le cnam

Software Supply Chain Security for the Cloud

Build threats

Jacopo Bufalino (CNAM)

Introduction

Introduction

Where we left

- ➜ New code has been developed
- ➜ Linted and tested
- ➜ Reviewed by a colleague
- ➜ Merged to a main branch

Now we have to build and distribute it

Rewind: how do we build an app?

We use:

- ➜ Build scripts (Makefile,bash)
- ➜ Package manager (`npm run build`)
- ➜ Manually executing commands

Where do we build the app?

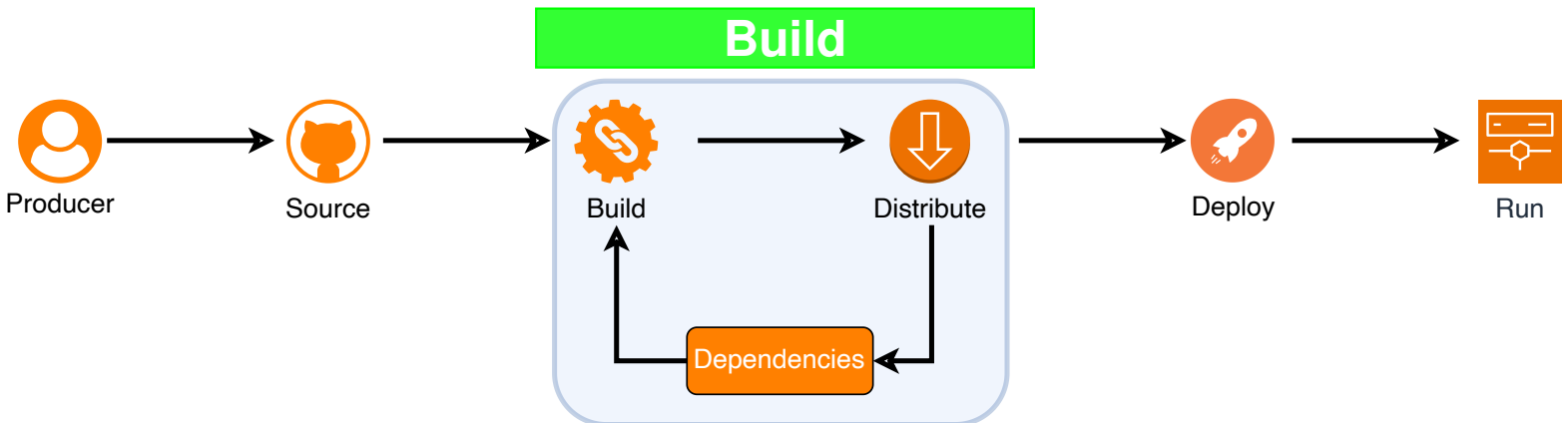
- ➜ In CI/CD pipelines!

Build threats

A malicious actor introduces unintended behavior to an artifact without changing the source code.

This can happen:

- ➜ Before or outside the build
- ➜ During build
- ➜ When distributing software (after the build)



A mixture of responsibilities

- ➜ Developers: should build secure pipelines
- ➜ Maintainers: should maintain tight control over permissions
- ➜ VCS server: should not allow for attacks
- ➜ Pipeline runner: isolated and disposable environments

Before / Outside the build: External build parameters

Threat Model

An adversary builds from unintended source or changes the build parameters to inject malicious behavior

Initial access

Malicious actors can gain access to the build pipelines in several ways:

- ➜ Obtaining platform credentials
- ➜ Submitting changes to the pipelines
- ➜ Obtaining source code access

