

npm Supply-Chain Security

Version npm 12.2 · verified 2026-11-02

Lock, gate install scripts, delay fresh releases, publish via OIDC, harden GitHub Actions — and what the Shai-Hulud worm exploited.

Contents

Mental model 3

Lockfiles and `npm ci` 3

Install scripts: the npm 12 default 5

Release-age delays and update policy 7

Audit, signatures and SBOMs 9

Publishing: trusted publishing and staging 10

Tokens and secrets 12

Hardening GitHub Actions 14

Worked example: Shai-Hulud 16

Incident checklist 19

Further reading 20

Mental model

Installing a package runs code: install scripts execute on your laptop and in CI with whatever credentials are lying around. Every dependency is therefore a trust decision about its maintainers and everyone who can publish as them, and the lockfile is the only thing that makes that decision repeatable. Most attacks do not break cryptography; they steal a token, publish a poisoned version, and wait for `npm install` to pull it.

Lockfiles and `npm ci`

Command	Does
`npm install`	Resolves ranges, may rewrite the lockfile
`npm ci`	Installs exactly the lockfile; fails on drift
`npm install --package-lock-only`	Updates the lock without installing

Command	Does
`npm ls <pkg>`	Shows why a package is in the tree

Commit `package-lock.json` and use `npm ci` everywhere except when you are deliberately changing dependencies. `npm ci` deletes `node_modules`, never writes `package.json` or the lockfile, and exits with an error when the two disagree, so CI installs exactly what was reviewed.

```
# CI and Dockerfiles: frozen install
npm ci

# Deliberate upgrade, then review the lock diff
npm install zod@latest
git diff package-lock.json
```

> **Gotcha:** A `^1.4.0` range plus `npm install` in CI means a version published five minutes ago can land in your build. That is exactly how worm-infected versions spread — the lockfile is what stops it.

Install scripts: the npm 12 default

Setting	Effect
npm 12 default	Dependency install scripts blocked until approved
`allowScripts` in `package.json`	Per-package approvals and denials
`strict-allow-scripts=true`	Unreviewed scripts fail the install
`ignore-scripts=true`	No scripts at all, any npm version
`allow-git` / `allow-remote`	Default `none` in npm 12

npm 12 (July 2026) no longer runs `preinstall`, `install` or `postinstall` for dependencies unless the root `package.json` approves them. Installs finish with a list of skipped packages; review them, approve the few that need native builds, then rebuild. Approvals are pinned to the reviewed version by default.

```
npm install-scripts ls        # what was skipped
npm install-scripts approve sharp
npm install-scripts deny telemetry-pkg
npm rebuild sharp             # run approved
script
# package.json gains:
# "allowScripts": { "sharp@0.34.5": true,
#   "telemetry-pkg": false }
```

> **Gotcha:** On npm 11 and older, every dependency's install script still runs. Check `npm -v` in CI and on developer machines; until all are on 12, set `ignore-scripts=true` in the project `.npmrc`.

Release-age delays and update policy

Tool	Setting
npm	`min-release-age=<days>` in `.npmrc`
npm	`min-release-age-exclude[]=@yourorg/*`
Dependabot	`cooldown`, default 3 days for version updates
Dependabot	`github-actions` ecosystem for workflows

A release-age window makes npm resolve only versions older than N days, which gives the registry and scanners time to detect and remove a malicious release before you install it. Dependabot's cooldown holds update PRs back the same way. Security updates bypass

Dependabot's cooldown, so patches still arrive promptly.

```yaml
# .github/dependabot.yml
version: 2
updates:
  - package-ecosystem: npm
    directory: /
    schedule: { interval: weekly }
    cooldown:
      default-days: 7
  - package-ecosystem: github-actions
    directory: /
    schedule: { interval: weekly }
```

> **Note:** When `min-release-age` blocks `npm audit fix` from installing a fix that is too new, npm keeps the vulnerable version, warns, and exits non-zero. Add the package to `min-release-age-exclude` to take the fix.

Audit, signatures and SBOMs

Command	Answers
`npm audit --audit-level=high`	Known vulnerabilities; fails CI
`npm audit signatures`	Registry signatures and provenance valid?
`npm sbom --sbom-format cyclonedx`	What exactly ships (also `spdx`)
`npm query ":attr(scripts, [postinstall])"`	Which packages have scripts

`npm audit` only knows published advisories, so it catches yesterday's problem, not today's malware. `npm audit signatures` verifies that what you downloaded is what the registry signed, including provenance attestations. Generate an SBOM per release so you can answer "are we affected?" in minutes when the next advisory lands.

```
npm audit --audit-level=high
npm audit signatures
npm sbom --sbom-format cyclonedx \
  --omit dev > sbom.cdx.json
```

> **Tip:** Run `npm audit signatures` in CI after `npm ci`. It is cheap, needs no account, and fails when a package's signature or provenance does not verify.

