Software Supply Chain Security Course

Flavio Copes

Updated August 3, 2026

Contents

Know what you build with **2**
- Map the dependency graph . . . 2
- Use lockfiles deliberately . . . 3
- Inspect package behavior . . . 4
- Choose fewer, better dependencies . . . 4
- Check your understanding: know what you build with . . . 5

Protect source and CI **6**
- Protect repository access . . . 6
- Require reviewed changes . . . 6
- Keep secrets out of source . . . 7
- Minimize CI authority . . . 8
- Check your understanding: protect source and ci . . . 9

Build verifiable artifacts **9**
- Build from clean inputs . . . 9
- Create an SBOM . . . 10
- Record provenance and signatures . . . 11
- Harden container images . . . 12
- Check your understanding: build verifiable artifacts . . . 13

Handle vulnerabilities **13**
- Scan the complete release . . . 13
- Triage reachability and impact . . . 14
- Patch transitive dependencies . . . 15
- Publish a vulnerability policy . . . 17
- Check your understanding: handle vulnerabilities . . . 18

Release and respond **18**
- Protect publishing authority . . . 18
- Verify before deployment . . . 19
- Respond to a dependency incident . . . 20
- Complete the supply-chain review . . . 21
- Check your understanding: release and respond . . . 22

Secure dependencies, repositories, CI workflows, artifacts, releases, vulnerability handling, publishing authority, and dependency incident response.

What you'll learn: Trace and strengthen the complete path from dependency choice and source review through a verifiable artifact and controlled release.

Know what you build with

Map dependencies, lock resolution, inspect package behavior, and choose trusted code deliberately.

Map the dependency graph

Distinguish direct, transitive, development, build, and runtime dependencies and identify which code reaches production.

Your application includes code you wrote and code selected by your dependencies. Start by seeing the complete graph.

A **direct dependency** is a package you added to `package.json` yourself. A **transitive dependency** is a package that one of your dependencies pulled in. You never chose it, but it runs with the same authority as everything else.

Print the whole tree:

```
npm ls --all
```

On a typical Node.js project this prints hundreds of lines. Ten direct dependencies commonly expand into several hundred transitive ones. Every line is code someone else can change.

When you want to know why a specific package is installed, ask npm to explain it:

```
npm explain qs
# qs@6.11.0
# node_modules/qs
#   qs@"6.11.0" from body-parser@1.20.2
#   node_modules/body-parser
#     body-parser@"1.20.2" from express@4.18.2
```

Read the chain bottom-up: you installed `express`, which chose `body-parser`, which chose `qs`. If `qs` ships a malicious version, it arrived through a decision you never reviewed.

The tree is not the whole graph

Direct dependencies are only the first layer. Build plugins, CI actions in your workflow files, container base images, and binaries downloaded during install can all influence the artifact, and none of them appear in `npm ls`.

Imagine an image upload service that imports one package. That package downloads a native image tool during installation. The downloaded binary never appears in the top-level manifest, but it still runs in production.

Record execution context, not just names

A dependency list without execution context is incomplete. Build tools can read CI secrets, while runtime packages can reach customer data. A test-only package that runs in CI still sees the job's environment.

So for each class of dependency, record two things: the path into the project, and the authority the component receives where it runs. `devDependencies` that never ship can still exfiltrate a CI token. That distinction drives everything else in this course.

Generate the complete dependency tree for a small project and save the output. Pick one transitive

package and show which direct dependency introduced it, where it runs, and whether removing the parent removes it. Then repeat the check for one build plugin or downloaded binary that the package tree does not show.

Use lockfiles deliberately

Commit and review the ecosystem lockfile so local, CI, and production builds resolve the same dependency graph.

A version range describes allowed updates. A lockfile records the exact graph selected for this application.

When `package.json` says `"express": "^4.18.0"`, you are telling npm that any 4.x version at or above 4.18.0 is acceptable. Which one you actually get depends on what the registry offers at install time. The **lockfile** (`package-lock.json` for npm) freezes that answer: exact versions, resolved URLs, and an integrity hash for every package in the tree.

Here is why that matters. A developer adds `mailer` with a version range and reviews version 3.2. CI resolves 3.3 the next morning because the lockfile was not committed. The source did not change, but the code entering the build did. If 3.3 was a compromised release, it shipped without anyone looking at it.

Install deterministically in CI

`npm install` may update the lockfile. `npm ci` never does:

```
npm ci
```

It deletes `node_modules`, installs exactly what the lockfile says, verifies each package against its recorded integrity hash, and fails if `package.json` and `package-lock.json` disagree. That failure is a feature: it means someone changed a range without regenerating the lock.

Add a guard so CI also catches an install step that silently modified the lockfile:

```
git diff --exit-code package-lock.json
```

A non-zero exit means the file changed during the build. Fail the job and investigate.

Review lockfile changes like code

Regenerate the lockfile through the package manager, never by editing `integrity` or `resolved` fields by hand. Review the lockfile diff together with the manifest change that caused it. A one-line dependency bump that rewrites two thousand lockfile lines deserves a question, not an approval reflex.

Know what the lockfile does not cover

A frozen lockfile prevents resolution drift. It does not cover a script that downloads `latest.zip` at install time, and it does not control your container base image. It also does not prove that a locked package is trustworthy — it only guarantees you get the same bytes every time.