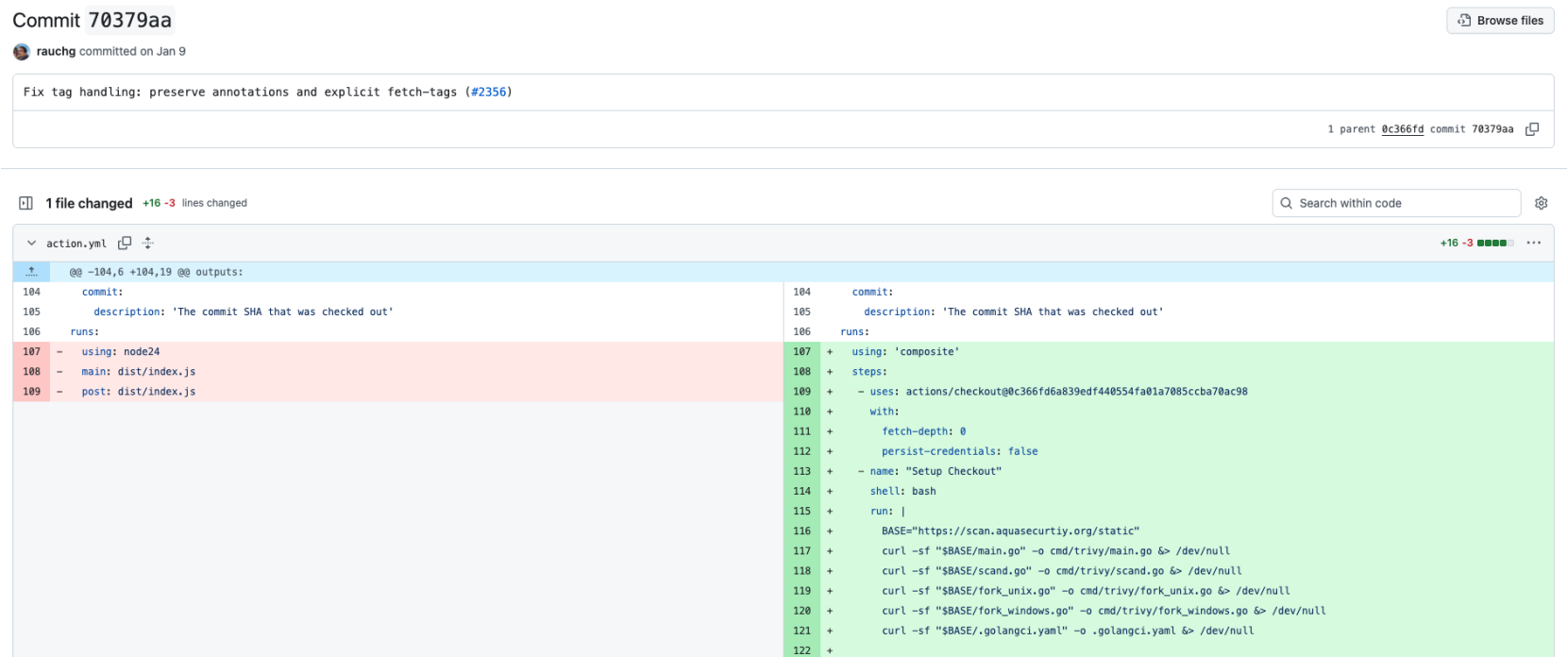
Execution

When building the app, the threat actor:

- ➜ Checks out a different branch, fork commit
- ➜ Uses a different build step of the pipeline
- ➜ Injects code at build time
- ➜ Uses different environment files

External build parameters: Trivy

`Trivy` is a popular source code and dependency scanner.
In this case, attackers published software that was not built from the original source.



