PuigFlows

Seguridad de la Cadena de Suministro: Lockfiles, Enfriamiento de Versiones, Trusted Publishing con OIDC y Firmas con Sigstore

Guía práctica para cerrar las cinco puertas por las que entra un ataque a la cadena de suministro: instalación estricta desde el lockfile, bloqueo de los scripts postinstall que propagaron el gusano Shai-Hulud, periodos de enfriamiento (min- release- age, minimumReleaseAge, npmMinimalAgeGate) coordinados con Dependabot y…

Guía · Nivel intermedio · 42 min de lectura
Actualizado el 2026-08-17 · puigflows.com

Un gusano, 700 paquetes y una lección incómoda

En septiembre de 2025 un correo de phishing que imitaba una alerta de seguridad de npm consiguió las credenciales de un mantenedor. No hubo un desbordamiento de búfer ni una cadena de exploits sofisticada: alguien escribió su contraseña en un formulario falso. A partir de ahí, el gusano *Shai-Hulud* hizo el resto. Su primera oleada comprometió cientos de paquetes; la segunda, en noviembre de ese mismo año, superó los 700 paquetes afectados y generó más de 27.000 repositorios públicos con secretos robados de las máquinas de quienes instalaron una versión envenenada.

El mecanismo de propagación tampoco era exótico. Era un `postinstall`: un script que tu gestor de paquetes ejecuta, sin preguntarte nada, la primera vez que instalas una dependencia. Ese script buscaba credenciales en el entorno, las exfiltraba y las usaba para publicar versiones maliciosas de otros paquetes del mismo mantenedor. Autorreplicación, sin intervención humana después del primer clic.

Lo incómodo del incidente es que casi todo el daño se podía haber evitado con configuración. No con una herramienta cara ni con un equipo de seguridad dedicado: con cinco o seis líneas repartidas entre un `.npmrc`, un fichero de workflow y una política de admisión. Esta guía recorre exactamente esas líneas.

El modelo de amenaza: cinco puertas, no una

Cuando alguien dice "seguridad de la cadena de suministro" suele estar pensando en una sola cosa: un paquete malicioso en npm o PyPI. Es la puerta más famosa, pero no es la única, y protegerla sola deja las demás abiertas. Conviene enumerarlas, porque cada una tiene una defensa distinta y barata.

1. **La instalación.** Descargas código de terceros y tu gestor de paquetes lo ejecuta. Vector: typosquatting, cuentas de mantenedor comprometidas, scripts de ciclo de vida.
2. **El build.** Tu CI ejecuta acciones y contenedores de terceros con acceso a tus secretos. Vector: una acción etiquetada como `v3` que un día apunta a otro commit.
3. **La publicación.** Un token de larga vida guardado en un secreto de repositorio puede publicar en tu nombre para siempre. Vector: filtración del token, phishing al mantenedor.
4. **La distribución.** Entre que construyes el artefacto y alguien lo despliega, ¿quién garantiza que es el mismo binario? Vector: registro comprometido, etiqueta mutable reapuntada.
5. **El despliegue.** Tu clúster arranca cualquier imagen que le pidan. Vector: cualquiera de los cuatro anteriores que no hayas detectado.

Las secciones siguientes cierran cada puerta, ordenadas por relación coste/beneficio. Si sólo tienes una tarde, haz la primera y la segunda.

Puerta 1 — La instalación: lockfiles, scripts y enfriamiento

Instala desde el lockfile, siempre

La diferencia entre `npm install` y `npm ci` parece cosmética y no lo es. `npm install` resuelve versiones,

puede actualizar el lockfile y sigue adelante si `package.json` y `package-lock.json` no coinciden. `npm ci` borra `node_modules`, instala exactamente lo que dice el lockfile y falla si hay discrepancia. En CI, la primera opción significa que dos ejecuciones del mismo commit pueden instalar árboles de dependencias distintos.

```
# En CI, nunca esto:
npm install

# Siempre esto:
npm ci

# Equivalentes en otros gestores:
pnpm install --frozen-lockfile
yarn install --immutable
uv sync --locked          # Python: falla si uv.lock está desactualizado
```

El lockfile sólo sirve si guarda hashes de integridad y si nadie lo regenera a la ligera. Trátalo como código y revisa sus diffs en las pull requests. Un cambio de una línea en `package.json` que produce cuatrocientas líneas nuevas de lockfile merece una mirada.

Deja de ejecutar código arbitrario al instalar

Los scripts `preinstall`, `install` y `postinstall` son la vía por la que Shai-Hulud se propagó. Algunos paquetes los necesitan de verdad (compilar bindings nativos, descargar un binario), pero la inmensa mayoría no. La postura correcta es apagarlos por defecto y aprobar excepciones de forma explícita.

```
# .npmrc en la raíz del proyecto
ignore-scripts=true
```

El inconveniente práctico de npm es que no tiene lista de permitidos: o todos o ninguno. La solución es reconstruir a mano lo poco que lo necesite, justo después de instalar:

```
npm ci                       # sin scripts, gracias a .npmrc
npm rebuild esbuild sharp    # sólo lo que has revisado y aprobado
```

pnpm lo resolvió mejor. Desde pnpm 10 los scripts de las dependencias están bloqueados por defecto, y en pnpm 11 la lista de permitidos se declara con `allowBuilds`, que sustituye al antiguo `onlyBuiltDependencies`:

```
# pnpm-workspace.yaml
allowBuilds:
  - esbuild
  - sharp
```

En Python el equivalente es evitar las distribuciones de código fuente, que ejecutan `setup.py` durante la instalación. Los wheels, en cambio, se descomprimen sin ejecutar nada:

```
# Falla si algún paquete sólo ofrece sdist, en lugar de compilarlo en silencio
uv pip install --only-binary=:all: -r requirements.txt
```

El periodo de enfriamiento: la mejor relación coste/beneficio que existe

Casi todas las versiones maliciosas se detectan y se retiran en cuestión de horas. El ataque depende de que tu CI las consuma inmediatamente después de publicarse. Si tu instalador se niega a resolver versiones publicadas hace menos de veinticuatro horas, la mayor parte de estos incidentes te pasan por encima sin tocarte.

Los gestores de Node lo soportan hoy, cada uno con un nombre y una unidad distintos. Este es el mapa que evita perder una tarde:

- **npm** (CLI 11.10.0 o superior): `min-release-age`, en **días**.
- **pnpm** (desde 10.16): `minimumReleaseAge`, en **minutos**. Desde pnpm 11.0 viene activado por defecto a 1440.
- **Yarn** (Berry 4.10+): `npmMinimalAgeGate`, en **minutos**, y como número: las cadenas tipo "7d" tienen un bug de parseo conocido.

```
# npm — .npmrc — unidad: DÍAS
min-release-age=1
```

```
# pnpm 11 — pnpm-workspace.yaml — unidad: MINUTOS
minimumReleaseAge: 1440
minimumReleaseAgeExclude:
  - '@mi-org/*'          # paquetes internos que no deben esperar
```

```
# Yarn Berry 4.10+ — .yarnrc.yml — unidad: MINUTOS
npmMinimalAgeGate: 1440
npmPreapprovedPackages:
  - "@mi-org/*"
```

Dos detalles que casi todo el mundo pasa por alto. El primero: el enfriamiento se aplica al instalar, no al proponer actualizaciones, así que hay que configurarlo también en el bot de dependencias o te abrirá pull requests que después no se pueden instalar. Renovate lo expone como `minimumReleaseAge` y Dependabot como `cooldown`, que desde julio de 2026 trae ya un valor por defecto.

```
# .github/dependabot.yml
version: 2
updates:
  - package-ecosystem: npm
    directory: "/"
    schedule:
      interval: weekly
    cooldown:
      default-days: 7
      semver-major-days: 14
```

El segundo: necesitas una vía de escape para el parche de seguridad urgente, y existe. En npm se salta el gate por invocación con `npm i --min-release-age 0 paquete@version`. Que exista esa salida es justo lo que hace viable poner el gate en primer lugar.

Un guardarraíl que falla en CI

La configuración se olvida y se revierte. Conviene tener una comprobación automática de que ninguna dependencia nueva ha introducido un script de instalación sin que nadie lo notara. Este script lee el lockfile de npm (formato v2/v3), busca los paquetes marcados con `hasInstallScript` y falla si alguno no está en tu lista aprobada:

```
// scripts/audit-install-scripts.mjs
import { readFile } from 'node:fs/promises';

// Lista revisada a mano. Añadir algo aquí exige justificarlo en la PR.
const APROBADOS = new Set(['esbuild', 'sharp', '@swc/core']);

const lock = JSON.parse(await readFile('package-lock.json', 'utf8'));
const sospechosos = [];

for (const [ruta, meta] of Object.entries(lock.packages ?? {})) {
  if (!meta.hasInstallScript) continue;
  const nombre = ruta.split('node_modules/').pop();
  if (APROBADOS.has(nombre)) continue;
  sospechosos.push(nombre + '@' + (meta.version ?? '?'));
}

if (sospechosos.length > 0) {
  console.error('Dependencias con scripts de instalación NO aprobados:');
  for (const s of sospechosos) console.error('  - ' + s);
  console.error('Revisa qué hacen antes de añadirlos a APROBADOS.');
  process.exit(1);
}

console.log('OK: ninguna dependencia ejecuta scripts de instalación sin aprobar.');
```

Se engancha como un paso más del pipeline, junto al linter. Cuesta cinco minutos y detecta justo la clase de cambio que nadie mira en un diff de cuatrocientas líneas.

Puerta 2 — El build: fija las acciones por SHA y recorta permisos

Una etiqueta no es una versión

Cuando escribes `uses: alguien/accion@v3` estás confiando en una etiqueta de Git. Las etiquetas son punteros móviles: el propietario del repositorio puede reapuntar `v3` a cualquier commit en cualquier momento, y tu workflow ejecutará ese código nuevo sin que tú cambies una línea. Si esa cuenta se compromete, el atacante hereda tus secretos en la siguiente ejecución. Ha pasado, y con acciones muy populares.