Run the frozen install command from a clean checkout and save the resolved dependency tree. Change one manifest range without updating the lockfile and show that the command fails. Also identify one build input, such as a remote binary or container base, that the lockfile does not control.

Inspect package behavior

Review maintainers, release history, source, permissions, install scripts, native binaries, and network behavior before trusting a dependency.

A package name and download count are not a security review. Look at what the package can do during install and runtime.

The npm registry lets any package define **lifecycle scripts** like `preinstall` and `postinstall`. These run arbitrary code on your machine, or in CI, the moment you install. A color-formatting package looks harmless at runtime but defines a `postinstall` script. During CI installation, that script can read environment variables and make network requests with the job token still present.

Inspect a package before adding it:

```
npm pack kleur --dry-run
npm view kleur scripts dist.integrity repository
```

The first command lists exactly which files the published tarball contains, without installing anything. The second shows the declared lifecycle scripts, the integrity hash, and the linked repository. If `scripts` prints nothing, the package runs no code at install time.

Inspect the exact published archive, not just the GitHub page. Repository source and packaged files can differ: the 2018 `event-stream` attack shipped malicious code only in the published tarball. Registry popularity cannot reveal that.

Find install scripts already in your tree

You can query your installed graph for packages with install-time hooks:

```
npm query ":attr(scripts, [postinstall])"
```

This prints every installed package that declares a `postinstall` script. Each one is code that executes with your CI job's authority on every fresh install.

Check the name and the people

Typosquatting is real: in 2017 a package named `crossenv` mimicked the popular `cross-env` and harvested environment variables via its install script. Read the name character by character before you install.

Then check who maintains it. Look at maintainer changes, recent release cadence, unresolved security reports, and whether the repository link actually points to the code you think it does. A tiny convenience package with broad install-time authority may cost more risk than a few local lines.

Inspect the published files, lifecycle scripts, maintainers, repository link, and integrity metadata for one real dependency. Save the command output and name every install-time capability you find. Then test the review method on a package with a lifecycle script or bundled native binary.

Choose fewer, better dependencies

Evaluate necessity, maintenance, ownership, update cost, alternatives, and removal before adding code to the trusted computing base.

A dependency can save months of work. It can also add a permanent update and incident obligation.

Every package you add joins your **trusted computing base**: the set of code that can compromise your product if it misbehaves. It brings a maintainer account, a registry release path, and an update stream. Those stay attached to your product for as long as the package does.

A team considers adding a package to test whether a number is even. The package saves one line today. It is not hypothetical: `is-even` exists on npm, and it depends on `is-odd`, which depends on `is-number`. One trivial check imports three maintainer accounts into your supply chain.

Evaluate before you install

Before adding a package, spend two minutes on evidence:

```
npm view fastify time.modified maintainers
npm view fastify dependencies
```

The first command shows when the package was last published and who controls it. The second shows how many packages it drags in. A package last published four years ago with one maintainer and thirty dependencies is a different proposition than an actively maintained package with two dependencies.

Ask these questions and write down the answers:

- Does it solve a substantial, difficult problem, or duplicate a platform feature?
- Is there a credible maintenance path — releases, issue triage, more than one owner?
- What would replacing it cost in six months?

Prefer mature packages that solve hard problems. Avoid abandoned wrappers around things the language already does. `Object.keys()`, `fetch`, and `structuredClone` replaced whole families of utility packages.

Writing it yourself is not automatically safer

The goal is not zero dependencies. A mature parser or cryptographic library is difficult to replace correctly, and your hand-rolled version will not get security patches from anyone. Compare implementation risk with continuing dependency risk instead of counting packages.

Record why a sensitive dependency was chosen and how it could be replaced. That one paragraph turns a future incident from archaeology into a lookup.

Write a one-page decision for one proposed dependency: add it, implement the feature locally, or defer it. Include the code size, maintenance activity, install behavior, replacement cost, and security impact. Test the decision against an edge case where the local implementation would be difficult to get right.

Check your understanding: know what you build with

Check dependency graphs, lockfiles, package metadata, install scripts, provenance, and dependency choices.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Protect source and CI

Control repository access, reviews, secrets, workflow permissions, and untrusted contributions.

Protect repository access

Require strong authentication, least privilege, attributable accounts, prompt offboarding, and recovery controls for source hosting.

Source hosting controls what becomes trusted code. Protect it as a production system.

Anyone who can push to your repository can change what every build after that moment contains. That includes people, but also machine identities: deploy keys, personal access tokens, and installed GitHub Apps. Each one is a door.

Start with authentication. Require MFA or passkeys for everyone in the organization, avoid shared accounts, and limit how many people are organization owners. An owner can change every other control, so each owner account is a single point of failure.

Inventory who and what has access

You cannot review access you have not listed. With the GitHub CLI:

```
gh api repos/acme/api/collaborators --jq '.[] | "\(.login) \(.permissions.admin)"'
gh api repos/acme/api/keys --jq '.[] | "\(.title) read_only=\(.read_only)"'
```

The first command prints every collaborator and whether they hold admin rights. The second prints deploy keys and whether each can write. A write-capable deploy key with a title nobody recognizes is a finding, not background noise.