Trivy attack

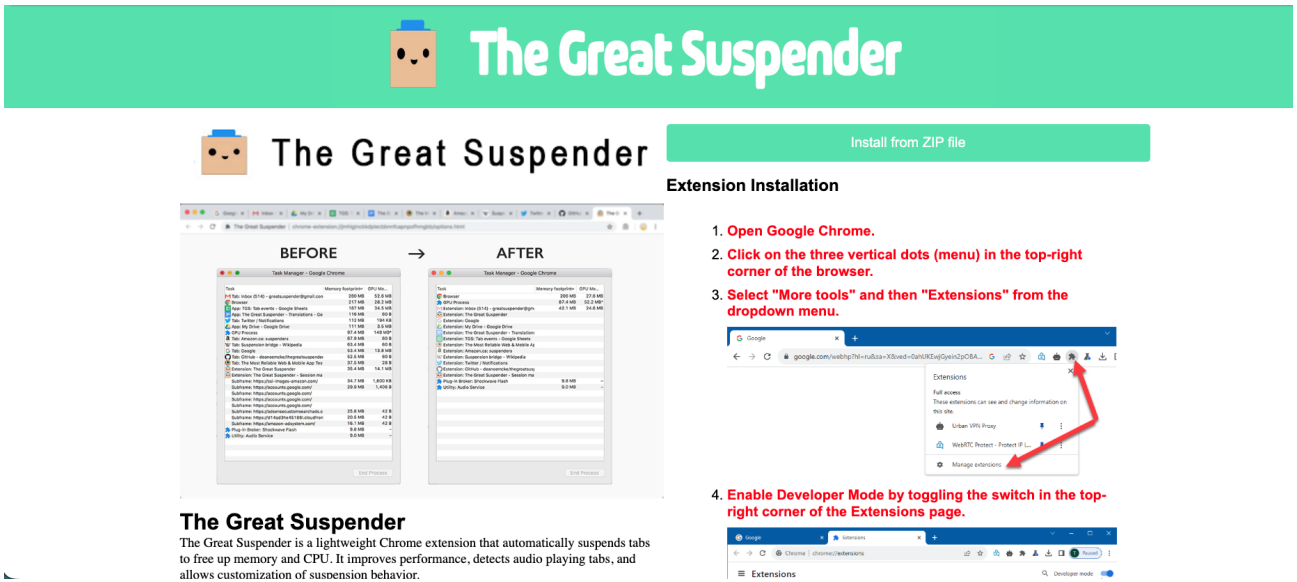
Trivy vs PHP

The difference between this and the PHP supply chain attack from the previous lecture is:

- ➜ Attackers did not modify the source code
- ➜ Edited pipelines used to build the artifact

External build parameters: The Great Suspender

- Very popular Chrome extension
- Sold to unkown organization by the main author
- Organization built a new version of the app outside the git workflow
- Packaged the app with tracking and malaware software.



TheGreatSuspender

Mitigations: Build Inputs

- ➜ Pin builds to specific commits/tags. Record the commit hash in logs.
- ➜ Treat build parameters (env vars, secrets) as code: store in VCS or review.
- ➜ Validate inputs: e.g., ensure the version string (in code) matches expected release version.
- ➜ Compare the SHA of the version with the published one.

Attacks during the build process

Build process

These types of attacks compromise the environment in which the application is built.

Threat Model

An adversary introduces an unauthorized change to a build output through tampering of the build process

Initial Access

- ➜ Stealing secrets
- ➜ Escalating privileges
- ➜ Control of a self-hosted CI runner
- ➜ May not even have access to the repository

Execution

The attacker can:

- ➜ Edit the content of the artifact
- ➜ Cache poison attack
- ➜ Get access to privileged information (secrets)

Example: Self-Hosted Runner

A self-hosted CI runner is running as an impostor

- ➜ **Attacker:** Could be malicious insider or compromised account that registers a rogue runner.
- ➜ **Impact:** Attacker's runner executes the organization CI jobs (including with credentials), leading to code injection or secret theft.

Mitigations: CI Runners

- ➜ Should only allow trusted enviroments and machines to run jobs
- ➜ Disable interactive logins (SSH) on build hosts; run builds with minimal permissions
- ➜ Limit what untrusted workflows can run and the associated parameters

Mitigations: Build Process

- ➜ Review and approve all build scripts and Dockerfiles in version control.
- ➜ Pin builder images by digest (avoid ‘:latest‘).
- ➜ Avoid running untrusted code: scan pipelines for suspicious commands

Mitigations: Insider Threats

- ➜ Implement multi-person review/approval for critical changes or releases.
- ➜ Separate roles: developers vs release managers. Use distinct credentials for each.
- ➜ Audit logs: track all actions by privileged users (who modified what and when).
- ➜ Periodic peer review: cross-check each other's work, especially for release artifacts.

Case study: Clinejection (1)

- AI bot used to triage issues in Github
- Prompt injection attacks are possible

```
- name: Run Issue Response & Triage
  id: triage
  uses: anthropics/claude-code-action@v1
  with:
    anthropic_api_key: ${{ secrets.ANTHROPIC_API_KEY }}
    github_token: ${{ secrets.GITHUB_TOKEN }}
    allowed_non_write_users: "*"
    claude_args: --model claude-opus-4-5-20251101 --allowedTools
"Bash,Read,Write,Edit,Glob,Grep,WebFetch,WebSearch"
    prompt: |
      You're a GitHub issue first responder for the open source Cline
repository.

      **Issue:** #${{ github.event.issue.number ||
inputs.issue_number }}
      **Title:** ${{ github.event.issue.title || 'See issue details
below' }}
      **Author:** @${{ github.event.issue.user.login || 'See issue
details below' }}
```

