DEVGUARD
BY L3MONTREE

# Bit-for-Bit

## How we built a Sovereign, Reproducible Container Supply Chain for DevGuard


Tim Bastin
L3montree GmbH
tim.bastin@l3montree.com

Jacek Galowicz
Applicative Systems GmbH
jacek@applicative.systems


L3montree Cybersecurity

APPLICATIVE

# Bit-for-Bit: How We Built a Sovereign, Reproducible Container Supply Chain for DevGuard

Tim Bastin
L3montree GmbH
tim.bastin@l3montree.com

Jacek Galowicz
Applicative Systems GmbH
jacek@applicative.systems

**Abstract.** Modern software supply chains rely on implicitly trusted build systems, package registries, CI/CD platforms, and container registries. Recent incidents such as the Trivy and Axios compromises demonstrate how attackers can exploit these trust gaps without modifying source code itself.

This paper presents how the DevGuard project rebuilt its OCI container pipeline around reproducible Nix builds and independent dual-platform digest verification. DevGuard images are built hermetically from pinned source revisions, signed with Sigstore/Cosign, and verified through digest comparison across GitHub Actions and sovereign GitLab infrastructure hosted on container.gov.de.

We describe the practical integration of reproducible OCI image builds into existing CI/CD workflows and argue that independently reproducible container digests provide a stronger integrity guarantee against build tampering than provenance alone. The paper further discusses remaining trust assumptions and the relevance of sovereign build infrastructure for government and regulated environments.

## 1 A Wake-Up Call From the Ecosystem

Early 2026 brought two supply chain compromises that hit close to home for anyone running security tooling in production.

The Trivy vulnerability scanner was compromised after an attacker gained direct write access to Aqua Security's repository and pushed a malicious release, tagging 76 of 77 versions of `trivy-action` and all tags in `setup-trivy` with credential-stealing malware, and distributing backdoored Docker images via Docker Hub. The german Federal Office for Information Security (BSI) issued advisory BSI 2026-237970-1032 and recommended that any organisation that ran affected versions treat all CI/CD secrets as compromised. Separately, the JavaScript HTTP library Axios was targeted after an attacker obtained the npm credentials of the primary maintainer and published two poisoned versions that silently injected a fake dependency as a postinstall hook, deploying a cross-platform remote access trojan to every machine that installed those versions (The Hacker News, March 2026).

The structural pattern is the same in both cases: somewhere between source and deployed artifact, an attacker found an unverified step. Build and distribution pipelines implicitly trusted intermediate systems without cryptographic proof of what passed through them.

## 2 The Problem: Trust Gaps in the Software Supply Chain

Every piece of software has a supply chain: the full chain of people, tools, systems, and dependencies involved in getting source code into the hands of a user. Think of it like a physical supply chain for a manufactured product, except instead of raw materials and factories, you have source code repositories, build servers, package registries, dependency trees, and container registries. At each stage, something is transformed and handed off to the next stage.

In a typical modern project that chain might look like this: a developer pushes code to a Git repository, a CI system clones that repository and runs a build, the build fetches dozens or hundreds of dependencies from external package registries, produces a binary or container image, pushes it to a registry, and from there it is pulled and deployed. Each of those steps involves trusting a system, a service, or a third party. Even without attacks, many build systems don't guarantee the same build result if the same process is repeated just a few days later.

The insidious part is that an attacker does not need to touch your source code at all [7, 3]. They can compromise a build runner and inject malicious code during compilation. They can publish a subtly modified version of a popular dependency to a package registry. They can gain write access to a container registry and replace a legitimate image with a backdoored one. They can tamper with a CI/CD action that thousands of projects use. In each case, the source code looks clean, the developer did nothing wrong, and the attack is invisible unless you have a way to verify the integrity of every unverified link and every unverified step between source and deployed artifact, as both incidents above demonstrate.