Every entry in those lists needs an owner and a reason. If nobody can say why an integration has access, remove it and see what breaks in staging.

Offboard immediately, not eventually

A contractor leaves, but their personal account and old deploy key remain attached to the repository. Months later, either credential can still change release source without using a current team identity. Access removal belongs in the offboarding checklist with the same urgency as revoking a production database password.

Recovery is a privileged path too

Keep emergency recovery controlled and tested. Strong daily authentication helps little if one shared recovery account can bypass it. Store recovery codes for the organization owners somewhere access-controlled and audited, and rehearse the lost-owner scenario before you need it.

Export or capture the current owners, collaborators, applications, deploy keys, and recovery methods for one repository. Assign an owner and reason to every privileged entry, then remove one disposable test identity and prove its access fails. Record how the team would recover if the last owner lost access.

Require reviewed changes

Protect release branches, require focused review and passing checks, and make emergency bypass visible and temporary.

A protected branch turns the expected release path into an enforceable rule. Review is strongest when the change is small enough to understand.

Without protection, "we always review changes" is a habit, and habits fail under deadline pressure. On GitHub, a branch protection rule or ruleset on `main` makes the process mechanical: require pull requests, require current approvals, require status checks to pass, and block force pushes.

Two settings matter more than people expect. **Dismiss stale approvals** means a new push after approval needs a fresh review — otherwise an attacker can get a harmless diff approved and then push the real payload. **Require branches to be up to date** means the checks ran against the code that will actually merge.

You can verify what a branch actually enforces:

```
gh api repos/acme/api/branches/main/protection --jq '{reviews: .required_pull_request_reviews
```

If that returns a 404, the branch is not protected at all, whatever the team believes.

Protect the files that control the pipeline

A pull request changes application code and the workflow that tests it. The altered workflow reports success without running the security check, so a green badge alone does not represent independent review.

Give pipeline-defining files stricter owners with a `CODEOWNERS` file:

```
/.github/workflows/  @acme/release-admins
package.json         @acme/platform
package-lock.json    @acme/platform
```

With code owner review required, changes to workflows and dependency manifests need approval from those specific teams, not just any teammate.

Make bypass visible, not impossible

Emergency bypass can be necessary during an outage. The tradeoff is concentrated authority, so the bypass must be attributable, time-bounded, and reviewed afterward. An admin override that appears in the audit log and triggers a post-incident review is a control. A permanently disabled rule "because deploys were annoying" is a hole.

Create a test branch rule that requires one approval and one real status check. Save evidence that a failing unreviewed change cannot merge. Then exercise the documented bypass in a disposable repository and show where the event is logged and how normal protection is restored.

Keep secrets out of source

Prevent, detect, and respond to committed credentials without treating history rewriting as a substitute for revocation.

Git remembers. Once a secret enters a repository, assume every clone and automation path may have copied it.

Prevention starts with boring hygiene. Keep real values in files Git never sees, and commit an example file with the shape but not the values:

```
# .gitignore
```

```
.env
.env.local
```

```
# .env.example - committed, no real values
DATABASE_URL=
RESEND_API_KEY=
```

Real secrets live in a secret manager or in your platform's environment configuration, injected at runtime. The example file documents what the app needs without exposing anything.

Detect before and after the commit

Add a scanner to catch mistakes before they land. **gitleaks** scans working trees and full history:

```
gitleaks detect --source . --verbose
# Finding: AWS access key
# Secret: AKIA...
# File: config/deploy.js  Commit: 4f2a91c
```

A non-zero exit code means it found something; the output tells you the file, the rule that matched, and the commit. Run it as a pre-commit hook and in CI. On GitHub, also enable secret scanning and push protection so the platform blocks known credential formats at push time.

Detection has a cost: it can flag harmless test strings. Pair it with a reviewed way to mark false positives — gitleaks reads fingerprints from a `.gitleaksignore` file — so developers never respond by disabling scanning globally.

Revoke first, clean up second

Here is the mistake that keeps repeating. A cloud key is committed, removed in the next commit, and assumed safe. A fork, CI log, or local clone may already contain it, so rewriting Git history cannot invalidate the credential.

Treat a committed secret as leaked from the moment it was pushed. Revoke or rotate it first. Then decide whether rewriting history is worth the coordination cost — for a public repository it usually is, to stop future scrapers, but it is cleanup, not containment.

Commit a scanner-approved fake credential to a disposable repository and save the alert evidence. Follow the response runbook through revocation simulation, history search, and affected-system inventory. Then test a harmless look-alike value and document how a developer can resolve the false positive without disabling scanning globally.

Minimize CI authority

Give workflows narrow token permissions, protect untrusted pull requests, pin external actions, and separate testing from publishing.

CI runs code and often holds credentials. A pull request may therefore be an attempt to execute code inside your trusted environment.

Every GitHub Actions job receives a `GITHUB_TOKEN`. Its default permissions are often broader than the job needs. Declare the minimum explicitly at the top of the workflow:

```
permissions:
  contents: read
```

Now a compromised test dependency running in this job cannot push commits, open releases, or write packages with that token. A job that only runs tests needs read access and nothing else.

Treat fork pull requests as untrusted code

The standard `pull_request` trigger runs fork code without secrets and with a read-only token. That is the safe default.