Publishing: trusted publishing and staging

Mechanism	Removes
Trusted publishing (OIDC)	Long-lived publish tokens in CI
Provenance	Doubt about which commit built it
`npm stage publish`	Publishing without a human 2FA approval

Mechanism	Removes
Stage-only token	Direct publish from a leaked token

Trusted publishing lets GitHub Actions, GitLab CI/CD or CircleCI publish with a short-lived OIDC credential instead of a stored token; it needs npm 11.5.1+ and Node 22.14+, and generates provenance automatically from GitHub and GitLab. Staged publishing goes further: CI uploads a version that stays invisible until a maintainer approves it with 2FA.

```
on: { release: { types: [published] } }
permissions:
  id-token: write   # OIDC for npm
  contents: read
jobs:
  publish:
    runs-on: ubuntu-latest
    steps:
```

```
      - uses: actions/checkout@v7
      - uses: actions/setup-node@v7
        with:
          node-version: "24"
          registry-url: https://registry.npmjs.org
      - run: npm ci
      - run: npm stage publish  # or npm publish
```

> **Warning:** Do not restore a dependency cache in release workflows — npm's own example sets `package-manager-cache: false`. A poisoned cache from a pull-request run becomes part of what you publish.

Tokens and secrets

Practice	Why
Granular, scoped, expiring tokens	Limits what a stolen token can do

Practice	Why
2FA on every maintainer account	Stops phished-password publishing
Read-only token for CI installs	Install never needs write
Secret scanning with push protection	Blocks commits that contain keys
OIDC to cloud providers	No long-lived cloud keys in CI

npm revoked all classic tokens in December 2025; granular write tokens now have capped lifetimes and require 2FA unless created with “bypass 2FA”. Direct publishing with bypass-2FA tokens is scheduled for removal in January 2027, with stage-only tokens as the replacement. Audit what exists today.

```
npm token list            # every token, with
scope
```

```
npm token revoke "$TOKEN_ID"  # kill one
gh secret list --repo acme/web
```

> **Gotcha:** A developer laptop is part of the supply chain. A `~/.npmrc` with a write token and a `gh` login is exactly what a malicious install script harvests — log in only when publishing, and keep those tokens off CI images.

Hardening GitHub Actions

Control	How
Least-privilege `GITHUB_TOKEN`	`permissions: {}` at top, grant per job
Pin third-party actions	Full commit SHA, or `actions.lock`
No checkout of fork code in privileged triggers	Avoid `pull_request_target`
Untrusted input via `env`, never `${{ }}` in `run`	Stops script injection

Control	How
`persist-credentials: false`	Token not left in `.git/config`

A tag like `@v7` can be moved to malicious code by whoever controls the action's repository; a 40-character commit SHA cannot. GitHub's newer alternative is a workflow lockfile, `.github/workflows/actions.lock`, generated by the `gh actions-lock` extension, which pins transitive actions too. Expressions inside `run:` are pasted into the script before it runs, so a pull-request title can become shell code.

```
permissions: {}
jobs:
  test:
    runs-on: ubuntu-latest
    permissions: { contents: read }
    env:
      TITLE:
```

```
${{ github.event.pull_request.title }}
    steps:
      - uses: actions/checkout@v7 # SHA: see Tip
        with: { persist-credentials: false }
      - run: echo "$TITLE"  # not ${{ ... }}
```

> **Tip:** Resolve a tag to its SHA with `git ls-remote https://github.com/actions/` , pin `checkout refs/tags/v7.0.1` that, and keep the tag in a trailing comment so Dependabot can still propose updates.

Worked example: Shai-Hulud

Wave	Entry	Spread and payload
Sep 2025	Hijacked maintainer accounts; `postinstall`	Stole tokens, republished victims' packages

Wave	Entry	Spread and payload
Nov 2025 (“2.0”)	`preinstall` loading Bun	Secrets to public repos; rogue self-hosted runners
2.0 fallback	No credentials found	Tried to wipe the home directory

The first wave was a self-replicating worm: CISA counted over 500 compromised packages. Its install script harvested credentials, posted them to a public GitHub repository created under the victim’s account, then used the victim’s npm token to publish infected versions of their other packages. The November wave ran earlier (`preinstall`), hid its payload behind a freshly installed Bun runtime, and registered infected machines as self-hosted runners with a backdoor workflow triggered by GitHub Discussions.

```
OWNER=acme
# repos the worm created to dump secrets
gh repo list "$OWNER" --limit 1000 | grep -i hulud
# runners you never registered
gh api "orgs/$OWNER/actions/runners" \
  --jq '.runners[].name'
# workflows added recently
git log --since=2025-09-01 --diff-filter=A \
  --name-only -- .github/workflows
```

> **Note:** Every control above breaks a step of this chain: a lockfile delays the new version, blocked install scripts stop execution, release-age windows outlast the takedown, and trusted publishing leaves no npm token to steal.

Incident checklist

First hour	Then
Freeze deploys and dependency updates	Identify affected versions via SBOM
Revoke npm, GitHub and cloud tokens	Reinstall from a known-good lockfile
Remove unknown runners and workflows	Rotate every secret the host could read
Check for new public repos and packages	Publish clean versions; notify users

Assume every secret readable by the install step is stolen: environment variables, `~/.npmrc`, cloud CLI credentials, SSH keys, CI secrets. Revoke before you investigate, because the worm uses those tokens within minutes, then rebuild from known-good inputs.

```
npm token list
npm token revoke "$TOKEN_ID"
gh auth logout          # then log in again
```

```
# restore a lockfile from before the incident
git checkout "$GOOD_SHA" -- package-lock.json
npm cache clean --force
rm -rf node_modules && npm ci
```

> **Warning:** Rotating a secret without revoking the old one leaves the attacker's copy working. Revoke first, then issue the replacement.

Further reading

- npm Docs — Trusted publishing
- npm CLI — install-scripts
- GitHub Docs — Secure use reference for Actions
- OpenSSF Scorecard
- CISA alert on the npm worm
- OWASP NPM Security Cheat Sheet