As the maintainers of the OWASP Incubation project DevGuard (an open source software supply chain security platform that helps development teams manage vulnerabilities, track dependencies, and enforce security policies), we are no strangers to this problem. Like any modern soft-

ware project, DevGuard has its own supply chain, and our primary distribution mechanism is OCI container images. That is where the risk concentrates.

A container image is not a single artifact. It is a layered composition: a base OS image typically pulled from Docker Hub or a third-party registry, language runtimes and dependency trees, application binaries produced by a CI pipeline, and metadata or signatures applied after the fact. By the time an image lands in your Kubernetes cluster or an air-gapped government network, it has passed through many hands.

Each layer is a trust assumption. Even if your own code is completely clean, a compromised upstream base image, a mutable `:latest` tag, or a build runner with registry write access can silently corrupt the final artifact. The consumer has no reliable way to verify that what they pulled is what the author intended to ship.

This is exactly the problem SLSA (Supply-chain Levels for Software Artifacts) [9] was designed to address. SLSA is a well-structured, practical framework for incrementally improving supply chain security. It defines three build levels of increasing assurance: at Build L1 you establish provenance showing how an artifact was built; at Build L2 that provenance is signed by a hosted platform; at Build L3 the build platform itself is hardened against tampering during the build. Critically, SLSA does not prescribe a specific build tool. It defines what properties your build system must have and what evidence it must produce. Complying with SLSA is achievable with standard OCI tooling, and the choice of how to implement those properties is left to the team.

# 3 Enter Nix: Reproducibility as a First Principle

SLSA compliance was a baseline, not a destination. DevGuard is a security and compliance platform: we help other teams reason about the integrity of their software supply chains. We wanted a guarantee that was mechanically enforceable and independently checkable — one that holds even when consumers do not actively verify anything, because in practice, most do not. That said, verifying build provenance is still independently valuable and we strongly encourage it: it provides an additional layer of assurance, particularly for the class of threats that reproducible builds alone cannot address, such as establishing which source revision an artifact was built from and potentially building from an attacker modified source itself.

Standard OCI tooling does not provide that. Images built with Docker, Podman, or Kaniko are inherently non-deterministic. A `RUN apt-get update && apt-get install` pulls whatever versions the package mirror serves that day. A `FROM ubuntu:latest` resolves to a different digest every few weeks. Without explicit version pinning at every layer, two builds from the same `Dockerfile` on different days, or on different machines on the same day, will routinely produce different image digests. You can sign such an image and attach provenance to it, and a diligent consumer can technically verify that metadata — but they still cannot independently reproduce the image from source to confirm it matches. For a tool that organisations deploy into sensitive environments and trust to audit their dependencies, that was not good enough.

Nix [**dolstra2004nix**] is a package manager and build system built around a single radical idea: a build should be a pure function. Given the same inputs, it must always produce the same output, on any machine, at any point in time, with no exceptions.

To understand why that is unusual, consider how a typical build works. You run `apt-get install`, and the package manager fetches whatever version is currently available on the mirror, and different distributions (e.g. Debian vs. Ubuntu) even provide different versions with different patches. You run `pip install requests`, and you get the latest release unless you pinned it. Your build script reads the current timestamp, or queries the network, or depends on a tool that happens to be installed on the build machine. All of these are implicit inputs: things that affect the output of the build but are not recorded anywhere. Two engineers on different laptops, or the same CI runner on two different days, will produce different binaries from the same source code. Most of the time nobody notices, because the binaries are functionally equivalent even if they are not bit-identical.

But "functionally equivalent" is not the same as "verifiably identical." If you cannot reproduce a binary independently, you cannot tell whether the artifact you downloaded was actually produced from the source you trust (and we are sure you audit all sources you use beforehand) — or whether something was injected somewhere along the build and distribution pipeline. Even worse, if you want to rebuild the same version but with a new CVE patch, you often can't.