The dangerous variant is `pull_request_target`. A workflow triggered with `pull_request_target` checks out code from an external contributor. That code now runs in the trusted base-repository context and may reach secrets or a write-capable token. If you must use it, never check out and execute the contributor's code in that job. This exact pattern has burned many real projects.

Pin external actions

A workflow line like `uses: someorg/setup-tool@v2` re-resolves on every run. Whoever controls that tag controls code in your CI. Pin third-party actions to a reviewed immutable commit SHA:

```
- uses: actions/checkout@b4ffde65f46336ab88eb53be808477a3936bae11 # v4.1.1
```

The comment keeps the version readable; the SHA makes it immutable. Update pins deliberately, through reviewed pull requests — Dependabot can open them for you with `package-ecosystem: "github-actions"`.

Separate testing from publishing

Tests and releases need different authority. Combining them makes every test execution a possible publishing event. Keep publish jobs in a separate workflow, gated behind protected events (a tag or release) and a protected environment that holds the release secrets. The test workflow should not even be able to see the npm token.

Map the trigger, token permissions, secrets, third-party actions, and environments for one workflow. Save a run showing that untrusted pull-request code receives no publishing credential. Then attempt a write operation from that job and capture the expected permission failure.

Check your understanding: protect source and ci

Check repository access, reviews, branches, secret scanning, CI permissions, untrusted contributions, and workflow pinning.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Build verifiable artifacts

Use clean builds, SBOMs, provenance, signatures, and hardened container images.

Build from clean inputs

Create releases in an isolated environment from reviewed source, locked dependencies, declared tools, and no hidden workstation state.

A release should come from source and declared inputs, not from whatever happened to exist on one laptop.

A release built on a laptop quietly includes an untracked generated file and a globally installed compiler. Nobody reviewing the repository can reproduce those inputs, and nobody can say whether the laptop was clean. This is why SLSA and similar frameworks push builds onto **ephemeral builders**: fresh environments that exist for one build and are destroyed after.

A minimal clean release job on GitHub Actions declares every input:

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@b4ffde65f46336ab88eb53be808477a3936bae11 # v4.1.1
      - uses: actions/setup-node@v4
        with:
          node-version-file: '.nvmrc'
      - run: npm ci
      - run: npm run build
```

The runner starts empty. The source is one exact revision. The Node version comes from a committed file, not from whatever is on someone's machine. `npm ci` installs the locked graph deterministically. Nothing undeclared can leak in, because nothing undeclared exists on the runner.

Keep release credentials out of normal test steps: the build job here holds no publishing token at all. Record the source revision and build identity with the artifact — in Actions, `github.sha` and the workflow run URL are the minimum worth saving.

Verify with a double build

Build the same revision twice in separate clean environments and compare digests:

```
shasum -a 256 dist/app.tar.gz
# 9f2c1e40b3a8...  dist/app.tar.gz
```

If the two digests match, your declared inputs fully determine the output. If they differ, compare the artifacts and explain the first meaningful difference.

Two clean builds can still differ because of timestamps, file order, or tool behavior. Reproducibility is evidence to investigate, not a promise produced by an ephemeral runner. Archive timestamps embedded by tar or zip are the most common culprit in JavaScript projects.

Build the same revision twice in separate clean environments and save both artifact digests plus the declared tool versions. If the digests differ, compare the artifacts and explain the first meaningful difference. Add an undeclared local file and prove the clean build ignores it or fails explicitly.

Create an SBOM

Generate a software bill of materials for each release so components, versions, relationships, and vulnerability impact can be queried later.

An **SBOM** (software bill of materials) is an inventory, not a certificate that software is safe. It helps answer what a release contains.

Here is the situation it exists for. A new compression-library advisory appears on Friday. The team has ten released images, and manifests alone cannot show which final images contain the affected operating-system package. With an SBOM per release, that becomes a query instead of a rebuild-and-check archaeology session.

Generate one

npm can produce an SBOM straight from the resolved dependency graph, in the standard SPDX format:

```
npm sbom --sbom-format spdx > sbom.spdx.json
```

That covers JavaScript packages. For a container image, generate the SBOM from the final image so operating-system packages and bundled binaries are included too. **syft** does this well:

```
syft ghcr.io/acme/api:1.4.2 -o spdx-json=sbom.spdx.json
#   Cataloged 212 packages
```

The count in the output usually surprises people: the application's npm tree plus every Debian or Alpine package in the base image.

Store it with the release, query it later

Store the SBOM beside the artifact, protect its integrity, and connect it to the exact artifact digest it describes. An SBOM that cannot be matched to specific released bytes answers nothing.

When Friday's advisory arrives, query instead of guessing:

