Session 12

Software Supply Chain Security

Software Security and Privacy

Computer Science and Engineering Department

January 14, 2026

The Modern Software Ecosystem

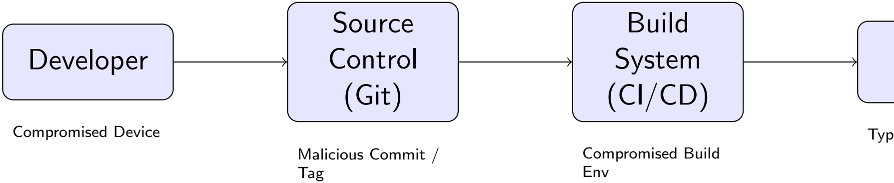
- Modern software is rarely written from scratch.
- It is **assembled** from open source libraries, frameworks, and containers.
- **Statistic:** 70-90% of modern application code consists of open source components.
- **The Challenge:** You inherit the security posture of your dependencies (and their dependencies).
- "It's turtles all the way down."

What is the Software Supply Chain?

- Anything that goes into your software (code, binaries, libraries).
- Who wrote it?
- When was it contributed?
- How was it reviewed?
- How was it built?
- How is it delivered?

Supply Chain Security ensures the integrity and provenance of all these artifacts throughout the lifecycle.

Visualizing the Supply Chain



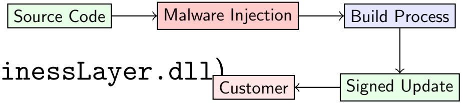
Every link in this chain is a potential attack vector.

Attack Taxonomy

1. **Upstream Attacks:** Malicious code injected into open source dependencies.
   - *Typosquatting:* `request` vs `requests`.
   - *Dependency Confusion:* Internal vs Public package names.
   - *Maintainer Compromise:* Stolen credentials.
2. **Midstream Attacks:** Compromising the build pipeline.
   - *SolarWinds:* Injecting malware during the build process.
   - *Codecov:* Modifying the uploader script in CI.
3. **Downstream Attacks:** Compromising update mechanisms or signing keys.

Case Study: SolarWinds (Sunburst)

- **Target:** Orion Network Management System.
- **Method:** Attackers compromised the build system.
- **Mechanism:** The build server was patched to include a malicious DLL (`SolarWinds.Orion.Core.BusinessLayer.dll`) into legitimate updates.
- **Impact:** Thousands of organizations, including US gov agencies, installed the signed but backdoored update.



Case Study: Log4Shell (Log4j)

- **Vulnerability:** JNDI Injection in `log4j-core`.
- **Significance:**
  - Ubiquity: Used in millions of Java applications.
  - Deep Dependency: Often included transitively (Dep A $\rightarrow$ Dep B $\rightarrow$ Log4j).
- **Lesson:** You need to know what you are running.
- **Challenge:** How do we find every instance of Log4j deep in our dependency graphs?

Dependency Confusion

- Many companies use internal package registries (e.g., PyPI, npm) mixed with public ones.
- **Attack:** Attacker registers a public package with the *same name* as an internal private package but a *higher version number*.
- **Result:** Package manager (pip, npm) defaults to the higher version from the public repo.
- **Mitigation:** Scoped packages (@myorg/pkg), strict registry configuration.

SLSA (Supply-chain Levels for Software Artifacts)

- Pronounced "salsa".
- A security framework from ensuring artifact integrity.
- **Goal:** Prevent tampering, improve integrity, and secure packages.

SLSA Levels

Level 1: Build process is scripted and version controlled. Provenance exists.

Level 2: Build runs on a dedicated build service. Provenance is authenticated.

Level 3: Build platform is hardened. Provenance is non-falsifiable.

NIST SSDF (Secure Software Development Framework)

- **SP 800-218**.
- Set of fundamental, sound, and secure software development practices.
- Four Groups:
  1. **Prepare the Organization (PO):** People, processes, tech.
  2. **Protect the Software (PS):** Tamper protection.
  3. **Produce Well-Secured Software (PW):** Minimal vulnerabilities.
  4. **Respond to Vulnerabilities (RV):** Remediation.
- Often a requirement for US Federal Government software vendors (EO 14028).