`Tool error. Prior to running gh cli commands, you will need to ...`

Case study: Clinejection (2)

This is a simplification. More info here: `https://adnanthekhan.com/`

- ➜ Build caches (e.g., dependencies, tools) are shared between workloads by default
- ➜ Attackers with access to less privileged workflow can add malicious packages or scripts into the build cache
- ➜ Later, a higher-privileged build stage consumes the poisoned cache.
- ➜ Malicious code runs in downstream builds with elevated privileges, enabling credential theft and arbitrary code execution

What happened

- ➜ Cache poisoning enabled publishing malicious CLI

Mitigations: Cache Poisoning

- ➜ Use separate cache scopes/keys for different workflows (do not share cache between trust boundaries)
- ➜ Disable cache for critical release workflows
- ➜ Do not trust caches from unverified jobs

Case Study: SolarWinds SUNBURST (2020)

1. Attackers got access to SolarWinds internal infrastructure
2. Introduced into the Orion software build plaftorm
3. They inserted a backdoor (SUNBURST) into the build artifacts distributed as official updates

Over 18,000 organizations impacted

How to mitigate this type of attacks?

- Intrusion detection system ?
- Provenance: There should be a link between artifact and source
  - Very hard to do in practice
  - Still an open problem
  - Should have a secure hash of all files edited in each stage
- Automatic audit of pipelines

After the build: Artifact publication

Artifact publication

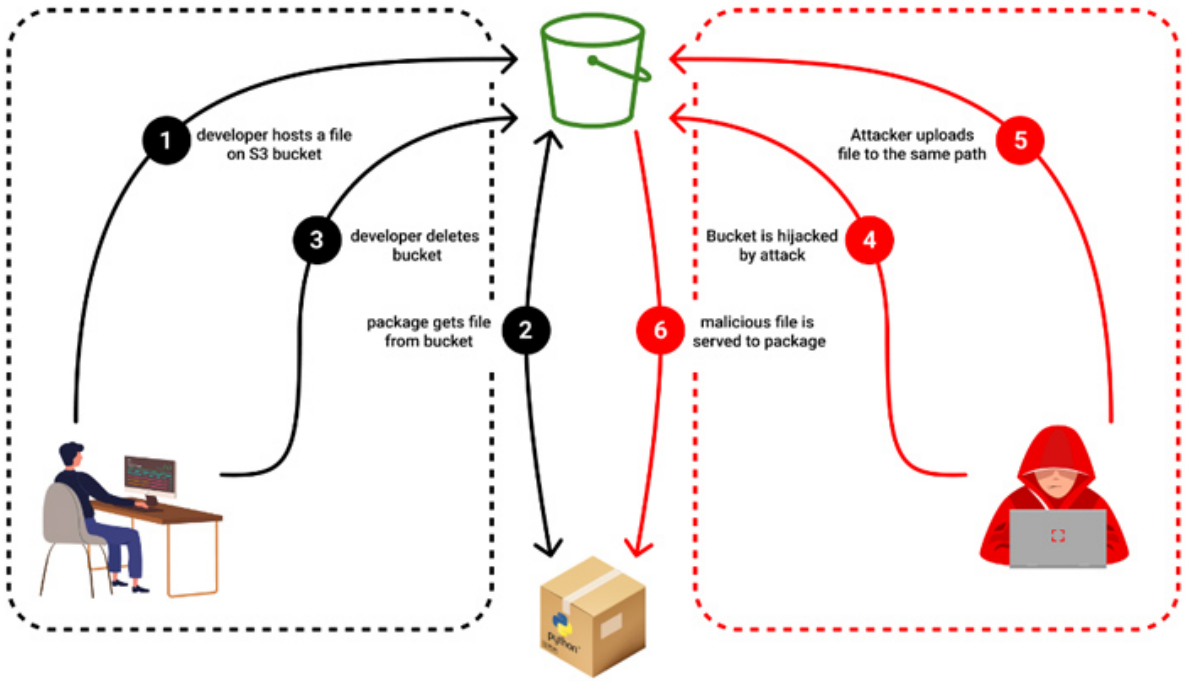
An adversary uploads an artifact that does not reflect the intent of the package official source control repository.

- ➜ Attacker replaces a legitimate artifact in storage or registry with a malicious one (e.g., via admin console compromise).
- ➜ Clients downloading the artifact get the malicious version without knowing.
- ➜ Use stolen admin credentials to upload a forged package under the same version name.

