next chapter, we'll move up the stack to the application layer, where developers often unknowingly import vulnerabilities and malware through open-source package managers like `pip`, `npm`, and `apt`.

Don't Do This	Do This Instead
Pull `ubuntu:latest` from Docker Hub	Pull `ubuntu:22.04` from a trusted, verified publisher
Skip image scanning	Scan every image before use
Trust "official-looking" names	Verify publisher identity and tags
Allow developers to pull directly into prod	Route all image ingestion through controlled, secure pipelines

05 The Hidden Danger of Public Packages

Even the most secure base image can be undermined by what you install on top. In nearly every containerized application, there's a second and often more insidious layer of risk: the application dependencies. Developers frequently rely on public package managers like `pip`, `npm`, `apt` , and `gem` to add functionality to their software. These tools are powerful, but they also introduce massive attack surfaces.

This chapter explores how **public package ecosystems become a vector for container compromise,** how attackers exploit them, and what you can do to detect and defend against these often invisible threats.

The Open-Source Supply Chain Dilemma

Modern software is assembled, not written from scratch. Most applications today are built from roughly 10% original code and 90% third-party libraries and dependencies. These dependencies are pulled from public repositories, including:

- **PyPI** for Python (`pip install`)
- **npm** for JavaScript (`npm install`)
- **Maven Central** for Java
- **RubyGems**, **NuGet**, and others

While these ecosystems provide unprecedented flexibility and innovation, they also operate on a dangerous premise: Anyone can publish. Everyone implicitly trusts.

How Attackers Target the Application Layer

Open-source registries are low-barrier targets for threat actors. Here are some of the most common attack vectors:

Typosquatting

Similar to the typosquatting for container images mentioned earlier, attackers create packages with names nearly identical to popular ones:

- `reqeusts` instead of `requests`
- `lodas` instead of `lodash`

If a developer fat-fingers the name or mindlessly copies and pastes it, it installs malware.

Dependency Confusion

By uploading a malicious version of an internal package to a public registry (e.g., `company-utils`), an attacker tricks build systems that don't prioritize internal sources properly.

Malicious Maintainers

Sometimes, a legitimate open-source project gets abandoned and taken over by bad actors, sold or transferred to a malicious maintainer, or even hijacked through credential theft. Once in control, the attacker publishes a new version with embedded malware, often hiding it in minified code or dependency chains. Many teams use floating version specifiers (`>=`, `^`, `~`), which allow packages to auto-update. If a new version introduces malicious code or vulnerabilities, it's pulled in without warning.

Why Containers Amplify This Risk

When dependencies are added during image builds via:

```
RUN pip install -r requirements.txt
RUN npm install
```

You're freezing vulnerable or malicious code *into* the container itself. That code persists and runs every time the container runs, across all environments (dev, staging, prod). Containers often don't get rebuilt frequently. That means:

- Known vulnerabilities remain unpatched
- Stale packages continue to run with escalated privileges
- Developers and security teams lose visibility into what's running

The False Confidence of Package Managers

Tools like `pip`, `npm`, and apt are designed for convenience, not security. Most software repositories don't verify publisher identity, cryptographically sign packages (or enforce it),warn about known vulnerable versions, or provide audit trails. Unless you're explicitly scanning and locking dependencies, you're blind to what's being installed.

Best Practices for Securing the Application Layer

Securing the app layer starts with breaking the habit of blind installs. To break this habit, here are a few best practices that can be implemented:

Pin Exact Versions

Avoid floating versions. Use explicit, locked versions (e.g., `==2.0.1`).

Use Dependency Lockfiles

Check in and review:

- `requirements.txt` (python)
- `package-lock.json` or `yarn.lock` (npm)
- `Pipfile.lock, Gemfile.lock`, etc.

They act as a snapshot of what's installed and should be reviewed in code audits.

Scan Dependencies Regularly

Use SCA (Software Composition Analysis) tools. Automate scans in CI/CD pipelines and trigger builds to fail if critical vulnerabilities are found.

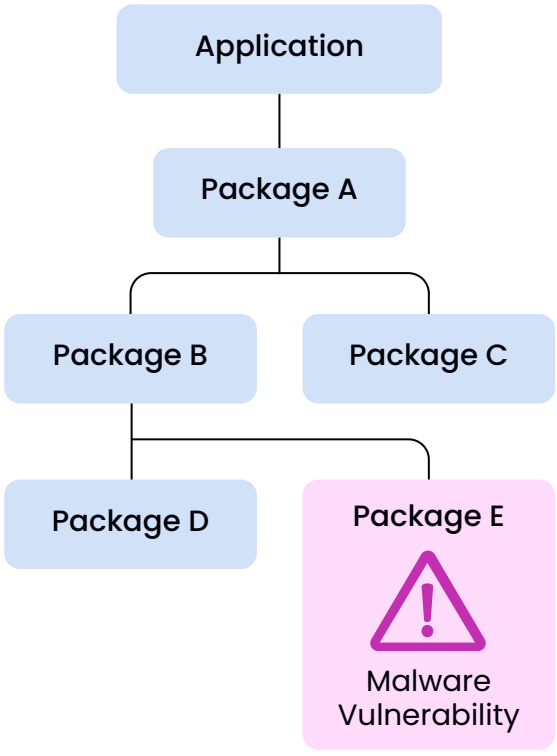
Audit Transitive Dependencies

Many vulnerabilities hide in indirect dependencies (packages pulled in automatically by your direct dependencies).