La única referencia inmutable en Git es el SHA completo del commit. Fijarlo cuesta escribir cuarenta caracteres feos, así que se acompaña de un comentario con la versión legible:

```
# Mutable: mañana v5 puede apuntar a otro commit
- uses: actions/checkout@v5
```

```
# Inmutable: este commit exacto y ningún otro
- uses: actions/checkout@SHA_COMPLETO_DE_40_CARACTERES # v5.0.0
```

Nadie sostiene esto a mano durante mucho tiempo, y no hace falta. La herramienta `pinact` reescribe todas las etiquetas de tus workflows a SHAs reales con el comentario correcto, y Dependabot entiende los SHAs fijados: abre pull requests que actualizan el SHA y el comentario a la vez. La combinación te da inmutabilidad sin quedarte congelado en versiones antiguas.

```
# Fija todas las acciones del repositorio de una vez
pinact run

# Comprueba en CI que nadie ha vuelto a introducir una etiqueta
pinact run --check

# Y deja que Dependabot proponga las subidas:
# .github/dependabot.yml -> package-ecosystem: "github-actions"
```

Permisos mínimos, y sólo donde hacen falta

Por defecto, el `GITHUB_TOKEN` de un workflow puede tener permisos de escritura sobre el repositorio. Un paso comprometido en cualquier job puede entonces empujar commits, crear releases o modificar los propios workflows. La regla es declarar el mínimo a nivel de workflow y ampliarlo únicamente en el job que lo necesite:

```
name: ci

on:
  pull_request:
  push:
    branches: [main]

# Todo el workflow arranca en sólo lectura
permissions:
  contents: read

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@SHA_COMPLETO_DE_40_CARACTERES # v5.0.0
      - uses: actions/setup-node@SHA_COMPLETO_DE_40_CARACTERES # v5.0.0
        with:
          node-version: 22
          cache: npm
      - run: npm ci
      - run: node scripts/audit-install-scripts.mjs
      - run: npm test

  release:
    if: github.ref == 'refs/heads/main'
    needs: test
```

```
    runs-on: ubuntu-latest
    permissions:
      contents: read
      id-token: write        # sólo este job puede pedir un token OIDC
      attestations: write    # sólo este job puede firmar artefactos
    steps:
      - uses: actions/checkout@SHA_COMPLETO_DE_40_CARACTERES # v5.0.0
      # ... construir y publicar
```

Ese `id-token: write` es la llave de la puerta 3. Ponerlo a nivel de workflow, donde también corren los tests, significa que cualquier dependencia de test puede pedir un token de identidad federada. Ponlo sólo en el job de release.

El disparador que regala el repositorio

Merece un aviso propio. `pull_request_target` ejecuta el workflow con los secretos del repositorio base, pero es habitual combinarlo con un checkout de la rama de la pull request. Eso equivale a ejecutar código de un desconocido con tus credenciales cargadas en el entorno.

```
# PELIGRO: código no confiable + secretos del repositorio
on: pull_request_target
jobs:
  build:
    steps:
      - uses: actions/checkout@SHA_COMPLETO_DE_40_CARACTERES
        with:
          ref: ${{ github.event.pull_request.head.sha }}   # código del atacante
      - run: npm ci && npm run build                       # con tus secretos
```

Si necesitas datos de la pull request, construye con un `pull_request` normal (que no recibe secretos) y haz las acciones privilegiadas en un workflow separado disparado por `workflow_run`, sin ejecutar nunca el código del PR en el contexto privilegiado.

Puerta 3 — La publicación: adiós a los tokens de larga vida

El patrón clásico es generar un token de API en npm o PyPI, pegarlo en un secreto del repositorio y olvidarlo. Ese token no caduca. Si se filtra —en un log, en una captura de pantalla, en un correo de phishing — el atacante puede publicar versiones de tu paquete hasta que alguien lo note y lo revoque a mano. Exactamente el escenario de Shai-Hulud.

Trusted Publishing elimina el token. En su lugar, el registro guarda una configuración que dice, literalmente: "acepto publicaciones de este paquete si vienen del workflow `release.yml` del repositorio `mi-org/mi-paquete`, en el entorno `pypi`". GitHub emite un token OIDC de vida muy corta que demuestra ese origen y el registro lo canjea por una credencial de publicación efímera. No hay nada que filtrar porque no hay nada guardado. Es el estándar de publicadores de confianza de la OpenSSF, y hoy lo implementan PyPI, npm, RubyGems y otros.

PyPI

Primero configuras el publicador de confianza en la web de PyPI: propietario, repositorio, nombre del fichero de workflow y entorno. Después, el workflow no contiene ni un secreto:

```
# .github/workflows/release.yml
name: release

on:
  release:
    types: [published]

permissions:
  contents: read

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@SHA_COMPLETO_DE_40_CARACTERES # v5.0.0
      - uses: astral-sh/setup-uv@SHA_COMPLETO_DE_40_CARACTERES # v6
      - run: uv build
      - uses: actions/upload-artifact@SHA_COMPLETO_DE_40_CARACTERES # v4
        with:
          name: dist
          path: dist/

  publish:
    needs: build
    runs-on: ubuntu-latest
    environment: pypi          # permite exigir aprobación manual antes de publicar
    permissions:
      id-token: write          # el ÚNICO permiso necesario: no hay token de PyPI
    steps:
      - uses: actions/download-artifact@SHA_COMPLETO_DE_40_CARACTERES # v4
        with:
          name: dist
          path: dist/
      - uses: pypa/gh-action-pypi-publish@SHA_COMPLETO_DE_40_CARACTERES # v1.13.0
```

Separar `build` de `publish` no es adorno: el job que tiene `id-token: write` no ejecuta código del proyecto, sólo mueve ficheros ya construidos. Y el `environment: pypi` te permite exigir una aprobación humana en GitHub antes de que ese job arranque.

npm

npm añadió Trusted Publishing en 2025 siguiendo el mismo estándar. Configuras el publicador de confianza en la página del paquete y el workflow queda así:

```
  publish:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      id-token: write
    steps:
```

```
      - uses: actions/checkout@SHA_COMPLETO_DE_40_CARACTERES # v5.0.0
      - uses: actions/setup-node@SHA_COMPLETO_DE_40_CARACTERES # v5.0.0
        with:
          node-version: 22
          registry-url: https://registry.npmjs.org
      - run: npm ci
      - run: npm publish        # sin NODE_AUTH_TOKEN: la identidad viene del OIDC
```

Como efecto secundario, npm adjunta automáticamente la procedencia del paquete, que es de lo que trata la siguiente sección. Si todavía dependes de un token clásico, al menos usa `npm publish --provenance` y limita el alcance del token a un único paquete.

Puerta 4 — La procedencia: atestaciones y SBOM

Hasta aquí has reducido la probabilidad de que entre algo malo. La procedencia responde a otra pregunta, la que aparece durante un incidente: *este binario que está corriendo en producción, ¿de qué commit salió, qué workflow lo construyó y qué dependencias lleva dentro?* Sin una respuesta firmada, la investigación se convierte en arqueología.

Atestaciones de procedencia con GitHub Actions

Una atestación de artefacto es una declaración firmada, en formato in-toto, que vincula el digest de un artefacto con el workflow que lo produjo: repositorio, commit, fichero de workflow, evento que lo disparó y claims del token OIDC. Se firma sin claves (keyless): Sigstore emite un certificado de vida corta ligado a la identidad del workflow y registra la firma en un log público de transparencia. Con esto solo llegas al Build Level 2 de SLSA v1.0, y con un workflow reutilizable bien aislado, al Level 3.

```
  release:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      id-token: write
      attestations: write
      packages: write
    steps:
      - uses: actions/checkout@SHA_COMPLETO_DE_40_CARACTERES # v5.0.0
      - run: npm ci && npm run build

      # 1) Procedencia del artefacto
      - uses: actions/attest-build-provenance@SHA_COMPLETO_DE_40_CARACTERES # v3
        with:
          subject-path: 'dist/**/*.tgz'

      # 2) SBOM: inventario de todo lo que hay dentro
      - name: Generar SBOM
        run: |
          curl -sSfL https://raw.githubusercontent.com/anchore/syft/main/install.sh | sh -s -- -
b /usr/local/bin
          syft scan dir:. -o spdx-json=sbom.spdx.json
```

```
      - uses: actions/attest-sbom@SHA_COMPLETO_DE_40_CARACTERES # v3
        with:
          subject-path: 'dist/**/*.tgz'
          sbom-path: 'sbom.spdx.json'
```

Un matiz que se escapa a menudo: el SBOM que generas *después* de instalar dependencias describe lo que realmente hay en el árbol, incluidas las transitivas. Un SBOM generado desde `package.json` describe lo que creías tener. La diferencia entre ambos es exactamente donde viven las sorpresas.

Imágenes de contenedor: firma el digest, nunca la etiqueta

Las etiquetas de imagen son mutables. `:1.4.2` puede apuntar mañana a otra cosa; `@sha256:...` no. Firmar la etiqueta te da una garantía que se evapora en el momento en que alguien vuelve a empujar. Con cosign en modo keyless:

```
# Publica y captura el digest exacto (no la etiqueta)
DIGEST=$(docker buildx build --push -t ghcr.io/mi-org/app:1.4.2 --metadata-file meta.json . && jq
-r '."containerimage.digest"' meta.json)

# Firma el digest. En Actions no hay que pasar claves: usa el OIDC del runner.
cosign sign --yes ghcr.io/mi-org/app@${DIGEST}

# Adjunta el SBOM como atestación firmada del mismo digest
syft scan ghcr.io/mi-org/app@${DIGEST} -o spdx-json=sbom.spdx.json
cosign attest --yes --predicate sbom.spdx.json --type spdxjson ghcr.io/mi-org/app@${DIGEST}
```

Puerta 5 — El despliegue: verificar no es opcional

Aquí está el error que invalida todo el trabajo anterior: **verificar la firma sin verificar la identidad**. En el modelo keyless, cualquiera puede firmar cualquier cosa; la firma sólo dice "alguien con una identidad OIDC firmó esto". Si no compruebas *qué* identidad, un atacante firma su imagen maliciosa con su propia cuenta de GitHub y tu verificación pasa.

Por eso `cosign verify` exige que declares la identidad esperada y el emisor:

```
# Correcto: firma + identidad del firmante + emisor del certificado
cosign verify ghcr.io/mi-org/app@sha256:abc123... --certificate-identity-regexp '^https://
github.com/mi-org/app/.github/workflows/release.yml@refs/tags/v.*' --certificate-oidc-issuer
'https://token.actions.githubusercontent.com'
```

Con el CLI de GitHub la comprobación equivalente es más corta, porque el repositorio ya delimita la identidad:

```
# Artefacto local
gh attestation verify ./dist/app-1.4.2.tgz --repo mi-org/app

# Imagen de contenedor
gh attestation verify oci://ghcr.io/mi-org/app:1.4.2 --repo mi-org/app
```

```
# Verificar el SBOM en concreto exige indicar el tipo de predicado
gh attestation verify oci://ghcr.io/mi-org/app:1.4.2 --repo mi-org/app --predicate-type https://
spdx.dev/Document
```

Que lo verifique el clúster, no una persona

Un comando en la terminal es una prueba, no un control. En producción la verificación tiene que ser una regla de admisión: Kubernetes rechaza el Pod si la imagen no está firmada por quien debe. Con Kyverno:

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: exigir-imagenes-firmadas
spec:
  validationFailureAction: Enforce
  webhookTimeoutSeconds: 30
  rules:
    - name: verificar-procedencia-ghcr
      match:
        any:
          - resources:
              kinds: [Pod]
              namespaces: [production]
      verifyImages:
        - imageReferences:
            - "ghcr.io/mi-org/*"
          mutateDigest: true      # reescribe la etiqueta al digest verificado
          verifyDigest: true
          attestors:
            - entries:
                - keyless:
                    subject: "https://github.com/mi-org/*/.github/workflows/release.yml@*"
                    issuer: "https://token.actions.githubusercontent.com"
                    rekor:
                      url: https://rekor.sigstore.dev
```

Fíjate en `mutateDigest: true`. Kyverno resuelve la etiqueta a un digest, lo verifica y reescribe el manifiesto para que el Pod arranque ese digest concreto. Sin eso, existe una ventana entre la verificación y el arranque en la que la etiqueta puede cambiar. Es una condición de carrera pequeña y perfectamente explotable.

Despliégalo primero en modo `Audit` durante una o dos semanas, mira qué imágenes fallarían, arregla los pipelines que las producen y sólo entonces pásalo a `Enforce`. Activar Enforce de golpe en un clúster con imágenes heredadas es la forma más rápida de que el equipo desactive la política.

Más allá de npm: las mismas cinco puertas en Python, Go, Rust y Java

Los titulares se los lleva npm porque es donde hay más volumen y más ruido, pero el modelo de amenaza

no cambia al cambiar de lenguaje. Lo que cambia es qué palanca existe en cada gestor de paquetes, cómo se llama y —esto es lo importante— cuál viene desactivada de fábrica. Este es el mapa que reviso cuando entro en un repositorio que no es JavaScript.

Python: la wheel es datos, el sdist es código

La distinción que casi nadie tiene interiorizada: instalar una *wheel* es descomprimir un zip y copiar ficheros; instalar un *sdist* ejecuta el `setup.py` o el backend de build del paquete en tu máquina, con tus permisos y tus variables de entorno. Es el equivalente exacto del `postinstall` de npm, sólo que sin interruptor para apagarlo. La forma de apagarlo es negarte a construir desde fuente.