```
jq '.packages[] | select(.name == "zlib") | .versionInfo' sbom.spdx.json
# "1.2.13"
```

Now you know which releases contain the affected version, and which do not.

Generate at the right moment

Timing is the common failure. An inventory generated too early — from `package.json` before install, or from source before the container build — can omit bundled or container components. Generate from the final build or the resolved graph, after the last stage that adds files.

Generate an SPDX SBOM from a final application artifact or container and save it beside the artifact digest. Locate one direct, one transitive, and one operating-system component. Then add a component during the final build stage and prove whether the SBOM includes or misses it.

Record provenance and signatures

Attach verifiable information about source, builder, workflow, and artifact identity so consumers can check where a release came from.

A checksum detects change only when the expected checksum arrives through a trusted channel. **Provenance** connects the artifact to its build process: which source revision, which builder, which workflow produced these exact bytes.

The modern tooling for this is **Sigstore**. Instead of long-lived signing keys that can be stolen, the build workflow signs with a short-lived certificate tied to its identity — for GitHub Actions, the repository and workflow path — and the signature is recorded in a public transparency log.

Publish npm packages with provenance

For npm packages this is nearly free. In the release workflow, grant an OIDC token and publish with the provenance flag:

```
permissions:
  id-token: write
  contents: read
```

```
npm publish --provenance
```

npm now attaches a Sigstore-backed attestation linking the published tarball to the exact repository, commit, and workflow run that built it. Generate attestations inside the trusted builder, and sign through a protected identity or key — never on a laptop after the fact.

Consumers can check the whole tree:

```
npm audit signatures
# audited 212 packages in 3s
# 212 packages have verified registry signatures
# 14 packages have verified attestations
```

If a package's bytes do not match what the registry signed, this command fails loudly.

A valid signature is not enough

Verification needs policy, not just cryptography. An attacker signs a package from a personal workflow using a valid identity. The signature verifies mathematically, but the artifact did not come from the protected release workflow or approved source revision.

So consumers must verify the signature, the expected project, the source revision, the builder identity, and the artifact digest. With cosign that policy is explicit:

```
cosign verify ghcr.io/acme/api:1.4.2 \
  --certificate-identity-regexp 'github.com/acme/api/.github/workflows/release.yml' \
  --certificate-oidc-issuer https://token.actions.githubusercontent.com
```

This rejects anything not signed by that specific workflow, even if the signature itself is valid.

Choose one release artifact and write the exact provenance policy a consumer should enforce. Save a successful verification showing its subject digest, source revision, and builder identity. Then verify a valid attestation from an unapproved branch or builder and prove the policy rejects it.

Harden container images

Use trusted minimal bases, pin image identity, build without secrets, scan final layers, and run with reduced runtime privileges.

A container image is a release artifact with its own operating-system packages and history. Review the final image, not only the Dockerfile.

Start with the base. Use supported base images, pin a digest where stability matters, and rebuild for security updates:

```
FROM node:22-alpine@sha256:9fcc1a6da2b9eaa4d8d8e2b6f26b2fcd8f9c1a3e5d4b7a8c2f1e0d9c8b7a6f5e
```

A tag like `node:22-alpine` is mutable; the digest is not. Pinning means base updates arrive through a reviewed pull request instead of silently on the next build.

Never bake secrets into layers

Here is the classic mistake. A Dockerfile copies `.npmrc`, installs private packages, and deletes the file in the next instruction. The credential remains recoverable from the earlier layer, because each instruction creates a layer and deletion only masks the file in later ones.

Use a BuildKit secret mount instead — the file exists during the one instruction and lands in no layer:

```
RUN --mount=type=secret,id=npmrc,target=/root/.npmrc npm ci --omit=dev
```

```
docker build --secret id=npmrc,src=.npmrc -t ghcr.io/acme/api:1.4.2 .
```

Verify it worked by inspecting the layer history:

```
docker history --no-trunc ghcr.io/acme/api:1.4.2
```

No layer should show the credential file being copied in.

Ship less, run with less

Use a multi-stage build so compilers, dev dependencies, and package caches stay in the build stage and out of the runtime stage. Then run as a non-root user when the application supports it — the official Node images include one:

```
USER node
```

A container compromise now starts without root inside the container.

One caveat: a minimal image reduces packages and findings, but it can make diagnosis harder. Keep a separate debugging path — an ephemeral debug container or a debug image variant — instead of shipping compilers and shells in every production image.

Inspect one final image for layers, packages, configured user, entry point, and embedded secrets, and save the results. Rebuild it with a secret mount and a separate runtime stage, then prove the credential is absent. Start the container as its configured non-root user and test one operation that should fail without elevated privileges.

Check your understanding: build verifiable artifacts

Check clean builds, SBOMs, provenance, signing, container bases, artifacts, and environment isolation.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Handle vulnerabilities

Scan completely, triage reachability, patch transitive code, and coordinate disclosure.

Scan the complete release

Check application dependencies, container packages, build tools, workflow components, and released artifacts without confusing scan coverage with proof of safety.

One package-manager audit sees one part of the supply chain. The shipped artifact may contain much more.

Here is the trap. A package-manager audit reports no findings, but the deployed container includes a vulnerable system library and an old workflow action. Each scanner sees only the ecosystem it understands, so ”the scan is green” quietly means ”the part we scanned is green”.

Layer the scanners

Start with the npm graph:

```
npm audit
# found 0 vulnerabilities
```

This checks your lockfile against the npm advisory database. Add **osv-scanner**, which queries the cross-ecosystem OSV database and often knows advisories npm does not:

```
osv-scanner --lockfile=package-lock.json
```

Its output is a table: one row per finding, with the OSV ID, package, installed version, and the lockfile that introduced it.

Neither tool sees the operating-system packages in your container. For the final image, use **trivy**:

```
trivy image ghcr.io/acme/api:1.4.2
# libcrypto3 (CVE-2024-…) HIGH  installed: 3.1.4-r5  fixed: 3.1.4-r6
```

That `fixed:` column tells you a base image rebuild resolves it. Finally, remember workflow components: outdated GitHub Actions are dependencies too, and Dependabot can watch them.

State the coverage boundary

Keep advisory data current — a scanner with a stale database gives stale answers. Use multiple signals where the risk warrants it, and record which ecosystems and artifact types the scan does not cover. A coverage table with five rows (source dependencies, build tools, workflow components, container packages, final artifact) makes the missing signal visible instead of assumed.

More scanners also create duplicate and false-positive work. The same CVE will appear from npm audit, osv-scanner, and trivy at once. Triage one finding once, and route each class of finding to one owning tool.

A clean scan is not proof of safety. It means no known advisory matched the components the scanner could see, today.

Create a coverage table for source dependencies, build tools, workflow components, container packages, and the final artifact. Save one scan result for each covered class and name every uncovered class. Add a known safe test fixture or outdated disposable package and prove the expected scanner detects it without treating detection as proof of exploitability.

Triage reachability and impact

Confirm affected versions, vulnerable behavior, runtime reachability, privileges, data exposure, and available fixes before prioritizing remediation.

A vulnerability identifier is the start of investigation. Product context determines urgency and containment.

Severity scores rank the bug in the abstract, not in your product. A critical advisory affects an XML parser used only by a local documentation command. Another medium advisory affects an internet-facing parser running with database access. Severity alone would rank them in the wrong operational order.

Confirm the basics first

Verify the installed version actually falls in the affected range:

```
npm ls lodash
#    report-builder@2.1.0
#      lodash@4.17.20
```

The advisory says "fixed in 4.17.21", and you run 4.17.20 — confirmed affected. Also check where it runs: a finding that disappears with `npm audit --omit=dev` lives only in development tooling, which changes the exposure story without making it automatically safe (dev tools run in CI, next to tokens).

Trace reachability

Next question: does the vulnerable function reach production? Read the advisory to learn which behavior is vulnerable — a specific function, a parsing mode, a configuration. Then check whether your code, or the dependency path into it, exercises that behavior:

```
npm explain lodash
grep -rn "template(" src/
```

If the vulnerable function is `lodash.template` and nothing in your graph calls it, the risk drops sharply. Write that evidence down; a bare "not exploitable" claim is worthless in six months.

Then consider attacker prerequisites and worst credible impact. Who can reach the vulnerable code path — anonymous internet users, or only admins? What privileges and data does the process hold when it runs?

Prioritize with evidence, not forever

Patch high-impact reachable paths quickly, but continue tracking lower-risk findings with evidence and deadlines.

Reachability evidence can justify priority, but it can become an excuse to postpone patches forever. Code changes, and last quarter's unreachable function becomes this quarter's hot path. Give every accepted delay an owner and review date.

Triage one real advisory by recording the installed version, vulnerable function, runtime path, attacker prerequisites, privileges, data exposure, and fix. Save evidence for each conclusion. Then force the vulnerable feature into a test build or disable the suspected call path and verify whether the reachability conclusion changes.

Patch transitive dependencies

Update the nearest responsible parent, use safe overrides temporarily, test behavior, and remove workaround pins when upstream releases a fix.

A vulnerable package may not appear in your manifest. Find which direct dependency introduced it.

```
npm explain qs
# qs@6.10.3
#   qs@"6.10.3" from body-parser@1.20.0
#     body-parser@"1.20.0" from express@4.18.0
```

You never installed `qs`, but `express` did, through `body-parser`. That chain tells you who is responsible for the fix.

Prefer the parent update

The cleanest fix is updating the nearest responsible parent so the graph resolves naturally:

```
npm update express
npm audit
# found 0 vulnerabilities
```

The framework's maintainers already tested their code against the newer `qs`. You inherit a combination that upstream actually supports. `npm audit fix` automates this for semver-compatible updates; treat `npm audit fix --force` with suspicion, because it will install breaking major versions to silence findings.

Overrides are a temporary tool

Sometimes the parent has no fixed release yet. npm lets you force a version anyway:

```
{
  "overrides": {
    "qs": "6.11.2"
  }
}
```

This works, and it is also how you break things. A vulnerable parser arrives through a web framework. Forcing a newer parser with an override removes the scanner warning but breaks an internal API the framework still expects. The override silenced the scanner while creating a runtime bug.

If an override is compatible and urgent, use it — with compatibility tests, a recorded owner and reason, and an expiry condition: "remove when express ships a release depending on qs >= 6.11". Forking the dependency creates a permanent maintenance obligation and should be a deliberate last resort.

Automate the routine updates

Most transitive fixes arrive through ordinary parent updates. **Dependabot** (or Renovate) turns them into reviewable pull requests:

```
version: 2
updates:
  - package-ecosystem: "npm"
    directory: "/"
    schedule:
      interval: "weekly"
```

Small weekly bumps keep you close to upstream, so an urgent security update is a patch-level hop instead of a three-major-version migration.

Trace one transitive dependency to every direct parent and save the dependency paths. Apply the smallest supported parent update and run the relevant behavior tests. If you trial an override, add an incompatible-version failure test and record its owner, reason, and removal date.

Publish a vulnerability policy

Give researchers a safe reporting channel, scope, response expectations, and coordinated disclosure process.

Someone who finds a weakness should not have to guess how to reach you. Publish a clear security contact.

The failure mode is predictable. A researcher finds an authorization flaw and reports it in a public issue because the project has no security contact. The report exposes users before maintainers can investigate. The researcher did nothing malicious — you just gave them no better path.

On GitHub the convention is a `SECURITY.md` file in the repository root. Keep it short and concrete:

```
# Security Policy

## Supported versions
Only the latest 2.x release receives security fixes.

## Reporting a vulnerability
Report privately via GitHub's "Report a vulnerability" button,
or email security@acme.dev. Do not open a public issue.