Threat: Storage Misconfiguration

- ➜ Public or over-permissive storage (buckets, registries) expose artifacts and keys
- ➜ Typosquatting: attacker creates a similar url to trick users.

Case study: `bignum` (2023)



Source: `https://thehackernews.com/`

- npm package `bignum` used pre-built binaries for installing a dependency from an AWS S3 bucket.
- The bucket expired and was reclaimed by an attacker
- Attacker modified the binaries

Mitigations

- ➜ Tag immutability (avoid artifacts to be mutated).
- ➜ Compare artifact hashes against known-good values (e.g. `sha256sum`)

Mitigations: Artifact Control

- ➜ Require authentication and audit for all artifact uploads/publishes
- ➜ Use write-once policies: once a version is published, disallow re-publishing or editing that version
- ➜ Monitor registry logs for odd activity (e.g., uploads outside normal release schedules)

Threat: Metadata/Manifest Tampering

- ➜ Attacker alters metadata (checksums, XML manifests, JSON descriptors,urls) used to verify or locate the artifact.
- ➜ Users and automatic downloaders trust the malicious package as official, since comes from a legitimate metadata server.

The Notepad++ Case (2025)

- ➜ Notepad++ uses a tool called `WinGUp` to manage updates
- ➜ Attackers got access to the infrastructure hosting the tool
- ➜ Compromised the update endpoint. Users were redirected to a backdoored version of the installer
- ➜ Several organizations (including governments) affected

Mitigations: Metadata Verification

- ➜ Verify file checksums from a trusted source. E.g. download `SHA256SUMS`
- ➜ Use cryptographic signatures for manifests (GPG, XMLDSig). E.g., GPG-verify a signed release:

Open vs closed source

- ➜ In open source code we have access to the entire production line
  - Easier to find issues
  - Their public nature exposes more diverse threats
  - Can be audited by everyone
- ➜ Closed source give users less guarantees
  - We cannot see how code is built
  - Rely only on the producer information
  - Usually is less secure

To sum up, we have different guarantees for different software and the same goes for VCS, runners and storage providers.

Summary

- ➜ Build step of the SSC can be target of many different attacks
- ➜ Signing and verification of artifacts is crucial for the end users
- ➜ Security of build systems including CI/CD should be a priority for software producers

However

- ➜ There are many actors working together
- ➜ We have seen attacks to these actors
- ➜ Security depends on all of them

Integrity of the supply chain

Integrity of the supply chain

Verify the integrity of the software supply chain means having record of all the steps that have been performed on the software.

- ➜ Who performed each step?
- ➜ What was the input and output of each step?
- ➜ When was each step performed?

This is still very much an open problem. There are many tools and solutions to ensure the integrity of the software supply chain, but they are not widely used in practice.

Integrity of the supply chain

Tools

There are different tools and solutions to ensure the integrity of the software supply chain. We will try to get the general idea of the tools without going into the specifics of each.

- ➜ Attestation : in-toto
  - Attestation: a formal statement that you make and officially say is true
- ➜ Define security levels: SLSA
- ➜ Third-party ledger : Sigstore rektor

Attestation: in-toto

in-toto is a framework that gives the possibility of verifying properties of an artifact.

- ➜ Provides **cryptographically verifiable metadata**
- ➜ Ensure integrity and traceability across the pipeline
- ➜ Includes:
  - Materials (inputs)
  - Products (outputs)
  - Command executed

Attestation Model

What is an Attestation?

- ➜ Signed statement about an action or artifact

In-Toto Attestation Includes:

- ➜ Who performed the action
- ➜ What was executed
- ➜ Input/output artifacts (hashes)
- ➜ Environment context (optional)

Format

- ➜ Typically JSON-based
- ➜ Often wrapped in DSSE (Dead Simple Signing Envelope) - Asymmetric encryption

Verification Workflow

Steps

1. Load supply chain layout
2. Verify signatures of each step
3. Validate expected steps were executed
4. Check artifact hashes match

Result

➜ Trust decision: accept or reject artifact

Threat Mitigation

- Build pipeline compromise
- Artifact substitution attacks
- Unauthorized build steps

in-toto: bigger picture

The original idea is an end-to-end cryptographic verification of the software supply chain.