```
# Sólo binarios: si no hay wheel para tu plataforma, falla en vez de ejecutar código
pip install --only-binary=:all: -r requirements.txt

# Instalación con hashes obligatorios: cualquier artefacto que no case, aborta
pip install --require-hashes -r requirements.txt

# Genera ese requirements.txt con hashes desde tu fichero de dependencias directas
uv pip compile requirements.in --generate-hashes -o requirements.txt

# Con uv como gestor de proyecto, el equivalente estricto de npm ci:
uv sync --locked --no-install-project
```

`--require-hashes` tiene un efecto secundario muy útil: activa un modo estricto en el que **toda** dependencia, incluidas las transitivas, debe estar fijada con `==` y con hash. No puedes olvidarte de una a medias. Si tu pipeline no lo usa, tu `requirements.txt` es una lista de deseos, no un lockfile.

El enfriamiento en Python vive en la configuración de uv o de pip-audit según la herramienta, y en Renovate con `minimumReleaseAge`, que funciona igual para PyPI que para npm. Y si publicas paquetes, PyPI lleva desde 2023 soportando Trusted Publishing con OIDC y desde 2024 emite atestaciones de procedencia verificables: es el mismo movimiento que en npm, con el mismo beneficio de no tener ningún token de larga vida que robar.

Go: el mejor control por defecto de todo el sector, y el flag que lo apaga

Go hizo dos cosas que nadie más hizo. La primera: no hay scripts de instalación. Descargar un módulo no ejecuta nada; el código sólo corre cuando compilas y ejecutas. La segunda: `go.sum` no es sólo un fichero local de hashes, es una comprobación contra `sum.golang.org`, una base de datos de checksums *append-only* y auditable que Google opera desde 2019. Si alguien reescribe la v1.4.2 de un módulo después de publicarla, la discrepancia sale a la luz en la siguiente descarga de cualquiera del mundo, no sólo de quien tenía el hash antiguo.

```
# Comprueba que lo que hay en el módulo cache coincide con go.sum
go mod verify

# En CI: falla si el build necesitara tocar go.mod o go.sum
go build -mod=readonly ./...
go test -mod=readonly ./...
```

```
# Fija también la toolchain: sin esto, el compilador es una dependencia flotante
export GOTOOLCHAIN=go1.24.5
```

El fallo recurrente está en la configuración corporativa. `GOPRIVATE`, `GONOSUMDB` y `GONOSUMCHECK` aceptan patrones glob, y en cuanto alguien escribe `GOPRIVATE=*` o `GONOSUMDB=github.com/*` para desatascar un proxy interno, la verificación contra la base de datos de transparencia desaparece para todo. Se estima que en torno a una de cada seis pipelines lleva alguna variable que desactiva la comprobación, casi siempre puesta hace años por alguien que ya no está. Revisa ese valor y acótalo al prefijo mínimo de tus módulos privados:

```
# Mal: desactiva la verificación para medio internet
GOPRIVATE=github.com/*

# Bien: sólo tu organización
GOPRIVATE=github.com/mi-empresa/*
GONOSUMDB=github.com/mi-empresa/*
```

Rust: el lockfile es fácil, el build.rs no

Cargo tiene lockfile determinista y `cargo build --locked` hace lo que esperas: falla si el `Cargo.lock` necesitaría cambiar. El problema de Rust es `build.rs`: un fichero de Rust arbitrario que se compila y se ejecuta en tu máquina antes que el *crate* real. Es exactamente el `postinstall`, y a diferencia de npm no existe un `ignore-scripts` equivalente, porque muchos crates legítimos lo necesitan para enlazar con C.

```
# Build reproducible: aborta si el lockfile no está al día
cargo build --locked --release

# Vulnerabilidades conocidas contra la base RustSec
cargo audit

# Política: licencias, fuentes permitidas, crates duplicados y avisos
cargo deny check

# Auditoría humana con confianza delegada entre organizaciones
cargo vet check
```

`cargo-vet` es la pieza distinta y merece un párrafo. En lugar de escanear en busca de CVEs conocidos, mantiene un registro de qué versiones de qué crates ha revisado alguien en quien confías —tu equipo, Mozilla, Google, Bytecode Alliance— y te obliga a cerrar el hueco cuando entra una versión que nadie ha mirado. Es caro de arrancar y no es para todo el mundo; en organizaciones con requisitos regulatorios serios es la única herramienta que responde a la pregunta *¿alguien ha leído este código?*.

Java: verificación de dependencias en Gradle

Gradle tiene un mecanismo de verificación potente y muy poco usado: un fichero `gradle/verification-metadata.xml` con los checksums —y opcionalmente las firmas PGP— de cada artefacto que tu build puede descargar. Una vez existe, cualquier artefacto que no case detiene el build.

```
# Genera el fichero inicial a partir de lo que resuelve tu build hoy
./gradlew --write-verification-metadata sha256,pgp help

# A partir de aquí, cualquier build verifica; usa --refresh-dependencies para regenerar
./gradlew build
```

Dos avisos prácticos. Uno: genera el fichero con la caché limpia, porque con dependencias ya cacheadas Gradle omite entradas y acabas con una verificación llena de agujeros. Dos: en versiones antiguas (6.2 a 7.4.2) había casos en los que Gradle aceptaba un artefacto si la verificación de firma no podía completarse; 7.5 lo corrigió obligando a caer al checksum. Si mantienes un build viejo, esa es una razón concreta para subir.

Contenedores: la etiqueta base también flota

Todo lo anterior se desmorona si tu `Dockerfile` empieza con `FROM node:22-alpine`. Esa etiqueta apunta a una imagen distinta cada semana, y lo que se construyó ayer no es lo que se construye hoy. Fíjala por digest, igual que fijas las acciones por SHA.

```
# Flotante: reproducibilidad cero
FROM node:22-alpine

# Inmutable: este contenido exacto de imagen
FROM node:22-alpine@sha256:5340cbfc2df14331ab021555fdd9f83f072ce811488e705b0e736b11adeec4bb

# Y deja que Dependabot suba el digest con una pull request revisable
```

Dependabot y Renovate entienden los digests de imagen igual que las versiones de paquete: te abren una pull request cuando hay una imagen base nueva, con el enlace a las notas de la versión. Ganas reproducibilidad sin perder actualizaciones.

La puerta que no está en el diagrama: el nombre del paquete

Las cinco puertas anteriores asumen que sabes qué paquete quieres instalar. Hay una familia de ataques que no compromete a ningún mantenedor ni roba ninguna credencial: consigue que instales, voluntariamente y con el lockfile perfectamente verificado, un paquete que nunca quisiste. Se han refinado mucho y merecen su propio apartado.

Confusión de dependencias: el registro público gana la puja

En 2021 Alex Birsan publicó paquetes en npm y PyPI con los nombres de módulos internos de Apple, Microsoft, PayPal y otros treinta más. No tuvo que adivinar contraseñas: los instaladores, al ver un paquete disponible en dos índices, elegían la versión más alta, y él publicó la 9.9.9. Consiguió ejecución de código dentro de las redes corporativas de media docena de gigantes tecnológicos y cobró más de 130.000 dólares en programas de *bug bounty*.

El fallo de diseño sigue vivo en `pip`. La bandera `--extra-index-url` no significa *busca aquí primero*: significa *busca en los dos y quédate con la versión más alta*. Si tu paquete interno `empresa-auth` va por la

2.3.1 y alguien publica `empresa-auth` 99.0.0 en PyPI, tu build instalará el de PyPI y no verás ni un aviso.

```
# Peligroso: pip consulta ambos índices y coge la versión mayor
pip install --extra-index-url https://pypi.empresa.com/simple -r requirements.txt

# Seguro: un único índice, que es un proxy que hace de espejo de PyPI y sirve lo interno
pip install --index-url https://pypi.empresa.com/simple -r requirements.txt
```

`uv` corrigió el comportamiento por defecto: su estrategia es *first-index*, es decir, se detiene en el primer índice que ofrece el paquete y no mira los demás. Puedes además marcar un índice como explícito, de modo que sólo se use para los paquetes que lo pidan por nombre.

```
# pyproject.toml
[[tool.uv.index]]
name = "interno"
url = "https://pypi.empresa.com/simple"
explicit = true          # sólo para los paquetes que lo referencien expresamente

[tool.uv.sources]
empresa-auth = { index = "interno" }

# Y no vuelvas a la estrategia insegura ni por comodidad:
# UV_INDEX_STRATEGY=unsafe-best-match reintroduce exactamente el bug de 2021
```

En npm el equivalente es asociar el ámbito al registro interno, de forma que `@empresa/*` nunca se resuelva contra el registro público:

```
# .npmrc, versionado en el repositorio
@empresa:registry=https://npm.empresa.com/
//npm.empresa.com/:_authToken=${NPM_INTERNAL_TOKEN}
ignore-scripts=true
```

Y una medida barata que sigue sin aplicarse lo suficiente: registra defensivamente en el registro público los nombres y ámbitos internos, aunque los paquetes estén vacíos. Cuesta diez minutos y cierra la puerta para siempre.

Typosquatting y slopsquatting: cuando el nombre lo escribe un modelo

El typosquatting clásico —`reqeusts` en lugar de `requests`— lleva años y se combate razonablemente bien con la detección de nombres cercanos que hacen hoy PyPI y npm. Lo nuevo es más incómodo. Los asistentes de código inventan nombres de paquetes que no existen, y lo hacen de forma consistente: los mismos nombres, una y otra vez, ante prompts parecidos. Un atacante sólo tiene que registrarlos y esperar.

El caso que lo hizo evidente: en 2024, Bar Lanyado observó que varios modelos recomendaban `pip install huggingface-cli`, cuando la herramienta real se instala como `pip install -U "huggingface_hub[cli]"`. Publicó un paquete vacío con ese nombre para medir el efecto. Recibió más de 30.000 descargas auténticas en tres meses. Un estudio de 2026 sobre casi 200.000 respuestas de generación de código midió tasas de alucinación de nombres de paquete de entre el 4,62 % y el 6,10 % en cinco modelos frontera, y encontró que 53 de esos nombres —41 en PyPI y 12 en npm— seguían libres

para registrar.

La defensa no es dejar de usar asistentes: es no dejar que un nombre de paquete entre en el lockfile sin que alguien lo haya mirado. Un control de pull request que resuma los datos que importan convierte una decisión invisible en una decisión consciente.