We acknowledge reports within 3 business days and aim to
release fixes within 90 days, coordinating disclosure with you.

## Scope
Do not access other users' data, run denial-of-service tests,
or use social engineering. Good-faith research within these
boundaries will not result in legal action.
```

Every element earns its place. Supported versions stop reports against code you will never patch. The private channel — enable GitHub's **private vulnerability reporting** so the button exists — keeps details out of public view while you investigate. Response expectations tell the researcher when silence means ”escalate” rather than ”ignored”.

Tell reporters what helps you reproduce the issue: affected endpoint or function, a proof-of-concept request, the observed and expected behavior.

Safe harbor cuts both ways

A policy must also protect the project from destructive testing. Clear scope and safe-harbor language help good-faith research without granting permission to access other users' data. State plainly what is out of bounds.

Then behave like the policy says. Acknowledge reports, protect reporter data, and coordinate fixes and disclosure. A policy you do not answer is worse than none: the researcher waits the promised three days, hears nothing, and publishes.

Draft `SECURITY.md` with supported versions, a monitored private contact, useful report details,

response expectations, and testing boundaries. Send a harmless test report and save proof that the responsible person receives and acknowledges it. Then test the public issue path and confirm it directs reporters to the private channel without exposing report contents.

Check your understanding: handle vulnerabilities

Check scanning, exploitability, patching, disclosure, transitive fixes, exceptions, and maintenance.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Release and respond

Protect publishing, verify deployment artifacts, and respond to compromised dependencies.

Protect publishing authority

Separate package and deployment credentials from development, use short-lived identity where available, and require protected release workflows.

Publishing credentials can replace trusted software for every consumer. Keep them out of developer laptops when the platform supports a safer path.

The threat is concrete. A long-lived registry token sits on a maintainer laptop and can publish every package in an organization. A stolen token lets an attacker replace trusted software without changing the repository. No pull request, no review, no trace in Git — just a malicious version on the registry. Several major npm compromises started exactly this way.

Audit what exists today

Start by listing the tokens that can publish right now:

```
npm token list
```

Every long-lived, broadly-scoped token in that list is an incident waiting for a laptop theft or a phished password. If a token must exist, make it a **granular access token** scoped to one package, with an expiry date — not a classic token that can touch everything you own.

Prefer identity over secrets

Trusted publishing removes the stored secret entirely. You register, on npmjs.com, which GitHub Actions workflow is allowed to publish the package. At release time the workflow proves its identity through OIDC and receives short-lived credentials for that one publish. There is no token to steal, because no token exists between releases.

Trusted publishing exchanges that reusable secret for a short-lived workflow identity. The workflow and release environment then become critical policy boundaries: protect them like the credential they now represent. Require review to change the release workflow, and put the publish job behind a protected environment:

```
jobs:
  publish:
    runs-on: ubuntu-latest
```

```
    environment: release
    permissions:
      id-token: write
```

The `release` environment can require named reviewers to approve each run, so high-impact publishing keeps a human gate.

Rotate whatever long-lived credentials remain, and test revocation before an incident — knowing the exact click that kills a token is response time you do not want to discover live.

Draw the identities from release approval to registry publication and save the effective permissions at every step. Configure or simulate a short-lived publisher limited to one package and protected environment. Then attempt publication from an unapproved branch and prove the registry or workflow rejects it.

Verify before deployment

Check artifact digest, provenance, signer, source revision, policy, and environment before promoting exactly the reviewed build.

Do not rebuild between approval and production. Promote the same verified artifact.

The failure looks harmless in a pipeline diagram. Staging passes, but production rebuilds the source an hour later. A dependency or tool has changed, so production receives bytes that never passed staging. Everything you validated — tests, scans, manual checks — validated a different artifact.

Promote by digest, not by tag

The fix is to build once and move the immutable artifact through environments. For containers, that means deploying by digest:

```
image: ghcr.io/acme/api@sha256:7d3e2c1f9a8b4e6d0c5f2a1b8e9d4c3f6a7b0e1d2c9f8a7b6e5d4c3b2a1f0e
```

A tag like `:1.4.2` can be repointed to different bytes. The digest cannot. Staging and production referencing the same digest are provably running the same release.

Make the gate verify, not trust

Before promotion, the deployment system should verify that the artifact came from the approved project and workflow, references the expected source, and matches its digest. If the build workflow attests its artifacts (with `actions/attest-build-provenance` or cosign), the check is one command:

```
gh attestation verify oci://ghcr.io/acme/api:1.4.2 --repo acme/api
#   Verification succeeded!
# sourceRepositoryURI: https://github.com/acme/api
# workflow: .github/workflows/release.yml@refs/tags/v1.4.2
```

Read that output as a policy check, not decoration: right repository, right workflow, right ref. A valid signature from an unapproved builder must fail this gate — run the verification in the deploy job and stop on non-zero exit.

Keep configuration outside the artifact

Promoting one immutable artifact keeps the reviewed bytes stable. Environment-specific configuration — URLs, feature flags, credentials — should remain separate, injected at deploy time, so

promotion does not require rebuilding the release.

Finally, record what entered production: the digest, the source revision, the provenance identity, and the configuration version. During an incident, that record is the difference between "we know exactly what is running" and guessing.

Record the artifact digest, source revision, provenance identity, and configuration version for one staging deployment. Promote that same digest and save production evidence. Then substitute an artifact with a valid signature from an unapproved builder and prove the deployment gate refuses it.

Respond to a dependency incident

Identify affected releases, stop new exposure, replace or remove the component, rotate reachable credentials, and communicate from verified facts.

A malicious or compromised dependency can affect source, build, artifacts, credentials, and users. Scope each layer.

This scenario is not theoretical — the 2021 `ua-parser-js` compromise followed exactly this shape. A package maintainer account is compromised and a malicious `postinstall` release runs in CI. It may affect developer clones, build logs, registry credentials, released artifacts, and users at different times. Your response has to walk each layer, in order.

First hour: freeze and preserve

Stop new exposure before investigating. Freeze releases and deployments, and pin or remove the affected version so no new install pulls it. Then preserve evidence — CI logs, artifacts, and their digests — because retention windows delete the proof while you sleep.

Establish the timeline from your own records

Find when the component entered your builds. The lockfile history answers precisely:

```
git log -S '"evil-parser": "2.1.4"' --oneline -- package-lock.json
# 8c1f3d2 chore: update dependencies
```

That commit's date bounds the exposure window. Query the SBOMs of your releases to list which shipped artifacts contain the version:

```
jq '.packages[] | select(.name == "evil-parser") | .versionInfo' sbom.spdx.json
```

Rotate what the package could reach

Now inspect what authority the package had where it ran. A `postinstall` script in CI could read every environment variable in the job. So the question is not "did it steal our tokens" — you usually cannot know — it is "which credentials were present in any run that executed it". Rotate all of them.

Deleting the dependency stops future execution but does not revoke credentials already copied. This is the most common incomplete response.

One nuance cuts scope honestly in both directions: a build that installed the package with lifecycle scripts disabled (`npm ci --ignore-scripts`) never executed the install-time payload. Scope every release and authority the package could reach before declaring recovery complete.

Then produce a clean release from trusted inputs, and notify affected users with specific actions and versions — verified facts, not speculation.

Write a first-hour incident timeline for a malicious package version and identify affected builds through lockfiles, SBOMs, logs, and artifact digests. Save the queries or commands that produce the affected-release list. Then include a build where the package was installed but its script was disabled and show how the impact assessment changes.

Complete the supply-chain review

Review dependency choice, source controls, CI authority, artifact evidence, vulnerability handling, publishing, and incident readiness as one system.

Supply-chain security is the path from contributor to user. A strong step cannot compensate for an untrusted handoff later.

Everything in this course protected one handoff each: dependency selection, source approval, CI execution, artifact creation, publishing, deployment. The review that ties it together is not another checklist per tool. It is tracing one real release across all of them and checking every point where authority changes hands.

Here is why the end-to-end view matters. A project protects source and signs releases, but its desktop updater downloads any file returned by one mutable URL. The final handoff bypasses every earlier control. Reviewed code, clean builds, verified signatures — and the user still executes whatever that URL serves.

Trace one release

Take your latest release and follow it through source approval, dependency resolution, build, artifact, signing, deployment, and update. At each handoff, answer three questions and write the answers down:

```
Handoff: CI build -> registry
  Identity:   who or what is trusted here?   (release workflow via OIDC)
  Evidence:   what proves the handoff?       (provenance attestation, digest)
  Revocation: how do we cut it off?          (delete trusted publisher config)
```

Repeat for every arrow in the path. Verify owners, evidence, and revocation at every handoff. The handoffs where you cannot name an identity, point to evidence, or describe revocation are your findings. Record gaps with a concrete next improvement — one sentence each, with an owner.

Fix in the right order

Two kinds of fixes come out of this review. My advice is to remove needless privileged handoffs first — a leftover deploy key, a laptop publish path, a rebuild between staging and production. Deleting a handoff is cheaper and more reliable than defending it. Then add verifiable evidence (provenance, digests, protected identities) where authority must remain.

A review like this is not a one-time event. Rerun it when the pipeline changes shape: a new registry, a new deploy target, a new update mechanism. Each new arrow in the diagram is a new handoff to name, prove, and be able to revoke.

Draw one release path from dependency selection through the user update, naming the identity, evidence, and revocation method at every handoff. Save proof for one complete path. Then replace

or compromise one test handoff and show exactly which downstream control detects or blocks it.

Check your understanding: release and respond

Check publishing authority, artifact verification, release evidence, dependency incidents, rollback, and final supply-chain review.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/software-supply-chain-security/.