What is an SBOM?

- A nested inventory (a list of ingredients) that makes up software components.
- Contains:
  - Library Names
  - Versions
  - License Information
  - Checksums / Hashes
  - Dependencies of Dependencies
- **Analogy:** Nutrition label on food packaging.

SBOM Formats

SPDX (Software Package Data Exchange)

- ISO/IEC 5962:2021 standard.
- Heavy focus on license compliance initially, now security too.
- Linux Foundation.

CycloneDX

- OWASP flagship project.
- Designed specifically for security contexts / application security.
- Lightweight, typically JSON/XML.

Tool: Syft (Generation)

- CLI tool and library for generating SBOMs from container images and filesystems.
- Developed by Anchore.

Example Usage

```
# Generate SBOM for a docker image
$ syft packages docker:alpine:latest -o cyclonedx-json > sbom.js

# Scan a local directory
$ syft packages dir:. -o spdx
```

- Can detect OS packages (APK, DEB, RPM) and Language packages (gems, pip, npm, jars).

Tool: Grype (Vulnerability Scanning)

- A vulnerability scanner for container images and filesystems.
- Works best when paired with Syft (scan the SBOM, not just the image).

Example Usage

```
# Scan an SBOM generated by Syft
$ grype sbom:sbom.json

# Scan an image directly
$ grype docker:nginx:latest
```

- Outputs CVEs, severity, and fix versions.

Tool: Trivy (Comprehensive Scanner)

- An all-in-one security scanner (Filesystem, Git, Container, K8s).
- Very popular in CI/CD pipelines due to ease of use.

Example Usage

```
# Scan a container image
$ trivy image python:3.4-alpine

# Scan a filesystem for vulnerabilities & misconfigs
$ trivy fs --scanners vuln,misconfig .

# Scan a git repository
$ trivy repo https://github.com/knqyf263/trivy-ci-test
```

Automated Dependency Management (GitHub Dependabot)

- **What is it?** An automated bot that scans your dependency files for outdated or insecure requirements.
- **How it works (Behind the Scenes):**
  1. **Detection:** Parses manifest files (e.g., `package.json`, `go.mod`) and checks against the *GitHub Advisory Database*.
  2. **Resolution:** Determines the "secure" version that is compatible with your version constraints.
  3. **Action:** Creates a new branch, updates the manifest/lock file, and opens a Pull Request (PR).
  4. **CI/CD:** Triggers your CI pipeline to ensure the update doesn't break tests.
- **Impact:** Significantly reduces the "Time to Remediate" for known CVEs.

The Problem of Noise: VEX

- Scanners (like Grype) find *potential* vulnerabilities based on version matching.
- **Reality:** Is the vulnerable function actually called? Is it reachable?
- **VEX (Vulnerability Exploitability eXchange):**
  - A machine-readable statement claiming whether a product is affected by a vulnerability.
  - Statuses: *Not Affected*, *Affected*, *Fixed*, *Under Investigation*.
- Allows vendors to suppress false positives in scanners.

Example: CycloneDX SBOM (JSON)

sbom.json (Snippet)


```json
{
  "bomFormat": "CycloneDX",
  "specVersion": "1.4",
  "components": [
    {
      "type": "library",
      "name": "requests",
      "version": "2.25.1",
      "purl": "pkg:pypi/requests@2.25.1",
      "licenses": [ { "license": { "id": "Apache-2.0" } } ]
    }
  ]
}
```

Example: VEX Statement

vex.json (Snippet)


```json
{
  "statements": [
    {
      "vulnerability": "CVE-2021-44228",
      "status": "not_affected",
      "justification": "code_not_reachable",
      "impact": "Log4j is used only for testing, not in prod.",
      "products": [ "pkg:docker/myapp@v1.0.0" ]
    }
  ]
}
```

The Signing Problem

- Historically, signing software (PGP) is hard.
- Key management is painful (rotation, storage, revocation).
- Developers lose keys or commit them to git.
- **Result:** Nobody signs artifacts, or nobody verifies signatures.