Nix solves this by creating an empty offline system sandbox for every package build that only provides access to explicitly mentioned dependencies. Every dependency, down to the C compiler, the standard library, and the shell used to run build scripts, must be mentioned in the package recipe and is not identified by a package name symbol but by the full cryptographic hash of its inputs, recursively. This comprises a full graph of all dependencies (down to the compiler of the compiler) and is called *the dependency graph*. The Nix store folder, where all build outputs are stored and act as inputs to other outputs, is append-only: an output is stored at a path derived from the hash of everything that went into building it. Nothing is ever overwritten or mutated. There are no ambient environment variables leaking into the build, no network access, no host tools implicitly available, no home folders or other stateful folders—and any external source that is fetched online must be defined as a special kind of recipe with a predefined output checksum.

The consequence is that a Nix build is hermetic and reproducible by construction. If you give two people the same Nix expression and they build it independently, they get the same output. Not just the same behavior, but the same bytes. You can verify this yourself without trusting anyone

(Strictly speaking, this is not a formal guarantee, but in practice it holds for deterministic toolchains, which include most mature ecosystems such as the Go compiler. Some toolchains still fall short, for example, Turbopack in the JavaScript ecosystem).

For container images specifically, Nix provides `dockerTools.buildLayeredImage`, which produces an OCI-compliant image entirely within the Nix build system. These images are distro-less by default and contain only the exact runtime dependencies of the packed binaries. Every layer digest is deterministic. Rebuilding the same expression on a different machine or six months later produces the same SHA256 for the final image. This is the core promise of the Reproducible Builds initiative [4]: an independently-verifiable path from source to binary.

Until recently, DevGuard published OCI container images built using Kaniko in GitHub Actions and pushed to a registry. We did already attach build provenance to our images — and that is a genuinely good practice that we encourage every team to adopt. But provenance alone only gets you so far: it attests to *how* an artifact was built, but it does not let you verify *what* the artifact actually is. A signed provenance record says *this image was built from commit X by platform Y* — it does not let you confirm that the bytes you pulled match what an honest build from that commit would produce. This guarantee is not better than saying, *We did our best to provide a clean build of the image, but you will never be able to check if our setup was clean because future builds might be slightly different anyway.*

Reproducible builds change this relationship fundamentally. If the build is deterministic, a second independent build from the same source must produce the same digest. Divergence can be reduced to such an extent that it may be considered evidence of manipulation. This means digest comparison between two independent builds is a stronger integrity guarantee than provenance alone — and it is one that holds structurally, without requiring the consumer to verify anything. Switching to Nix was the step that made this possible — but Nix alone is not sufficient. A reproducible single-pipeline build still requires trusting that one pipeline. The structural guarantee comes from combining Nix reproducibility with two independent build environments: if both produce the same digest, neither can have been tampered with unilaterally.

---

# 4 How We Did It: A Practical Walkthrough

Here is exactly how the pipeline is structured, so you can replicate it for your own project. All sources are public. The central architectural feature is two fully independent build pipelines — one on GitHub Actions, one on openCode (GitLab CI on German sovereign infrastructure) — that each build the same Nix expressions and whose output digests are compared after every release. The approach covers three images: the main DevGuard API server, the DevGuard scanner, and a PostgreSQL image. Each follows the same pattern.

## 4.1 Structuring the Nix Flake

The entry point is `flake.nix`. A Nix flake is a self-contained, hermetically sealed description of a project's build: all inputs are pinned via a cryptographic `flake.lock`, and outputs are declared explicitly. DevGuard's flake pins `nixpkgs` to a specific `nixos-25.11` channel commit and declares separate OCI image targets for `x86_64-linux` and `aarch64-linux`. This can be seen in listing 1

**Listing 1:** Nix flake.nix structure

```
targetPkgsAmd64 = nixpkgs.legacyPackages.
    x86_64-linux;
targetPkgsArm64 = nixpkgs.legacyPackages.
    aarch64-linux;

ociImagesAmd64 = import ./nix/oci.nix {
    pkgs = targetPkgsAmd64; ... };
ociImagesArm64 = import ./nix/oci.nix {
    pkgs = targetPkgsArm64; ... };
```