```
# scripts/nuevas-dependencias.sh — ejecútalo en CI sobre el diff de la PR
#!/usr/bin/env bash
set -euo pipefail

base="${1:-origin/main}"

# Nombres que aparecen en package.json en esta rama y no en la base
comm -13 \
  <(git show "$base:package.json" | jq -r '.dependencies + .devDependencies | keys[]' | sort) \
  <(jq -r '.dependencies + .devDependencies | keys[]' package.json | sort) \
| while read -r pkg; do
    meta=$(npm view "$pkg" --json 2>/dev/null || echo '{}')
    if [ "$meta" = "{}" ]; then
      echo "::error::$pkg no existe en el registro. ¿Nombre alucinado?"
      continue
    fi
    creado=$(echo "$meta" | jq -r '.time.created')
    repo=$(echo "$meta" | jq -r '.repository.url // "sin repositorio"')
    mant=$(echo "$meta" | jq -r '.maintainers | length')
    edad=$(( ( $(date +%s) - $(date -d "$creado" +%s) ) / 86400 ))
    echo "::notice::$pkg — creado hace $edad días, $mant mantenedor(es), $repo"
    [ "$edad" -lt 90 ] && echo "::warning::$pkg tiene menos de 90 días. Revisión manual
obligatoria."
  done
```

No bloquea nada por sí solo, y ese es el punto: pone en el hilo de la pull request la edad del paquete, cuántos mantenedores tiene y si apunta a un repositorio real. La mayoría de los paquetes maliciosos de esta familia tienen dos días de vida, un mantenedor y ningún repositorio. Ver esas tres cifras juntas basta para que el revisor pregunte.

Cómo funciona Sigstore por dentro (y por qué cambia lo que verificas)

Antes hemos usado `cosign` como si fuera magia. Vale la pena abrir la caja, porque entender las tres piezas explica por qué la verificación correcta comprueba una identidad y no una clave, y por qué eso es una mejora y no una rareza.

Fulcio es una autoridad de certificación que no emite certificados a personas, sino a identidades OIDC. Le presentas un token de tu proveedor —el token que GitHub Actions genera para tu workflow, o tu cuenta de Google— y una clave pública efímera que acabas de generar. Fulcio devuelve un certificado de vida cortísima, del orden de diez minutos, que ata esa clave a esa identidad. Firmas, y tiras la clave privada. No hay nada que rotar ni que robar, porque a los diez minutos no queda nada.

Rekor es un registro de transparencia *append-only*, del mismo tipo que Certificate Transparency. Cada

firma se anota con su certificado y su marca de tiempo. Esto resuelve el problema evidente del párrafo anterior: si el certificado caducó hace un año, ¿cómo verificas hoy? Porque Rekor atestigua que la firma existió mientras el certificado era válido. Y de paso, cualquiera puede auditar el registro: si alguien firmara algo con tu identidad, quedaría escrito en un log público que puedes vigilar.

Cosign es la herramienta que coordina las tres cosas. Y aquí está el matiz que se escapa: como cualquiera puede obtener un certificado de Fulcio para su propia identidad, *cualquiera puede firmar tu imagen*. Una firma válida no dice nada por sí sola. Lo que dice algo es *quién* firmó y *desde dónde*.

```
# Inútil: comprueba que existe una firma válida de alguien. De quien sea.
cosign verify ghcr.io/mi-org/api@sha256:abc123...

# Correcto: exige identidad y emisor concretos
cosign verify ghcr.io/mi-org/api@sha256:abc123... \
  --certificate-oidc-issuer=https://token.actions.githubusercontent.com \
  --certificate-identity-regexp='^https://github.com/mi-org/api/.github/workflows/
release.yml@refs/tags/v.*$'

# Verificar la atestación de procedencia SLSA, no sólo la firma
cosign verify-attestation ghcr.io/mi-org/api@sha256:abc123... \
  --type slsaprovenance \
  --certificate-oidc-issuer=https://token.actions.githubusercontent.com \
  --certificate-identity-regexp='^https://github.com/mi-org/' \
  | jq '.payload | @base64d | fromjson | .predicate.buildDefinition.externalParameters'
```

Fíjate en el `--certificate-identity-regexp`: no sólo ata la firma a tu organización, la ata al *workflow concreto* y a que el disparador fuera una etiqueta de versión. Con eso, una imagen firmada desde una rama de prueba de tu propio repositorio tampoco pasa la verificación. Es el mismo principio del mínimo privilegio, aplicado a la procedencia.

Para paquetes hay atajos que no requieren cosign. En npm, `npm audit signatures` verifica las firmas del registro y las atestaciones de procedencia de todo tu árbol de dependencias de una vez. Con artefactos publicados desde GitHub, `gh attestation verify` hace lo propio contra la API de atestaciones:

```
# Verifica firmas del registro y procedencia de todo el árbol instalado
npm audit signatures

# Verifica un binario descargado contra la atestación publicada por el repositorio
gh attestation verify ./mi-binario --repo mi-org/mi-proyecto

# En un entorno sin red hacia GitHub, con el bundle descargado previamente
gh attestation verify ./mi-binario --bundle bundle.jsonl --repo mi-org/mi-proyecto
```

El SBOM que sí se usa: CycloneDX, VEX y la prueba de los cinco minutos

Ya hemos dicho que un SBOM que nadie consume es un fichero JSON caro. La prueba para saber si el tuyo sirve es concreta: **sale un CVE crítico en una librería de compresión a las cuatro de la tarde. ¿Cuánto tardas en decir qué servicios en producción la incluyen, en qué versión y en qué imagen exacta?** Si la

respuesta es *preguntamos por Slack y alguien greps*, el SBOM no está haciendo su trabajo.

Generar: qué formato y en qué momento

CycloneDX y SPDX conviven y ambos valen. CycloneDX tiende a ser más cómodo para el caso de uso de seguridad —tiene VEX integrado y un modelo de *purl* muy directo— y SPDX pesa más en cumplimiento y licencias. Elige uno y sé consistente; convertir entre ambos es posible pero se pierden matices.

El momento importa más que el formato. Un SBOM generado desde el lockfile te dice qué pediste; uno generado desde la imagen construida te dice qué hay realmente dentro, incluidos los paquetes del sistema operativo que nadie declaró. Genera el segundo, o los dos.

```
# Desde la imagen final: incluye paquetes de SO, no sólo dependencias de aplicación
syft ghcr.io/mi-org/api@sha256:abc123... -o cyclonedx-json=sbom.cdx.json

# Desde el árbol de dependencias del proyecto (más rico en metadatos de licencia)
cdxgen -t js -o sbom.cdx.json .

# Adjúntalo como atestación firmada sobre el digest, no como fichero suelto
cosign attest --predicate sbom.cdx.json --type cyclonedx \
  ghcr.io/mi-org/api@sha256:abc123...

# Escaneo del SBOM, no de la imagen: más rápido y funciona sin acceso al registro
grype sbom:./sbom.cdx.json --fail-on high
```

Ese último comando es el que convierte el SBOM en algo operativo: escanear el documento en lugar de la imagen significa que puedes responder a un CVE nuevo sin reconstruir ni volver a descargar nada, y que puedes hacerlo sobre artefactos que ya no están en ningún registro.

Consumir: un inventario, no una carpeta de artefactos

El paso que casi todos se saltan es publicar el SBOM en un sitio consultable. Dependency-Track es la opción de referencia en open source: le envías cada SBOM en cada build, mantiene el inventario por proyecto y versión, y cuando aparece una vulnerabilidad nueva te avisa de qué proyectos están afectados sin que tú preguntes.

```
# .github/workflows/release.yml — fragmento
      - name: Generar SBOM
        run: syft "$IMAGE_DIGEST" -o cyclonedx-json=sbom.cdx.json

      - name: Publicar en el inventario
        env:
          DT_URL: ${{ vars.DEPENDENCY_TRACK_URL }}
          DT_KEY: ${{ secrets.DEPENDENCY_TRACK_API_KEY }}
        run: |
          curl -sSf -X POST "$DT_URL/api/v1/bom" \
            -H "X-Api-Key: $DT_KEY" \
            -F "projectName=api" \
            -F "projectVersion=${GITHUB_REF_NAME}" \
            -F "autoCreate=true" \
            -F "bom=@sbom.cdx.json"
```

Con eso, la pregunta de las cuatro de la tarde se responde con una consulta por *purl*. Si no quieres desplegar Dependency-Track, el mínimo viable es guardar los SBOM en un bucket con una convención de rutas por servicio y digest, y tener un script que haga `jq` sobre todos ellos. Es feo, pero contesta en cinco minutos, que es el requisito real.

VEX: la mitad de los CVE de tu informe no te afectan

Cualquiera que haya pasado un escáner por una imagen de producción conoce la sensación: doscientas vulnerabilidades, de las que la gran mayoría están en código que tu aplicación nunca llama. El problema no es que el escáner mienta; es que sin más contexto no puede distinguir *presente* de *explotable*. VEX (*Vulnerability Exploitability eXchange*) es el documento donde tú aportas ese contexto, de forma legible por máquina y auditable.

Los cuatro estados son `not_affected`, `affected`, `fixed` y `under_investigation`. El valioso es el primero, porque debe ir acompañado de una justificación de un vocabulario cerrado: el componente no está presente en el ejecutable, el código vulnerable no es alcanzable, no puede ser controlado por un atacante, o existe una mitigación.

```
{
  "bomFormat": "CycloneDX",
  "specVersion": "1.6",
  "vulnerabilities": [
    {
      "id": "CVE-2026-11111",
      "source": { "name": "NVD" },
      "affects": [ { "ref": "pkg:npm/libcompresion@3.2.1" } ],
      "analysis": {
        "state": "not_affected",
        "justification": "code_not_reachable",
        "detail": "El fallo está en el decodificador de streams. La API sólo expone compresión
sincrona sobre buffers en memoria; el decodificador no se enlaza en el bundle de producción
(verificado con analisis de alcanzabilidad el 2026-08-14).",
        "response": [ "will_not_fix" ]
      }
    }
  ]
}
```

El detalle que hace que esto funcione a largo plazo: la justificación se escribe una vez y se reutiliza en cada escaneo posterior. Sin VEX, tu equipo vuelve a investigar el mismo CVE cada trimestre, llega a la misma conclusión y no la escribe en ningún sitio. Con VEX, el informe de mañana ya viene con esos doscientos filtrados a los ocho que importan, y cada exclusión tiene nombre, fecha y razón.

Te ha tocado: runbook de las primeras 24 horas

Todo lo anterior reduce la probabilidad. No la lleva a cero. La diferencia entre un susto y un incidente serio suele estar en la primera hora, y esa hora se juega mucho mejor con un guion escrito de antemano que improvisando a las once de la noche.

Hora 0 — Contener

Antes de investigar nada, corta la propagación. La lógica es sencilla: cada instalación nueva puede ejecutar el payload en una máquina más, y cada máquina infectada busca credenciales para publicar más paquetes.