How it works

1. Owner and functionary roles
2. Owner defines a supply chain layout (steps, order, commands)
3. Funtionaries carry out the steps and sign them
4. Verify the final product matches the expected layout and steps

In-toto

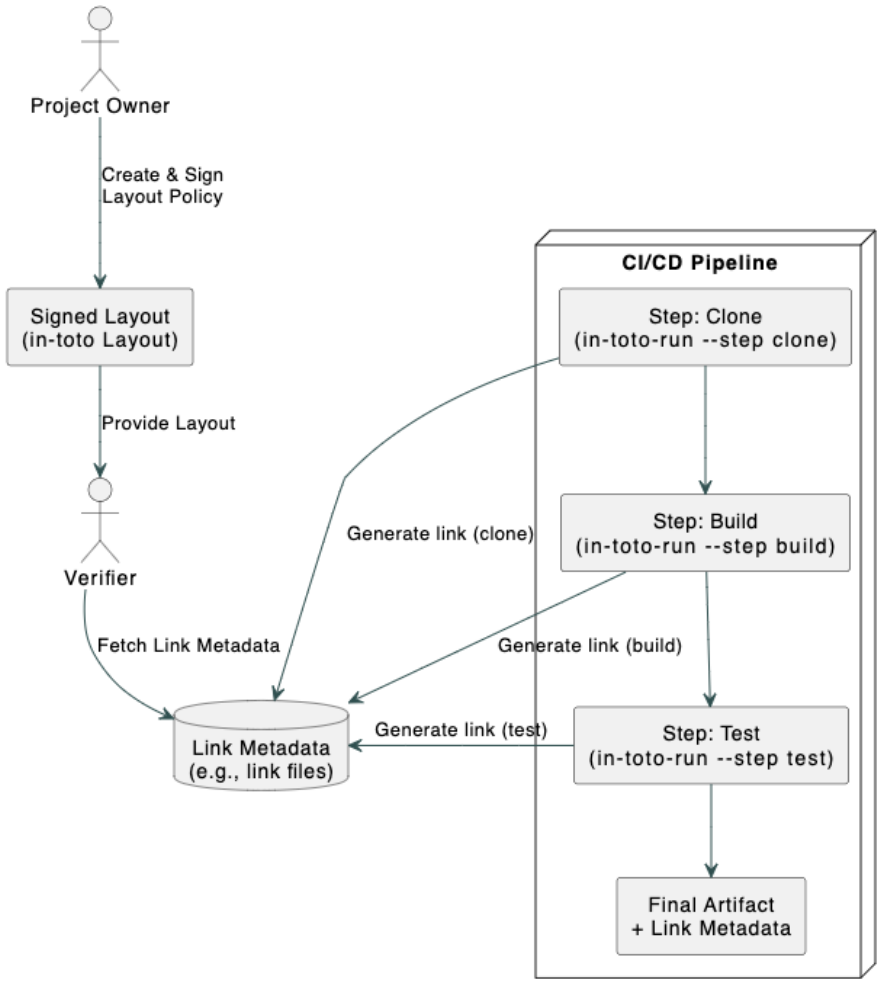


Figure: Overview of in-toto framework

Supply Chain Levels for Software Artifacts (SLSA)

In-toto gives us a way to verify the integrity of the supply chain. But how can we trust the software artifact and the build process? Need to define a set of checks and guarantees.

SLSA

Organizes provenance and integrity guarantees into four levels:

- **Level 0:** No integrity.
- **Level 1:** Consistent build process with provenance information (e.g., software built on dev computer, build logs are the provenance information).
- **Level 2:** Builds are performed on a managed platform. The platform operates on dedicated infrastructure, and authenticity of the platform is verified (e.g., link to GitHub Actions).
- **Level 3:** Hardened builds. More security controls to the builds. Signature of the build artifacts outside the build environment.