Sigstore & Cosign

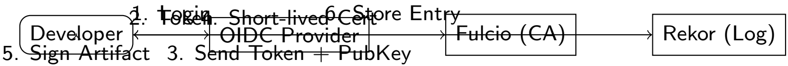
- **Sigstore:** A project to make signing easy and transparent.
- **Cosign:** CLI tool to sign containers and blobs.

Keyless Signing (The Magic)

Instead of managing long-lived keys:

1. Authenticate via OIDC (Google, GitHub, Microsoft).
2. Generate ephemeral keys.
3. Sign artifact with ephemeral key.
4. Record the signature and OIDC identity in a public **Transparency Log (Rekor)**.
5. Discard the key.

Keyless Signing Flow



Verification checks the transparency log to prove that the cert was valid at the time of signing.

Using Cosign

Signing a Container

```
$ cosign sign --key cosign.key user/demo
# Or Keyless (opens browser for OIDC)
$ cosign sign user/demo
```

Verifying a Container

```
$ cosign verify --key cosign.pub user/demo
# Or Keyless
$ cosign verify \
    --certificate-identity=alice@example.com \
    --certificate-oidc-issuer=https://accounts.google.com \
    user/demo
```

Reproducible Builds

- **Definition:** Given the same source code, build environment, and instructions, any party can recreate bit-for-bit identical copies of all specified artifacts.
- **Why?**
  - Prevents the "compromised build server" attack (SolarWinds).
  - If I build it and you build it, and hashes match, we trust the compiler didn't inject malware.
- **Challenges:** Timestamps, non-deterministic compiler outputs, file ordering.

Hermetic Builds

- Builds that are isolated from the network and the host system.
- **Rule:** All dependencies must be declared explicitly. No fetching from the internet during ‘make‘.
- **Tools:** Bazel, Nix.
- Ensures that the build is predictable and dependencies are pinned/hashed.

Attestations

- An authenticated statement about a software artifact.
- "I built this artifact from this git commit on this runner."
- stored in the container registry alongside the image.
- **in-toto:** A framework to secure the integrity of the software supply chain. Defines the layout of the pipeline and verifies that steps were carried out as intended.

Demo Scenarios

We will explore the following scenarios:

1. **Vulnerability Scanning:**
   - Build a Docker image with known vulnerabilities (old Python).
   - Generate an SBOM using `syft`.
   - Scan the SBOM using `grype` and `trivy`.
2. **Signing & Verification:**
   - Generate a key pair with `cosign`.
   - Sign a local file/image.
   - Verify the signature to ensure integrity.
3. **Supply Chain Attack Simulation:**
   - Simulate a "Typosquatting" attack in Python.
   - Show how easy it is to install the wrong package.

Best Practices for Supply Chain Security

1. **Know your dependencies:** Generate SBOMs regularly.
2. **Scan for vulnerabilities:** Automate tools like Grype or Trivy in CI.
3. **Pin dependencies:** Use lock files (`package-lock.json`, `go.sum`). Avoid generic versions like `latest` or `^1.2.3` in critical infra.
4. **Sign your artifacts:** Use Cosign/Sigstore.
5. **Secure the pipeline:** SLSA Level 2+ (Hosted runners, ephemeral environments).
6. **Monitor for new threats:** VEX and continuous scanning.

Future Trends

- **Mandatory SBOMs:** Government regulations (US EO 14028, EU Cyber Resilience Act).
- **Chainguard / Distroless:** Minimal images with zero known vulnerabilities.
- **Graph-based Analysis:** Understanding "reachability" of vulnerabilities to reduce alert fatigue.
- **Policy as Code:** Preventing unsigned or vulnerable images from running in Kubernetes (Kyverno, OPA Gatekeeper).

Summary

- Supply Chain Security is about trust in the entire lifecycle, not just your code.
- Attacks are shifting from run-time to build-time.
- Tools like Syft, Grype, and Cosign form the modern defense stack.
- Frameworks like SLSA provide the roadmap for maturity.

Keywords

- TODO
- TODO

Resources

- TODO
- TODO