```
# 1. Apaga los scripts de ciclo de vida en toda la organización, ya
npm config set ignore-scripts true --location=global

# 2. Congela CI: desactiva los workflows que instalan dependencias
gh workflow list --repo mi-org/api --json name,id \
  | jq -r '.[].id' \
  | xargs -I{} gh workflow disable {} --repo mi-org/api

# 3. Bloquea la publicación desde la organización mientras dure el triaje
#    (en npm: revoca los tokens de publicación; con Trusted Publishing,
#     desactiva el trusted publisher del paquete en la web del registro)
```

Si usas Trusted Publishing, este paso es notablemente más limpio: no hay tokens repartidos por cincuenta repositorios que rastrear, hay una configuración por paquete que desactivas en un minuto. Es una de esas ventajas que no aparecen en la comparativa hasta que te hace falta.

Hora 1 — Determinar el alcance

La pregunta es qué versiones exactas entraron y dónde. No basta con mirar el `package.json`: lo que importa es qué hay resuelto en los lockfiles, incluyendo dependencias transitivas que nadie declaró nunca.

```
# scripts/buscar-comprometidos.sh
#!/usr/bin/env bash
# Uso: ./buscar-comprometidos.sh comprometidos.txt  (una linea por "nombre@version")
set -euo pipefail
lista="$1"

find . -name 'package-lock.json' -o -name 'pnpm-lock.yaml' -o -name 'yarn.lock' \
  -o -name 'uv.lock' -o -name 'poetry.lock' | while read -r lock; do
  while read -r entrada; do
    nombre="${entrada%@*}"; version="${entrada##*@}"
    if grep -qF "$nombre" "$lock" && grep -qF "$version" "$lock"; then
      echo "COINCIDENCIA  $lock  ->  $entrada"
    fi
  done < "$lista"
done

# Y en un repositorio concreto, la ruta exacta que la introdujo:
#   npm ls paquete-comprometido
# que te dice qué dependencia directa tuya arrastra la maliciosa.
```

En paralelo, construye la línea temporal: la fecha de publicación de la versión maliciosa y las fechas de tus builds. Todo build anterior a la publicación está limpio con certeza, y eso reduce enormemente el trabajo. Los logs de CI y las marcas de tiempo de las imágenes en el registro te dan esa lista sin esfuerzo.

Horas 2 a 6 — Rotar, en el orden correcto

El error clásico es rotar credenciales desde una máquina que sigue comprometida, o rotarlas en un orden que permite al atacante usar una para renovar otra. El orden que funciona va de lo que concede más poder a lo que concede menos:

1. **Credenciales de identidad**: tokens de GitHub (PAT, OAuth de apps, claves de despliegue), porque con ellas se obtiene todo lo demás. Revoca primero, reemite después.
2. **Credenciales de nube de larga vida**: claves de acceso IAM, cuentas de servicio con clave JSON. Si alguna es innecesaria porque ya podrías usar OIDC, este es el momento de no reemitirla.
3. **Tokens de registro de paquetes** y credenciales de firma que no sean efímeras.
4. **Secretos de aplicación**: claves de API de terceros, cadenas de conexión, claves de firma de sesión y de webhooks.

Y una comprobación que se olvida siempre: los *forks* y los repositorios personales de la gente del equipo. Los gusanos de esta familia usan las credenciales robadas para crear repositorios nuevos —a menudo con un nombre fijo y reconocible— y para registrar *runners* autoalojados que sobreviven a la rotación de secretos. Busca ambas cosas explícitamente.

```
# Repositorios creados en la ventana del incidente en toda la organización
gh api "orgs/mi-org/repos?sort=created&direction=desc&per_page=50" \
  --jq '.[] | select(.created_at > "2026-08-10") | [.name, .created_at, .private] | @tsv'

# Runners autoalojados registrados: persistencia que sobrevive a rotar secretos
gh api orgs/mi-org/actions/runners --jq '.runners[] | [.name, .status, .os] | @tsv'

# Workflows nuevos o modificados en repositorios que no deberian tenerlos
gh api "search/commits?q=org:mi-org+path:.github/workflows+committer-date:>2026-08-10" \
  --jq '.items[] | [.repository.full_name, .commit.author.date, .commit.message] | @tsv'
```

Horas 6 a 24 — Reconstruir y cerrar

Reconstruye desde entornos limpios, no desde las máquinas afectadas: un `npm ci` en un portátil comprometido no te devuelve a un estado bueno. Regenera los lockfiles fijando versiones seguras conocidas, reconstruye las imágenes, y verifica las nuevas atestaciones antes de desplegar —es justo el momento en el que toda la maquinaria de procedencia demuestra para qué servía—.

El postmortem tiene una pregunta que importa más que las demás, y no es *cómo entró*. Es *¿cuánto tardamos en saber que había entrado, y por qué tanto?*. La detección casi siempre llega desde fuera —un aviso del registro, un boletín, un cliente— y ese retraso es la variable en la que de verdad puedes influir con lo que hemos visto: enfriamiento, inventario consultable y verificación en admisión.

Gobernar esto en cincuenta repositorios: Scorecard, niveles SLSA y política como código

Aplicar todo lo anterior en un repositorio es una tarde. Aplicarlo en una organización entera y que siga

aplicado dentro de seis meses es un problema distinto, y se resuelve con tres herramientas: una métrica, un objetivo y un mecanismo que impida la regresión.

La métrica: OpenSSF Scorecard

Scorecard es un análisis automático que puntúa un repositorio de 0 a 10 sobre una veintena de comprobaciones, varias de las cuales son exactamente las que hemos ido viendo: si las acciones están fijadas por SHA, si los permisos de los tokens están acotados, si hay revisión de código obligatoria, si hay ramas protegidas, si existen dependencias con vulnerabilidades conocidas. Lo interesante es que también puedes ejecutarlo sobre tus *dependencias*, no sólo sobre tu código.

```
# .github/workflows/scorecard.yml
name: Scorecard
on:
  branch_protection_rule:
  schedule: [{ cron: '30 4 * * 1' }]
  push: { branches: [main] }

permissions: read-all

jobs:
  analysis:
    runs-on: ubuntu-latest
    permissions:
      security-events: write   # para subir el SARIF a code scanning
      id-token: write          # para publicar el resultado firmado
      contents: read
    steps:
      - uses: actions/checkout@11bd71901bbe5b1630ceea73d27597364c9af683 # v4.2.2
        with: { persist-credentials: false }
      - uses: ossf/scorecard-action@62b2cac7ed8198b15735ed49ab1e5cf35480ba46 # v2.4.0
        with:
          results_file: results.sarif
          results_format: sarif
          publish_results: true
      - uses: github/codeql-action/upload-sarif@v3
        with: { sarif_file: results.sarif }
```

Un consejo sobre cómo usar la puntuación: no persigas el 10. Persigue que ninguna comprobación concreta esté en 0 y que la tendencia suba. Un panel con la puntuación de los cincuenta repositorios, ordenado ascendente, es una herramienta de priorización mucho mejor que cualquier reunión.

El objetivo: qué compra cada nivel de SLSA

SLSA se explica mal casi siempre porque se presenta como una escalera de cumplimiento. Vista como una escalera de *propiedades* es mucho más útil, porque cada nivel responde a un ataque distinto:

- **Build L1 — hay procedencia.** Sabes qué build produjo este binario y con qué entradas. No impide nada por sí solo, pero convierte una investigación de dos días en una consulta. Cuesta las cinco líneas de `attest-build-provenance`.

- **Build L2 — la procedencia está firmada por el servicio de build.** Ya no basta con que alguien escriba un JSON plausible: la atestación viene de la plataforma y es verificable. Bloquea la falsificación por parte de un tercero.
- **Build L3 — la procedencia es infalsificable incluso para quien controla el repositorio.** El build corre en un entorno aislado que el propio proceso de build no puede manipular. En GitHub se consigue firmando desde un *reusable workflow* cuyo contenido el proyecto no puede alterar. Esto es lo que bloquea al atacante que ya tiene permiso de escritura: puede meter código malicioso, pero no puede mentir sobre de dónde salió el binario.

Para la mayoría de equipos, L2 en todo y L3 en los artefactos que otros consumen es un objetivo honesto. L3 en todos los repositorios es un proyecto de plataforma, no una tarea.

El mecanismo: la plantilla que no se puede editar

La forma que funciona de que cincuenta repositorios hagan lo mismo no es documentación: es un *reusable workflow* que ya trae resueltos los permisos, el pinning, la publicación OIDC y la atestación, más una regla de organización que exija su uso.

```
# .github/workflows/release.yml en cada repositorio consumidor: esto es todo
name: Release
on:
  push:
    tags: ['v*']

permissions: {}          # nada por defecto; el workflow reutilizable pide lo suyo