Building multi-architecture images is straightforward in Nix: you simply evaluate the same expression against a different `pkgs` set. There is no `docker buildx`, no QEMU emulation, no cross-compilation configuration to manage manually. The `aarch64-linux` build runs natively on an ARM runner and the `x86_64-linux` build on an AMD64 runner. Both produce bit-identical digests to any other machine of the same architecture running the same expression. The result is that all DevGuard OCI images are now published for both ARM64 and AMD64, with reproducibility guaranteed on each.

## 4.2 What Goes Into Each Image

The scanner image is the most complex, and it illustrates the level of control Nix gives you. Its contents are declared explicitly in `oci.nix` (listing 2)

**Listing 2:** Building an OCI-Image using Nix

```
devguardScannerOCI = pkgs.dockerTools.
    buildLayeredImage {
  contents = [
    pkgs.cacert
    devguardBinaries.devguardScanner
    trivyFromSource
    pythonTools.venv
    craneFromSource
    gitleaksFromSource
    pkgs.jq
    pkgs.gettext
    pkgs.busybox
  ];
};
```

There is no base coreutils layer, no package manager, no shell unless explicitly included. Every single component is declared. If it is not listed here, it is not in the image. The (partly shared) implicit dependencies (e.g. openssl, libc, etc.) are automatically calculated from the dependency graph behind each package entry in this list.

Generally, Nix package definitions describe each package built from source. Rather than pulling pre-built binaries from GitHub releases (which is what most Dockerfiles do), each tool has its Nix derivation with a pinned source revision and a verified hash (Nix does provide a caching mechanism that provides prebuilt packages. This implies some level of trust and can be completely disabled anytime). Trivy, for example, is on date of writing pinned to `v0.69.3` with a specific `sha256` for both the source and its Go module vendor tree. The expression can be found in 3

**Listing 3:** Nix Expression to fetch trivy from source

```
src = fetchFromGitHub {
  owner = "aquasecurity";
  repo  = "trivy";
  rev   = "v${version}";
  hash  = "sha256-lzFcLyrORA+1
    LxS4nzJVvilg29GTNiGRmnjJ47ev/yU=";
};

vendorHash = "sha256-lzFcLyrORA+1L..."; #
    truncated
```

If either the source or any of its dependencies changes by even a single byte, the hash check fails and the build breaks. This is a direct application of the threat model SLSA describes: by compiling from source ourselves, we eliminate the class of attacks that target the gap between a project's source code and its published artifacts. The backdoored Trivy release from earlier in this post would never have reached us at all — we never download a Trivy, Crane, or Gitleaks release artifact. We compile all three directly from pinned source revisions, skipping the entire build and distribution segment of each tool's own supply chain. We are not just verifying the supply chain — we are making it shorter.

**PostgreSQL comes directly from nixpkgs.** We use `pkgs.postgresql_16`, which means we inherit the version, patches, and security fixes maintained by the nixpkgs community. The version is determined entirely by our pinned `nixpkgs` commit in `flake.lock`.

**Semgrep and Checkov are managed via uv2nix.** Using Nix, these can't be managed by `pip` as the Nix sandbox does not provide other tools random access to the internet. Established Python tooling like `uv` already provides dependency constraint-solving facilities that result in exact lock files, which can be picked up by additional Nix tooling like `uv2nix` to automatically compute and download the complete dependency tree for the required Python virtual environment [8]. All version pins are captured in that lockfile, including explicit constraints that fix known CVEs in transitive dependencies.