For example, let's say you're looking to pull in a specific Python package (Package A) for your application. You go to PyPI and pip install the required package, targeting only that specific Python library.

For that library to function, Package A requires Packages B, C, and D, which are then pulled into your build automatically.

Malicious actors frequently target these transitive dependency trees, creating tainted versions of Package A that will unknowingly bring in malicious Package E without any additional interaction from you as a developer.



This threat highlights the need for clear visibility and transparency into your direct and transitive dependency trees. Use tree visualizers (`pipdeptree`, `npm ls`) to understand the full graph.

Freeze and Mirror Trusted Packages

Mirror vetted packages to a private registry (e.g., Artifactory, Nexus) and install from there. This cuts off the public registry as a live dependency and gives you control over what's pulled into your builds.

Limit Network Access at Build Time

Prevent build scripts from calling out to the internet. If something in your dependency chain requires network access just to build, that's a red flag.

Secure by Design: Building with Integrity

Treat your containers as immutable artifacts. Every build should be verifiable, reproducible, and signed. Software Bill of Materials (SBOMs) should be generated and stored. Your build process must enforce rules that include: no new dependencies without review, no public packages from unverified sources, and all packages must pass security scans before use. Ensure that your application stack is as secure as your infrastructure stack.

Takeaway: The Top of the Stack Is Often the Weakest

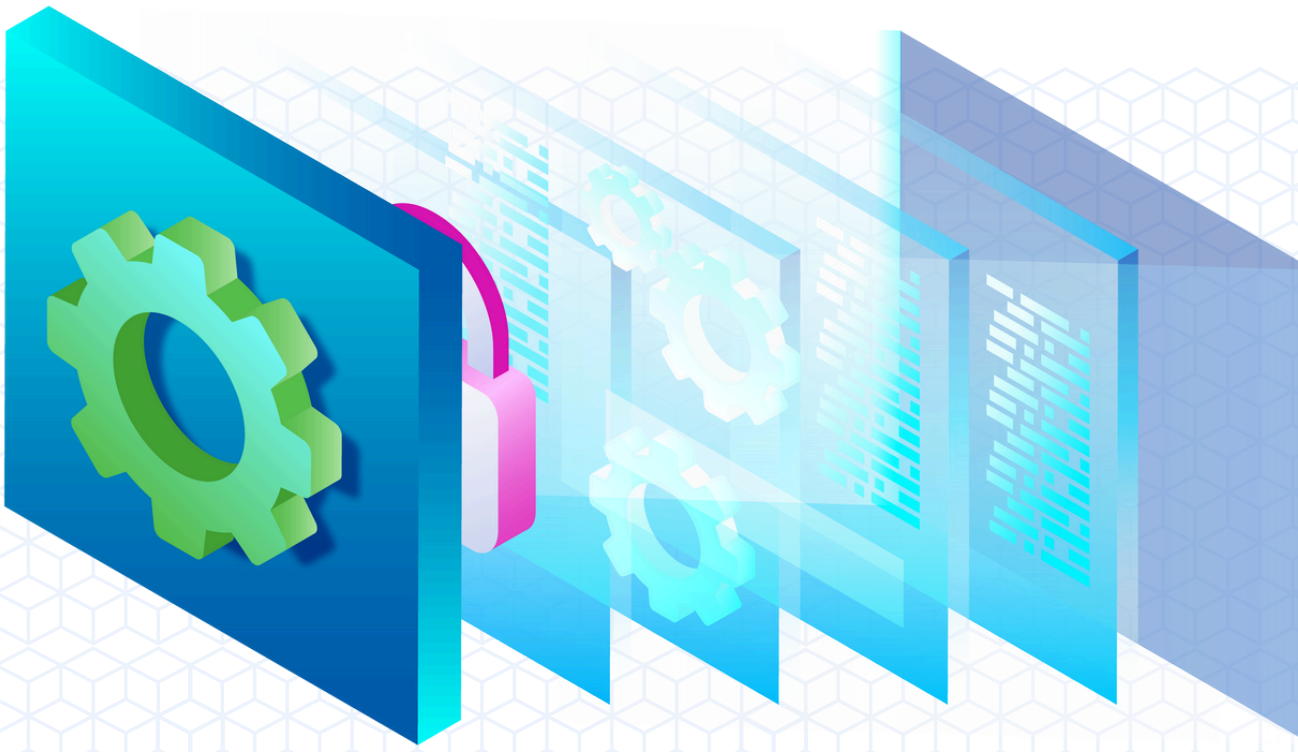
Weakness	Risk
Typosquatted packages	Silent malware installation
Unpinned versions	Pulling vulnerable or malicious updates
Blind use of public registries	Exposure to abandoned, hijacked, or unverified code
Lack of visibility into transitive deps	Hidden risks, difficult to track and audit
Infrequent image rebuilds	Vulnerabilities persist for months or years

Up Next:
A Step-by-Step Approach to Container Security

Now that we've explored the risks of public containers and packages, it's time to build something better. In Part 2, we'll walk through real, practical strategies for creating secure container images from the ground up. You'll learn how to harden your base images, lock your dependencies, and lay the groundwork for a truly trustworthy container.

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 2 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!