jobs:
  publicar:
    uses: mi-org/plataforma/.github/workflows/publicar-npm.yml@v3
    permissions:
      contents: read
      id-token: write      # OIDC para Trusted Publishing y para firmar
      attestations: write
    with:
      node-version: '22'
    secrets: inherit
```

El repositorio consumidor no puede equivocarse en los permisos ni olvidarse de la atestación, porque no los escribe. Y cuando salga la siguiente buena práctica, la despliegas subiendo una etiqueta en un único sitio. Combínalo con los *rulesets* de organización para exigir que ese workflow se ejecute antes de poder publicar, y tienes un control que sobrevive a la rotación del equipo.

Qué medir para saber si esto está vivo

Cuatro números en un panel, revisados una vez al mes, detectan la erosión antes de que sea un problema:

1. **Porcentaje de repositorios que publican con OIDC** frente a los que aún tienen un token en secretos. El objetivo es cero tokens.
2. **Porcentaje de acciones de terceros fijadas por SHA** sobre el total de usos. Un solo `@v4` nuevo baja el

número y salta a la vista.

3. **Edad mediana de las dependencias en el momento de instalarse.** Si cae por debajo de tu ventana de enfriamiento, alguien la ha desactivado en algún sitio.
4. **Cobertura de la política de admisión**: qué proporción de los despliegues pasa por verificación de firma en modo *Enforce*, no *Audit*. Es habitual quedarse años en Audit por inercia.

Ninguno es un indicador de seguridad en sentido estricto. Todos son indicadores de si los controles siguen enchufados, que en la práctica es la pregunta que se falla.

Los errores que anulan todo lo anterior

He visto los ocho repetirse en equipos serios. La mayoría no rompen nada visiblemente, que es justo lo que los hace peligrosos.

- **Verificar la firma sin verificar la identidad.** Ya lo hemos dicho, pero es el fallo número uno: `cosign verify` sin `--certificate-identity` ni `--certificate-oidc-issuer` acepta cualquier firma de cualquiera.
- **Firmar la etiqueta en lugar del digest.** La firma sigue siendo válida mientras la etiqueta apunta a otra imagen.
- `npm install` **en CI.** Convierte el lockfile en una sugerencia.
- `ignore-scripts` **sólo en CI.** El portátil del desarrollador tiene credenciales de nube, tokens de npm y claves SSH. Es un objetivo mejor que tu runner. Ponlo también en la configuración de usuario.
- `id-token: write` **a nivel de workflow.** Concede a los jobs de test la capacidad de obtener credenciales de publicación.
- **Enfriamiento en el instalador pero no en el bot.** Acabas con una cola de pull requests que fallan al instalar y el equipo termina desactivando el gate por frustración.
- **SBOM que se genera y nadie consume.** Un fichero JSON en los artefactos del build no es un control. Si no lo verificas en admisión ni lo cruzas con avisos de vulnerabilidades, es papeleo.
- **Un token de publicación con alcance de organización.** Si un único paquete se compromete, se llevan por delante todos los demás. Alcance mínimo, o mejor, ningún token.

Si sólo tienes una tarde

Orden de implantación por impacto real, no por elegancia:

1. Añade el enfriamiento (`min-release-age` o equivalente) y configúralo también en Dependabot o Renovate. Media hora, y te habría protegido de la mayoría de incidentes de los últimos dos años.
2. Pon `ignore-scripts=true`, ejecuta `npm rebuild` con la lista corta de lo que de verdad lo necesita, y añade el script de auditoría al pipeline.
3. Cambia `npm install` por `npm ci` en todos los workflows y declara `permissions: contents: read` arriba del todo.
4. Ejecuta `pinact run` y activa el ecosistema `github-actions` en Dependabot.

5. Migra la publicación a Trusted Publishing y borra el token del repositorio. Borrarlo es parte del trabajo: un token configurado y sin usar sigue siendo un token válido.
6. Añade `attest-build-provenance` al job de release. Son cinco líneas.
7. Despliega la política de admisión en modo Audit, y pásala a Enforce cuando el ruido llegue a cero.

Checklist de producción

- Todas las instalaciones en CI usan el lockfile de forma estricta (`npm ci`, `--frozen-lockfile`, `--immutable`, `uv sync --locked`).
- Los scripts de ciclo de vida están desactivados por defecto, con una lista corta y revisada de excepciones.
- Hay un periodo de enfriamiento configurado en el instalador y en el bot de dependencias, con una vía documentada para saltárselo ante un CVE.
- Todas las acciones de terceros están fijadas por SHA completo, y CI falla si alguien introduce una etiqueta.
- Los permisos por defecto del workflow son `contents: read`; `id-token` y `attestations` viven sólo en el job de release.
- Ningún workflow ejecuta código de una pull request externa en un contexto con secretos.
- La publicación usa OIDC. No queda ningún token de registro de larga vida en los secretos.
- Cada artefacto y cada imagen llevan atestación de procedencia y SBOM firmados sobre su digest.
- El clúster rechaza en admisión las imágenes sin firma válida, comprobando identidad y emisor, con `mutateDigest` activado.
- El runbook del incidente responde en menos de cinco minutos a: qué commit produjo este binario y qué dependencias contiene.

Preguntas frecuentes

¿No basta con un escáner de vulnerabilidades? No. Un escáner compara tu inventario con una base de datos de vulnerabilidades *conocidas*. Un paquete comprometido hace dos horas no está en ninguna base de datos. El enfriamiento y el bloqueo de scripts atacan la ventana en la que el escáner todavía no sabe nada.

¿Un lockfile no es suficiente para reproducibilidad? El lockfile fija *qué* versiones instalas, no *qué hacen* al instalarse ni quién las construyó. Son problemas ortogonales: por eso hacen falta las cinco capas.

El enfriamiento retrasa parches de seguridad. ¿No es contraproducente? Es el compromiso central, y por eso todas las implementaciones traen una vía de escape por invocación. En la práctica, el riesgo de consumir una versión maliciosa recién publicada es mucho mayor que el de aplicar un parche legítimo veinticuatro horas más tarde. Si un CVE crítico te afecta de verdad, te saltas el gate conscientemente.

¿Sirve todo esto si mi código es privado? Sí, y en algunos aspectos más. Tus dependencias siguen siendo públicas y tu CI sigue teniendo credenciales de producción. Además, un repositorio privado con secretos de nube es un objetivo más rentable que una librería open source.

¿Qué gano exactamente al llegar a SLSA Build Level 3? Que la procedencia sea inforjable incluso para alguien con permisos de escritura en el repositorio, porque el build corre en un entorno aislado que el proceso de build no puede manipular. En GitHub se consigue firmando desde un workflow reutilizable en el que el código del proyecto no puede influir sobre la generación de la atestación.

¿Por dónde empiezo si mi organización tiene cincuenta repositorios? Por una plantilla. Un workflow reutilizable con los permisos, el pinning y la publicación OIDC ya resueltos, más un fichero `.npmrc` distribuido con la configuración base. Migra primero los repositorios que publican paquetes o imágenes; los que sólo consumen pueden esperar a la segunda tanda.

Más preguntas frecuentes

¿Merece la pena el enfriamiento si mi equipo actualiza dependencias una vez al trimestre? Más todavía, y por una razón que no es obvia: la actualización trimestral trae de golpe cientos de versiones nuevas, y algunas tendrán días de vida en el momento de instalarse. El enfriamiento no encarece nada en ese modelo —de hecho no lo notarás— y filtra exactamente el escenario de riesgo.

Uso runners autoalojados. ¿Cambia algo? Cambia bastante, y a peor si no lo tienes en cuenta. Un runner autoalojado persistente comparte estado entre jobs: caché de paquetes, credenciales de Docker, directorio home. Un payload que corre en el build de una pull request puede quedarse esperando al siguiente job. Si los usas, que sean efímeros —una máquina limpia por job— y nunca los expongas a pull requests de forks.

Mi organización usa un proxy o espejo interno de npm y PyPI. ¿Eso me protege? Te protege de la indisponibilidad y de la confusión de dependencias si está bien configurado, pero no de un paquete legítimamente publicado y malicioso: el espejo replicará la versión comprometida igual de fiel. Donde sí ayuda mucho es como punto único donde aplicar el enfriamiento y una lista de bloqueo, para toda la organización a la vez. Es la mejor razón para tener uno.

¿Y las dependencias de mis dependencias de desarrollo? Son cientos. Son el vector más habitual, precisamente porque nadie las mira y porque corren con los mismos permisos que el resto. La respuesta práctica no es auditarlas una a una, sino que los controles no distingan: `ignore-scripts`, enfriamiento y lockfile estricto se aplican al árbol completo sin que tengas que decidir nada paquete a paquete.

¿Puedo verificar firmas si mis nodos de producción no tienen salida a internet? Sí. Tanto `cosign` como `gh attestation verify` aceptan un *bundle* con el certificado y la prueba de inclusión en Rekor, de forma que la verificación es completamente offline. Descarga el bundle en el paso de build, que sí tiene red, y publícalo junto al artefacto.

Mi equipo tiene tres personas y ninguna es de seguridad. ¿Qué hago? Los tres primeros puntos de la lista de la tarde: enfriamiento, `ignore-scripts` y `npm ci`. Son configuración, no proyectos; no requieren mantenimiento y cubren la forma en que han entrado la mayoría de los incidentes reales de los últimos dos años. Lo demás puede esperar a que haya alguien que lo mantenga.

Cierre

La lección de los dos últimos años no es que necesitemos herramientas más sofisticadas. Es que las

defensas más eficaces son sorprendentemente aburridas: un retraso de veinticuatro horas, un interruptor que apaga los scripts, cuarenta caracteres en lugar de dos, y un permiso movido de un sitio a otro. Ninguna de esas líneas aparecerá en una demo impresionante. Todas ellas habrían evitado que un correo de phishing a un mantenedor terminara en tus máquinas.

PuigFlows

Software Supply Chain Security: Lockfiles, Release Cooldowns, OIDC Trusted Publishing and Sigstore Signatures

A practical guide to closing the five doors a supply chain attack walks through: strict lockfile installs, blocking the postinstall scripts that propagated the Shai-Hulud worm, release cooldowns (min-release-age, minimumReleaseAge, npmMinimalAgeGate) coordinated with Dependabot and Renovate, pinning GitHub Actions to f…

Guide · Intermediate · 40 min read
Updated 2026-08-17 · puigflows.com

A worm, 700 packages, and an uncomfortable lesson

In September 2025, a phishing email impersonating an npm security alert harvested a maintainer's credentials. There was no buffer overflow, no sophisticated exploit chain: someone typed their password into a fake form. From there, the *Shai-Hulud* worm did the rest. Its first wave compromised hundreds of packages; the second, in November of that same year, pushed past 700 affected packages and generated more than 27,000 public repositories stuffed with secrets stolen from the machines of anyone who installed a poisoned version.

The propagation mechanism wasn't exotic either. It was a `postinstall` script — the kind your package manager runs, without asking you anything, the first time you install a dependency. That script scanned the environment for credentials, exfiltrated them, and used them to publish malicious versions of the maintainer's other packages. Self-replication, with no human involvement after the first click.

What stings about this incident is that nearly all the damage was preventable with configuration. Not with an expensive tool or a dedicated security team: with five or six lines spread across an `.npmrc`, a workflow file, and an admission policy. This guide walks through exactly those lines.

The threat model: five doors, not one

When people say "supply chain security" they're usually picturing one thing: a malicious package on npm or PyPI. That's the most famous door, but it isn't the only one, and guarding it alone leaves the rest wide open. It's worth enumerating them, because each has its own cheap defense.

1. **Installation.** You download third-party code and your package manager executes it. Vector: typosquatting, compromised maintainer accounts, lifecycle scripts.
2. **The build.** Your CI runs third-party actions and containers with access to your secrets. Vector: an action tagged `v3` that one day points at a different commit.
3. **Publishing.** A long-lived token sitting in a repository secret can publish under your name forever. Vector: token leak, maintainer phishing.
4. **Distribution.** Between building an artifact and someone deploying it, who guarantees it's the same binary? Vector: compromised registry, mutable tag repointed.
5. **Deployment.** Your cluster will start whatever image it's handed. Vector: any of the four above that you failed to catch.

The sections that follow close each door, ordered by cost-to-benefit ratio. If you only have one afternoon, do the first two.

Door 1 — Installation: lockfiles, scripts, and cooldowns

Always install from the lockfile

The difference between `npm install` and `npm ci` looks cosmetic and isn't. `npm install` resolves

versions, may rewrite the lockfile, and carries on when `package.json` and `package-lock.json` disagree. `npm ci` wipes `node_modules`, installs exactly what the lockfile says, and fails on any mismatch. In CI, the first option means two runs of the same commit can install different dependency trees.