**Listing 4:** pyproject.toml which allows overriding dependencies

```
[tool.uv]
constraint-dependencies = [
  "urllib3>=2.6.3",
  "python-multipart>=0.0.22",
  "starlette>=0.49.1",
  "wheel>=0.46.2",
  "mcp>=1.23.0",
]
```

If a CVE is discovered in a transitive dependency of Checkov or Semgrep, we update the constraint in `pyproject.toml`, regenerate `uv.lock`, and the fix is applied across the entire environment. No additional overhead introduced by Nix, no hoping that upstream has patched their dependency tree.

We are in control of exactly what versions end up in the image, but control comes with responsibility. When a CVE surfaces in a transitive dependency that upstream has not yet patched, we can apply the fix ourselves and keep shipping. That is genuinely powerful. It is also a burden: we are now maintaining patches against codebases we do not own, and we cannot always be certain a backported fix behaves correctly in the context of the version we are running.

Nix makes applying patches straightforward. When a CVE fix exists as a patch but has not yet been merged upstream or picked up by nixpkgs, you can apply it directly in the derivation using the `patches` attribute in any package recipe (see 5 for an example).

**Listing 5:** Using a custom patch in Nix

```
stdenv.mkDerivation {
  pname   = "some-tool";
  version = "1.2.3";

  src = fetchFromGitHub { ... };

  patches = [
    # Backported CVE fix -- applied
    directly rather than waiting for
    upstream.
    ./CVE-2025-12345.patch
  ];
}
```

The build will fail if any patch does not apply cleanly, which gives you an immediate signal if an upstream change has made your patch obsolete. This is how nixpkgs itself handles backported security fixes — and how we apply them to our own derivations when a CVE surfaces in a pinned dependency before upstream ships a patched release.

## 4.3 The Builder Image

Both CI environments pull a shared Nix builder image maintained by Applicative Systems and hosted on container.gov.de: `oci-community/images/applicative-systems/nix`. This image comes with Nix pre-installed and configured. Using a shared, pinned builder image is important: it means both environments start from an identical foundation, which is a precondition for the final image digests being meaningfully comparable.

The builder image is hosted on German sovereign infrastructure (publicly operated, jurisdiction-specific infrastructure independent of commercial cloud providers) and continuously checked against known vulnerabilities.

This is about as close as we can get to the classic "trusting the compiler" problem described by Ken Thompson in his 1984 Turing Award lecture Reflections on Trusting Trust [10] — at some point you have to trust a foundation, and we have chosen to make that foundation as transparent, auditable, and sovereign as possible.

We are currently in the process of providing a sovereign Kaniko fork on container.gov.de as well, so teams building regular OCI images can start from a similarly trusted foundation — even if, honestly, nobody can claim to have audited every line of source code that goes into it.

## 4.4 The GitHub Actions Pipeline

The GitHub workflow delegates to a reusable workflow from `devguard-action`, called once per image. An example is presented in listing **??**

**Listing 6:** Example github action pipeline

```
  uses: l3montree-dev/devguard-action/.
    github/workflows/full-nix.yml@nix
  with:
    nix-target-amd64:      devguard-scanner
    -amd64
    nix-target-arm64:      devguard-scanner
    -arm64
    nix-version:           '2.34.4'
    image-name:            ghcr.io/${{
    github.repository }}/scanner
    asset-name:            l3montree-
    cybersecurity/projects/devguard/assets/
    devguard
    api-url:               https://api.main
    .devguard.org
    nix-cache-substituter: https://nix.
    garage.l3montree.cloud
    nix-cache-public-key:  nix.garage.
    l3montree.cloud:...
```

The `full-nix.yml` reusable workflow builds both architectures, pushes the multi-arch manifest, and triggers DevGuard to generate and attest build provenance.

## 4.5 The GitLab CI Pipeline

The GitLab pipeline mirrors the same structure using a reusable template. Both the GitHub Actions workflows and the GitLab CI templates are open source and can be found in the devguard-action and devguard-ci-components repositories. Listing 7 shows the concrete YAML-Block needed.