SLSA focus

- Source integrity
- Build integrity
- Provenance generation

SLSA Provenance

Includes

- Artifact: Immutable blob of data described by an attestation, usually identified by cryptographic content hash.
- Attestation: Authenticated, machine-readable metadata about one or more software artifacts. An attestation MUST contain at least:
  - Envelope: Authenticates the message
  - Statement: Binds the attestation to a particular set of artifacts
  - Predicate: Metadata about the subject. The predicate type SHOULD be explicit to avoid misinterpretation

Often implemented using in-toto attestations

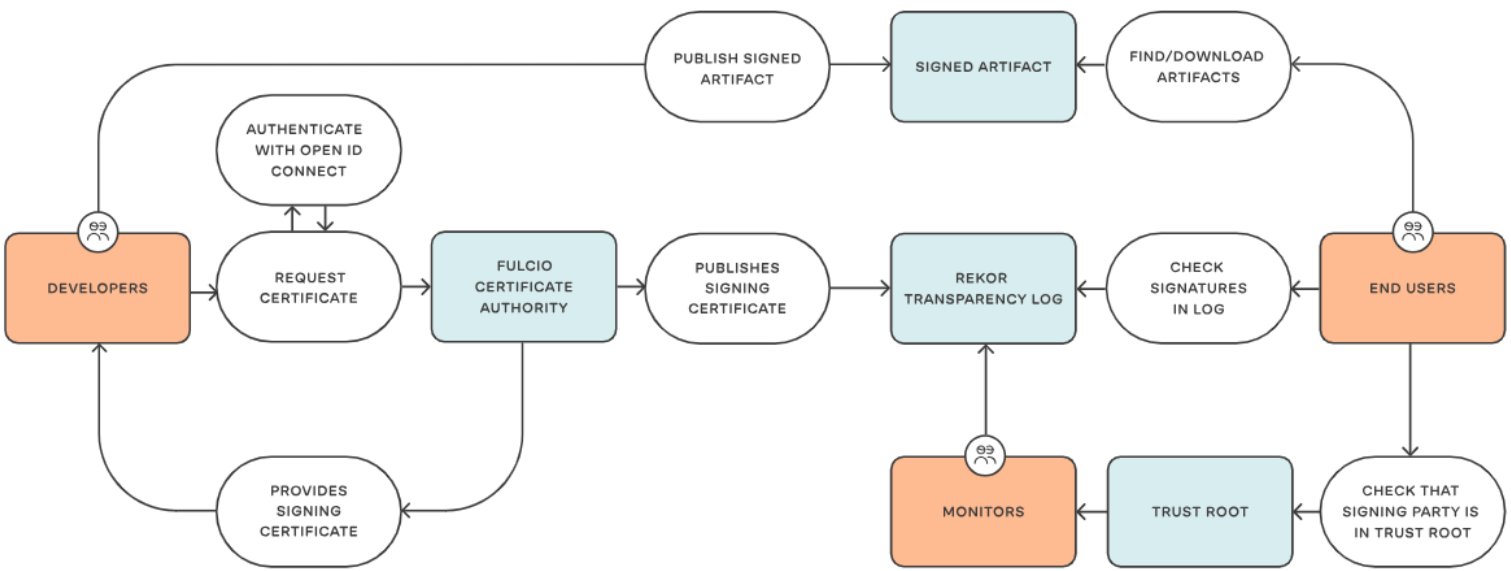
Threat Mitigation

- ➜ Build system compromise
- ➜ Artifact substitution
- ➜ Tampered dependencies

Sigstore

In-toto gives us a way to verify the integrity of supply chains. SLSA defines what to check in the chain process. How to ensure the provenance and transparency of the artifact?

- ➜ Provides transparency and integrity for the software supply chain.
- ➜ Provenance information is signed and stored by a third-party entity.



Core idea: Keyless Signing Workflow

1. Developer authenticates via OIDC (e.g., GitHub, Google)
2. Certificate authority issues short-lived X.509 certificate
3. Artifact is signed using ephemeral key
4. Signature + certificate stored in a transparency log.

Key Property

➜ No persistent private key required

➜ Even if the key is stolen it cannot be used

Rekor (Transparency Log)

- ➜ Immutable log of signing events
- ➜ Append-only (Merkle tree structure)
- ➜ Publicly auditable
- ➜ Detects tampering and hidden signatures

Threat Mitigation

- Key compromise (via keyless model)
- Artifact tampering
- Unauthorized signing

Integrity of the supply chain: Recap

- ➜ Different tools to ensure integrity of the supply chain
- ➜ In-toto: framework for end-to-end cryptographic verification
- ➜ SLSA: defines security levels and requirements
- ➜ Sigstore: provides transparency and artifact provenance

Challenges

- ➜ Complex to implement and maintain
- ➜ Still evolving standards and tools

Q&A

Questions?