```
# In CI, never this:
npm install

# Always this:
npm ci

# Equivalents in other package managers:
pnpm install --frozen-lockfile
yarn install --immutable
uv sync --locked          # Python: fails if uv.lock is stale
```

A lockfile is only useful if it stores integrity hashes and nobody regenerates it casually. Treat it as code and review its diffs in pull requests. A one-line change in `package.json` that produces four hundred new lockfile lines deserves a look.

Stop executing arbitrary code at install time

`preinstall`, `install`, and `postinstall` scripts are how Shai-Hulud spread. Some packages genuinely need them (compiling native bindings, downloading a binary), but the vast majority don't. The correct posture is off by default, with explicitly approved exceptions.

```
# .npmrc at the project root
ignore-scripts=true
```

npm's practical drawback is that it has no allowlist: all or nothing. The workaround is to rebuild by hand the few packages that need it, right after installing:

```
npm ci                       # no scripts, thanks to .npmrc
npm rebuild esbuild sharp    # only what you have reviewed and approved
```

pnpm solved this more gracefully. Since pnpm 10, dependency scripts are blocked by default, and in pnpm 11 the allowlist is declared with `allowBuilds`, which replaces the old `onlyBuiltDependencies`:

```
# pnpm-workspace.yaml
allowBuilds:
  - esbuild
  - sharp
```

In Python the equivalent is avoiding source distributions, which execute `setup.py` during installation. Wheels, by contrast, are simply unpacked:

```
# Fails if a package only ships an sdist, instead of silently building it
uv pip install --only-binary=:all: -r requirements.txt
```

Release cooldowns: the best cost-to-benefit ratio available

Nearly all malicious versions are detected and pulled within hours. The attack depends on your CI consuming them immediately after publication. If your installer refuses to resolve versions published less than twenty-four hours ago, most of these incidents pass right over you.

All the Node package managers support this today, each with a different name and a different unit. Here's the map that saves you an afternoon:

- **npm** (CLI 11.10.0 or newer): `min-release-age`, in **days**.
- **pnpm** (since 10.16): `minimumReleaseAge`, in **minutes**. Enabled by default at 1440 since pnpm 11.0.
- **Yarn** (Berry 4.10+): `npmMinimalAgeGate`, in **minutes**, and as a number: duration strings like "7d" hit a known parsing bug.

```
# npm — .npmrc — unit: DAYS
min-release-age=1
```

```
# pnpm 11 — pnpm-workspace.yaml — unit: MINUTES
minimumReleaseAge: 1440
minimumReleaseAgeExclude:
  - '@my-org/*'          # internal packages that shouldn't wait
```

```
# Yarn Berry 4.10+ — .yarnrc.yml — unit: MINUTES
npmMinimalAgeGate: 1440
npmPreapprovedPackages:
  - "@my-org/*"
```

Two details almost everyone misses. First: the cooldown applies at install time, not when updates are proposed, so you have to configure it in your dependency bot too or it will open pull requests you can't actually install. Renovate exposes it as `minimumReleaseAge` and Dependabot as `cooldown`, which has shipped with a default since July 2026.

```
# .github/dependabot.yml
version: 2
updates:
  - package-ecosystem: npm
    directory: "/"
    schedule:
      interval: weekly
    cooldown:
      default-days: 7
      semver-major-days: 14
```

Second: you need an escape hatch for the urgent security patch, and there is one. npm lets you bypass the gate per invocation with `npm i --min-release-age 0 package@version`. That escape hatch is precisely what makes the gate viable in the first place.

A guardrail that fails in CI

Configuration gets forgotten and reverted. It's worth having an automated check that no new dependency slipped an install script past you. This script reads the npm lockfile (v2/v3 format), finds packages flagged

with `hasInstallScript`, and fails if any of them isn't on your approved list:

```
// scripts/audit-install-scripts.mjs
import { readFile } from 'node:fs/promises';

// Hand-reviewed list. Adding something here requires justifying it in the PR.
const APPROVED = new Set(['esbuild', 'sharp', '@swc/core']);

const lock = JSON.parse(await readFile('package-lock.json', 'utf8'));
const offenders = [];

for (const [path, meta] of Object.entries(lock.packages ?? {})) {
  if (!meta.hasInstallScript) continue;
  const name = path.split('node_modules/').pop();
  if (APPROVED.has(name)) continue;
  offenders.push(name + '@' + (meta.version ?? '?'));
}

if (offenders.length > 0) {
  console.error('Dependencies with UNAPPROVED install scripts:');
  for (const o of offenders) console.error('  - ' + o);
  console.error('Review what they do before adding them to APPROVED.');
  process.exit(1);
}

console.log('OK: no dependency runs install scripts without approval.');
```

It hooks into the pipeline as one more step, next to the linter. Five minutes of work, and it catches exactly the kind of change nobody looks at in a four-hundred-line diff.

Door 2 — The build: pin actions by SHA and trim permissions

A tag is not a version

When you write `uses: someone/action@v3` you're trusting a Git tag. Tags are movable pointers: the repository owner can repoint `v3` at any commit at any time, and your workflow will execute that new code without you changing a line. If that account is compromised, the attacker inherits your secrets on the next run. This has happened, to very popular actions.

The only immutable reference in Git is the full commit SHA. Pinning costs you forty ugly characters, so it's conventionally paired with a comment carrying the human-readable version:

```
# Mutable: tomorrow v5 may point at a different commit
- uses: actions/checkout@v5

# Immutable: this exact commit and no other
- uses: actions/checkout@FULL_40_CHARACTER_SHA # v5.0.0
```

Nobody sustains this by hand for long, and nobody has to. The `pinact` tool rewrites every tag in your workflows to real SHAs with the correct comment, and Dependabot understands pinned SHAs: it opens

pull requests that bump the SHA and the comment together. The combination gives you immutability without freezing on old versions.

```
# Pin every action in the repository at once
pinact run

# Verify in CI that nobody reintroduced a tag
pinact run --check

# And let Dependabot propose the bumps:
# .github/dependabot.yml -> package-ecosystem: "github-actions"
```

Least privilege, and only where it's needed

By default, a workflow's `GITHUB_TOKEN` may hold write permissions on the repository. A single compromised step in any job can then push commits, cut releases, or modify the workflows themselves. The rule is to declare the minimum at workflow level and widen it only in the job that needs it:

```
name: ci

on:
  pull_request:
  push:
    branches: [main]

# The whole workflow starts read-only
permissions:
  contents: read

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@FULL_40_CHARACTER_SHA # v5.0.0
      - uses: actions/setup-node@FULL_40_CHARACTER_SHA # v5.0.0
        with:
          node-version: 22
          cache: npm
      - run: npm ci
      - run: node scripts/audit-install-scripts.mjs
      - run: npm test

  release:
    if: github.ref == 'refs/heads/main'
    needs: test
    runs-on: ubuntu-latest
    permissions:
      contents: read
      id-token: write        # only this job can request an OIDC token
      attestations: write    # only this job can sign artifacts
    steps:
      - uses: actions/checkout@FULL_40_CHARACTER_SHA # v5.0.0
      # ... build and publish
```

That `id-token: write` is the key to door 3. Putting it at workflow level, where tests also run, means any test dependency can request a federated identity token. Put it only on the release job.

The trigger that hands over your repository

This one deserves its own warning. `pull_request_target` runs the workflow with the base repository's secrets, and it's commonly combined with a checkout of the pull request branch. That amounts to executing a stranger's code with your credentials loaded into the environment.

```
# DANGER: untrusted code + repository secrets
on: pull_request_target
jobs:
  build:
    steps:
      - uses: actions/checkout@FULL_40_CHARACTER_SHA
        with:
          ref: ${{ github.event.pull_request.head.sha }}   # attacker's code
      - run: npm ci && npm run build                       # with your secrets
```

If you need pull request data, build with a plain `pull_request` trigger (which receives no secrets) and do the privileged work in a separate `workflow_run` workflow, never executing the PR's code in the privileged context.

Door 3 — Publishing: goodbye to long-lived tokens

The classic pattern is to generate an API token on npm or PyPI, paste it into a repository secret, and forget about it. That token never expires. If it leaks — in a log, in a screenshot, in a phishing email — the attacker can publish versions of your package until someone notices and manually revokes it. Exactly the Shai-Hulud scenario.

Trusted Publishing removes the token entirely. Instead, the registry stores a configuration that says, literally: "I accept publishes for this package if they come from the `release.yml` workflow in the `my-org/my-package` repository, in the `pypi` environment." GitHub mints a very short-lived OIDC token proving that origin, and the registry exchanges it for an ephemeral publishing credential. There's nothing to leak because there's nothing stored. It's the OpenSSF trusted publishers standard, implemented today by PyPI, npm, RubyGems, and others.

PyPI

First you configure the trusted publisher on the PyPI website: owner, repository, workflow filename, and environment. After that, the workflow contains no secrets at all:

```
# .github/workflows/release.yml
name: release

on:
  release:
    types: [published]
```

```yaml
permissions:
  contents: read

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@FULL_40_CHARACTER_SHA # v5.0.0
      - uses: astral-sh/setup-uv@FULL_40_CHARACTER_SHA # v6
      - run: uv build
      - uses: actions/upload-artifact@FULL_40_CHARACTER_SHA # v4
        with:
          name: dist
          path: dist/

  publish:
    needs: build
    runs-on: ubuntu-latest
    environment: pypi          # lets you require manual approval before publishing
    permissions:
      id-token: write          # the ONLY permission needed: there is no PyPI token
    steps:
      - uses: actions/download-artifact@FULL_40_CHARACTER_SHA # v4
        with:
          name: dist
          path: dist/
      - uses: pypa/gh-action-pypi-publish@FULL_40_CHARACTER_SHA # v1.13.0
```

Splitting `build` from `publish` isn't decoration: the job holding `id-token: write` runs none of the project's code, it only moves already-built files. And `environment: pypi` lets you require human approval in GitHub before that job starts.

npm

npm added Trusted Publishing in 2025 following the same standard. You configure the trusted publisher on the package page and the workflow becomes:

```yaml
  publish:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      id-token: write
    steps:
      - uses: actions/checkout@FULL_40_CHARACTER_SHA # v5.0.0
      - uses: actions/setup-node@FULL_40_CHARACTER_SHA # v5.0.0
        with:
          node-version: 22
          registry-url: https://registry.npmjs.org
      - run: npm ci
      - run: npm publish        # no NODE_AUTH_TOKEN: identity comes from OIDC
```

As a side effect, npm automatically attaches package provenance, which is what the next section is about.

If you still depend on a classic token, at least use `npm publish --provenance` and scope the token to a single package.

Door 4 — Provenance: attestations and SBOMs

So far you've reduced the odds of something bad getting in. Provenance answers a different question, the one that surfaces during an incident: *this binary running in production — which commit did it come from, which workflow built it, and what's inside it?* Without a signed answer, the investigation turns into archaeology.

Build provenance attestations with GitHub Actions

An artifact attestation is a signed statement, in in-toto format, binding an artifact's digest to the workflow that produced it: repository, commit, workflow file, triggering event, and OIDC token claims. It's signed keylessly: Sigstore issues a short-lived certificate tied to the workflow's identity and records the signature in a public transparency log. This alone gets you to SLSA v1.0 Build Level 2, and with a properly isolated reusable workflow, to Level 3.