**Listing 7:** GitLab CI-Template für das DevGuard-OCI-Image

```
include:
  - remote: https://gitlab.com/l3montree/
    devguard/-/raw/nix/templates/build-nix-
    multiarch.yml
    inputs:
      image_suffix:     "scanner"
      nix_target_amd64: "devguard-scanner-
    amd64"
      nix_target_arm64: "devguard-scanner-
    arm64"
      arm64_runner_tag: "arm"
```

## 4.6 The Nix Binary Cache

Both pipelines use a shared Nix binary cache at `nix.garage.l3montree.cloud` to avoid rebuilding unchanged derivations, which keeps CI fast without sacrificing reproducibility. The cache stores outputs keyed by their derivation hash, so a cache hit is itself a proof of identity.

Cache poisoning is a real attack vector: if an attacker can push a malicious derivation output into the cache under a

legitimate hash, every subsequent build pulling from that cache is compromised. To mitigate this, all cache entries are signed, and the only signing key currently lives on a personal MacBook — meaning nothing can be written to the cache without physical access to that machine. We are considering moving the key to a YubiKey to further harden this, but even in its current form the trust boundary is narrow and explicit: the cache is a read-through optimisation, not an implicit trust anchor.

Even without our custom cache for our own package definitions, most upstream packages from nixpkgs are cached on cache.nixos.org. Applicative Systems already built completely air-gapped Nix build pipelines for organisations with use cases where trusting the upstream caches was deemed unacceptable [1].

## 4.7 Signing and Attestations

Every published image is signed via Sigstore/Cosign [5]. You can verify any image yourself with the commands presented in listing 8

**Listing 8:** Verifying a devguard image using cosign

```
cosign verify \
  --insecure-ignore-tlog=true \
  --key https://raw.githubusercontent.com/
    l3montree-dev/devguard/main/cosign.pub
    \
  ghcr.io/l3montree-dev/devguard:main-amd64
```

The `--insecure-ignore-tlog` flag tells Cosign to skip the Rekor transparency log lookup and verify the signature directly against the provided public key instead. We sign with a long-lived key rather than Sigstore's keyless ephemeral certificate flow, which means there is no Rekor entry to look up. The flag name sounds alarming but the verification itself is not weakened: you are checking the signature against a specific, publicly auditable key hosted in our repository.

A signature alone only proves that we published the image. Each image therefore also carries SLSA build provenance in the in-toto predicate format [**torresarias2019intoto**] documenting what was built, from which source revision, and under which build environment; a VEX attestation stating which known vulnerabilities apply or have been mitigated; and a CycloneDX SBOM [6] enumerating every component in the image — all attached directly to the OCI artifact and retrievable by anyone who pulls it, without consulting an external service. Because the provenance includes Nix derivation inputs, a consumer can not only check that we signed something, but independently confirm what went into building it.

## 4.8 Dual-Build Digest Verification

After both pipelines complete their builds independently, a verification step compares the resulting image digests. They must match byte for byte for the release to proceed. You can watch this comparison in real time at devguard.org/reproducible-builds.

This check is only meaningful because the builds are reproducible. With a non-deterministic pipeline, two independent builds would always produce different digests, making comparison useless. Nix is what makes the comparison a genuine integrity check.

Before mapping individual threats, a structural observation is worth making explicit: for the majority of build integrity threats, reproducible builds and dual-build digest verification provide a stronger guarantee than SLSA build provenance alone. Provenance attests to *how* an artifact was produced; digest verification demonstrates that the artifact *is* what an independent, honest build from the same source produces. An adversary who tampers with a build artifact must produce an output whose digest matches the other platform's honest build — something that is only possible with a genuinely unmodified build. Provenance therefore becomes primarily relevant for the one threat this structural check cannot address: establishing *which* source was used.

Mapping this to the SLSA framework: **Build L1** is met by the signed in-toto provenance attached to every image. **Build L2** is met by both build environments being hosted platforms that generate provenance independently. **Build L3** is met by the Nix sandbox, which provides a hardened, hermetic build environment with no network access during the build and minimal, near-zero host state leaks (Nix uses Linux namespaces, which, e.g., can't hide the host architecture or kernel interface). The dual-build digest comparison goes beyond what SLSA mandates at any level — it is our own architectural choice, and it is what makes the reproducibility property externally verifiable rather than a claim consumers must take on trust.

The following maps each SLSA v1.2 build threat to the specific properties of the pipeline.

**Threat (D) — External build parameters.**

- *Unofficial build steps / unofficial parameters* — **Structurally eliminated.** The entire build is declaratively described in the flake, every input is pinned in `flake.lock`, and the Nix sandbox enforces hermeticity by construction.
- *Build from unofficial fork or branch* — **Residual gap; detectable via provenance.** An adversary with write and trigger access to both pipelines could build from a tampered branch; both environments would produce matching digests, so the digest comparison alone cannot detect this. The provenance attestation records the exact source revision, making such a release anomalous in both the in-toto record and the repository history.

**Threat (E) — Build process.**

- *Forge values of the provenance (Build L2+)* — **Structurally mitigated.** Forging signed provenance re-

quires compromising the control plane of GitHub Actions or OpenCode. Beyond that, the dual-build comparison requires the forged provenance to reference a digest matching the honest build on the other platform — only achievable with a genuinely unmodified build, defeating the purpose of the forgery.

- *Compromise other build (Build L3)* — **Structurally mitigated.** Each Nix build executes in an ephemeral, namespace-isolated environment. More importantly, releasing a tampered artifact requires simultaneously compromising both GitHub Actions and OpenCode and causing both to produce identical tampered outputs — a substantially higher bar than a single-pipeline compromise.
- *Steal cryptographic secrets (Build L3)* — **Partially mitigated; acknowledged gap.** The Cosign signing key is a CI secret. The Nix cache signing key is better isolated and never touches CI. Exfiltrating the Cosign key is nevertheless insufficient to release a tampered artifact: the adversary must also produce a matching digest on the second platform, making the stolen key a necessary but not sufficient condition. We are working on a trusted builder architecture where DevGuard signs attestations using a key never exposed to build workers; details at docs.devguard.org/explanations/attestations-provenance/slsa-level-3.
- *Poison the build cache (Build L3)* — **Structurally mitigated.** The Nix binary cache is keyed by the full transitive input closure. A substitution attack requires invalidating the derivation hash. Write access requires a signing key that never touches any build worker — an explicit trust assumption we are working to harden with a hardware security module.
- *Compromise build platform admin* — **Partially mitigated.** We rely on each platform's own controls. The dual-build check compensates partially: a platform-level compromise altering outputs on one side produces a digest mismatch against the other, preventing release.

**Threat (F) — Artifact publication.**

- *Upload without provenance / tamper with artifact after CI/CD (Build L1)* — **Structurally mitigated; additionally monitored.** Every published image carries Cosign-signed provenance with the artifact digest as its subject. Separately, we operate DevGuard deployments with active alerting on signature and provenance check failures, enabling an operational response independent of any consumer action.
- *Tamper with provenance (Build L2)* — **Structurally mitigated.** The Cosign signing key is a CI secret, so key isolation is not a strong guarantee. However, retroactive modification of signed provenance invalidates the signature, and any forged attestation must reference a digest matching the honest build on the other platform.
- *Build with untrusted CI/CD* — **Structurally mitigated.** Our threat model does not assume either CI platform is unconditionally trustworthy. The assurance rests on the assumption that GitHub Actions and OpenCode will not be simultaneously compromised — converting a single trust assumption into a compound one.

**Threat (G) — Distribution channel — Structurally mitigated.** Any post-signing modification in transit or on the registry produces a digest mismatch against both the provenance subject and the comparison baseline at devguard.org/reproducible-builds.

Crucially, none of the structural mitigations above require the consumer to verify anything. The dual-build digest comparison runs unconditionally on every release. Consumer-side signature and provenance verification adds a complementary layer and is strongly encouraged, but is not a precondition for the guarantees described here.

Returning to the incidents that opened this post: the injected `plain-crypto-js` dependency in the Axios attack would have produced a derivation hash mismatch, failing the Nix build before any payload could execute. The backdoored Trivy release would never have reached us at all — we compile Trivy, Crane, and Gitleaks from pinned source revisions and never consume their distribution artifacts, skipping the entire build and distribution segment of each tool's own supply chain. The attack surface is not eliminated, but it is dramatically narrowed, and what remains is visible and auditable.

# 5 What This Means for Government and Regulated Environments

When a federal authority deploys a container, "where did this image come from, and can I trust it?" is not an abstract question. It has legal, operational, and security dimensions [2].

Building DevGuard on container.gov.de with Nix-based reproducible builds makes the full build chain independently auditable: source expressions are public, both build environments are hosted on governed infrastructure, and the real-time digest comparison at devguard.org/reproducible-builds provides a mechanically verifiable integrity guarantee, not just a policy assertion.

Container.gov.de is Germany's publicly operated container registry and open source platform, developed under the Secure Government Container Initiative. Images hosted there are governed by public institutions rather than private cloud providers, independent of Docker Hub, GitHub Container Registry, or any commercial third party. For federal agencies and regulated industries, this matters both legally and operationally. Using it as one of our two independent build and distribution nodes means the entire supply chain operates without a single proprietary chokepoint.

We hope this serves as a concrete example of what a sovereign, reproducible container supply chain can look like in practice, and that it is useful to the container.gov.de community as a reference for other projects looking to build on the same foundation.

*DevGuard is open source and developed by L3montree, a software engineering company focused on security and digital sovereignty. The Nix build expressions, pipeline configurations, and digest verification tooling are all publicly available. Contributions and independent reproductions are welcome.*

*Need help building reproducible, sovereign NixOS infrastructure? Applicative Systems offers consulting and engineering support for teams working in high-assurance and regulated environments.*

# References


[1] Jacek Galowicz. *Demonstrably Secure Software Supply Chains with Nix*. May 2025. URL: https://nixcademy.com/posts/secure-supply-chain-with-nix/.

[2] White House. *Executive Order 14028, Improving the Nation's Cybersecurity*. 2021.

[3] Piergiorgio Ladisa et al. "SoK: Taxonomy of Attacks on Open-Source Software Supply Chains". In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023, pp. 1509–1526. DOI: 10.1109/SP46215.2023.10179304.

[4] Chris Lamb and Stefano Zacchiroli. "Reproducible Builds: Increasing the Integrity of Software Supply Chains". In: *IEEE Software* 39.2 (2022), pp. 62–70. DOI: 10.1109/MS.2021.3073045.

[5] Zachary Newman, John Speed Meyers, and Santiago Torres-Arias. "Sigstore: Software Signing for Everybody". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, 2353–2367. ISBN: 9781450394505. DOI: 10.1145/3548606.3560596. URL: https://doi.org/10.1145/3548606.3560596.

[6] US NTIA. "The Minimum Elements for a Software Bill of Materials (SBOM)". In: *https://www. ntia. doc. gov/files/ntia/publications/sbom_minimum_elements_report. pdf* (2021).

[7] Marc Ohm et al. "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Clémentine Maurice et al. Cham: Springer International Publishing, 2020, pp. 23–43. ISBN: 978-3-030-52683-2.

[8] *pyproject-nix/uv2nix: Uv2nix - Ingest uv workspaces using Nix [maintainer=@adisbladis]*. URL: https://github.com/pyproject-nix/uv2nix.

[9] *SLSA • SLSA specification*. URL: https://slsa.dev/spec/v1.2/.

[10] Ken Thompson. "Reflections on trusting trust". In: *Communications of the ACM* 27.8 (1984), pp. 761–763.