```
  release:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      id-token: write
      attestations: write
      packages: write
    steps:
      - uses: actions/checkout@FULL_40_CHARACTER_SHA # v5.0.0
      - run: npm ci && npm run build

      # 1) Provenance for the artifact
      - uses: actions/attest-build-provenance@FULL_40_CHARACTER_SHA # v3
        with:
          subject-path: 'dist/**/*.tgz'

      # 2) SBOM: an inventory of everything inside
      - name: Generate SBOM
        run: |
          curl -sSfL https://raw.githubusercontent.com/anchore/syft/main/install.sh | sh -s -- -
b /usr/local/bin
          syft scan dir:. -o spdx-json=sbom.spdx.json

      - uses: actions/attest-sbom@FULL_40_CHARACTER_SHA # v3
        with:
          subject-path: 'dist/**/*.tgz'
          sbom-path: 'sbom.spdx.json'
```

A nuance that often escapes people: an SBOM generated *after* installing dependencies describes what's actually in the tree, transitives included. An SBOM generated from `package.json` describes what you thought you had. The gap between the two is exactly where the surprises live.

Container images: sign the digest, never the tag

Image tags are mutable. `:1.4.2` may point somewhere else tomorrow; `@sha256:...` won't. Signing the tag gives you a guarantee that evaporates the moment somebody pushes again. With cosign in keyless mode:

```
# Push and capture the exact digest (not the tag)
DIGEST=$(docker buildx build --push -t ghcr.io/my-org/app:1.4.2 --metadata-file meta.json . && jq
-r '."containerimage.digest"' meta.json)

# Sign the digest. In Actions there are no keys to pass: it uses the runner's OIDC.
cosign sign --yes ghcr.io/my-org/app@${DIGEST}

# Attach the SBOM as a signed attestation on the same digest
syft scan ghcr.io/my-org/app@${DIGEST} -o spdx-json=sbom.spdx.json
cosign attest --yes --predicate sbom.spdx.json --type spdxjson ghcr.io/my-org/app@${DIGEST}
```

Door 5 — Deployment: verification is not optional

Here's the mistake that invalidates all the work above: **verifying the signature without verifying the identity**. In the keyless model, anyone can sign anything; a signature only says "somebody with an OIDC identity signed this." If you don't check *which* identity, an attacker signs their malicious image with their own GitHub account and your verification passes.

That's why `cosign verify` requires you to declare the expected identity and issuer:

```
# Correct: signature + signer identity + certificate issuer
cosign verify ghcr.io/my-org/app@sha256:abc123... --certificate-identity-regexp '^https://
github.com/my-org/app/.github/workflows/release.yml@refs/tags/v.*' --certificate-oidc-issuer
'https://token.actions.githubusercontent.com'
```

With the GitHub CLI the equivalent check is shorter, because the repository already scopes the identity:

```
# Local artifact
gh attestation verify ./dist/app-1.4.2.tgz --repo my-org/app

# Container image
gh attestation verify oci://ghcr.io/my-org/app:1.4.2 --repo my-org/app

# Verifying the SBOM specifically requires naming the predicate type
gh attestation verify oci://ghcr.io/my-org/app:1.4.2 --repo my-org/app --predicate-type https://
spdx.dev/Document
```

Let the cluster verify, not a person

A command in a terminal is a test, not a control. In production, verification has to be an admission rule: Kubernetes rejects the Pod if the image isn't signed by the right identity. With Kyverno:

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
```

```
  name: require-signed-images
spec:
  validationFailureAction: Enforce
  webhookTimeoutSeconds: 30
  rules:
    - name: verify-ghcr-provenance
      match:
        any:
          - resources:
              kinds: [Pod]
              namespaces: [production]
      verifyImages:
        - imageReferences:
            - "ghcr.io/my-org/*"
          mutateDigest: true      # rewrites the tag to the verified digest
          verifyDigest: true
          attestors:
            - entries:
                - keyless:
                    subject: "https://github.com/my-org/*/.github/workflows/release.yml@*"
                    issuer: "https://token.actions.githubusercontent.com"
                    rekor:
                      url: https://rekor.sigstore.dev
```

Note `mutateDigest: true`. Kyverno resolves the tag to a digest, verifies it, and rewrites the manifest so the Pod starts that specific digest. Without it, there's a window between verification and startup in which the tag can change. It's a small race condition and a perfectly exploitable one.

Roll it out in `Audit` mode for a week or two first, see which images would fail, fix the pipelines producing them, and only then switch to `Enforce`. Flipping Enforce on a cluster full of legacy images is the fastest way to get the team to disable the policy.

Beyond npm: the same five doors in Python, Go, Rust and Java

npm gets the headlines because that is where the volume and the noise are, but the threat model does not change when you change language. What changes is which lever each package manager gives you, what it is called and — this is the part that matters — which one ships turned off. This is the map I go through when I land in a repository that is not JavaScript.

Python: a wheel is data, an sdist is code

The distinction almost nobody has internalised: installing a *wheel* unzips an archive and copies files; installing an *sdist* executes that package's `setup.py` or build backend on your machine, with your permissions and your environment variables. It is the exact equivalent of npm's `postinstall`, except there is no switch to turn it off. The way to turn it off is to refuse to build from source.

```
# Binaries only: if there is no wheel for your platform, fail instead of running code
pip install --only-binary=:all: -r requirements.txt

# Hash-checking mode: any artifact whose hash does not match aborts the install
```

```
pip install --require-hashes -r requirements.txt

# Generate that requirements.txt with hashes from your direct dependencies
uv pip compile requirements.in --generate-hashes -o requirements.txt

# With uv as project manager, the strict equivalent of npm ci:
uv sync --locked --no-install-project
```

`--require-hashes` has a very useful side effect: it turns on a strict mode in which **every** dependency, transitive ones included, must be pinned with `==` and carry a hash. You cannot half-forget one. If your pipeline does not use it, your `requirements.txt` is a wish list, not a lockfile.

Cooldowns in Python live in uv's configuration depending on your tooling, and in Renovate through `minimumReleaseAge`, which works the same for PyPI as for npm. And if you publish packages, PyPI has supported OIDC Trusted Publishing since 2023 and has been issuing verifiable provenance attestations since 2024: the same move as npm, with the same benefit of leaving no long-lived token around to steal.

Go: the best default in the industry, and the flag that switches it off

Go did two things nobody else did. First: there are no install scripts. Downloading a module runs nothing; code only runs when you compile and execute. Second: `go.sum` is not just a local file of hashes, it is a check against `sum.golang.org`, an append-only, auditable checksum database Google has operated since 2019. If someone rewrites v1.4.2 of a module after publishing it, the mismatch surfaces on the next download by anyone in the world, not just by whoever happened to hold the old hash.

```
# Check that what is in the module cache matches go.sum
go mod verify

# In CI: fail if the build would need to touch go.mod or go.sum
go build -mod=readonly ./...
go test -mod=readonly ./...

# Pin the toolchain too: without this, the compiler is a floating dependency
export GOTOOLCHAIN=go1.24.5
```

The recurring failure is in corporate configuration. `GOPRIVATE`, `GONOSUMDB` and `GONOSUMCHECK` take glob patterns, and the moment somebody writes `GOPRIVATE=*` or `GONOSUMDB=github.com/*` to unblock an internal proxy, verification against the transparency log disappears for everything. Roughly one in six pipelines is estimated to carry some variable that disables the check, usually set years ago by somebody who has since left. Go look at that value and narrow it to the minimum prefix your private modules need:

```
# Bad: disables verification for half the internet
GOPRIVATE=github.com/*

# Good: your organisation only
GOPRIVATE=github.com/my-company/*
GONOSUMDB=github.com/my-company/*
```

Rust: the lockfile is easy, build.rs is not

Cargo has a deterministic lockfile and `cargo build --locked` does what you expect: it fails if `Cargo.lock` would have to change. Rust's problem is `build.rs`: an arbitrary Rust file that gets compiled and executed on your machine before the actual crate. It is precisely `postinstall`, and unlike npm there is no `ignore-scripts` equivalent, because plenty of legitimate crates need it to link against C.

```
# Reproducible build: abort if the lockfile is not up to date
cargo build --locked --release

# Known vulnerabilities against the RustSec database
cargo audit

# Policy: licences, allowed sources, duplicate crates and advisories
cargo deny check

# Human audits with trust delegated between organisations
cargo vet check
```

`cargo-vet` is the odd one out and deserves a paragraph. Instead of scanning for known CVEs, it keeps a record of which versions of which crates somebody you trust — your team, Mozilla, Google, the Bytecode Alliance — has actually reviewed, and forces you to close the gap when a version nobody has looked at shows up. It is expensive to bootstrap and it is not for everyone; in organisations with serious regulatory requirements it is the only tool that answers the question *has anyone read this code?*.

Java: dependency verification in Gradle

Gradle has a powerful and badly underused verification mechanism: a `gradle/verification-metadata.xml` file holding the checksums — and optionally the PGP signatures — of every artifact your build is allowed to download. Once it exists, any artifact that does not match stops the build.

```
# Generate the initial file from what your build resolves today
./gradlew --write-verification-metadata sha256,pgp help

# From here on every build verifies; use --refresh-dependencies to regenerate
./gradlew build
```

Two practical warnings. One: generate the file with a clean cache, because with dependencies already cached Gradle skips entries and you end up with a verification file full of holes. Two: in older versions (6.2 through 7.4.2) there were cases where Gradle accepted an artifact when signature verification could not be completed; 7.5 fixed this by falling back to the checksum. If you maintain an old build, that is a concrete reason to upgrade.

Containers: your base tag floats too

Everything above falls apart if your `Dockerfile` starts with `FROM node:22-alpine`. That tag points at a different image every week, and what you built yesterday is not what you build today. Pin it by digest, exactly as you pin actions by SHA.

```
# Floating: zero reproducibility
FROM node:22-alpine
```

```
# Immutable: this exact image content
FROM node:22-alpine@sha256:5340cbfc2df14331ab021555fdd9f83f072ce811488e705b0e736b11adeec4bb

# And let Dependabot bump the digest through a reviewable pull request
```

Dependabot and Renovate understand image digests the same way they understand package versions: they open a pull request when a new base image lands, with a link to the release notes. You get reproducibility without giving up updates.

The door that is not on the diagram: the package name

The five doors above assume you know which package you want to install. There is a family of attacks that compromises no maintainer and steals no credential: it gets you to install, willingly and with a perfectly verified lockfile, a package you never wanted. They have become considerably more refined and they deserve their own section.

Dependency confusion: the public registry wins the auction

In 2021 Alex Birsan published packages to npm and PyPI carrying the names of internal modules belonging to Apple, Microsoft, PayPal and thirty- odd others. He did not have to guess a password: installers, seeing a package available on two indexes, picked the highest version, and he published 9.9.9. He got code execution inside the corporate networks of half a dozen tech giants and collected over 130,000 dollars in bug bounties.

The design flaw is still alive in `pip`. The `--extra-index-url` flag does not mean *look here first*: it means *look in both and take the highest version*. If your internal `company-auth` package is on 2.3.1 and somebody publishes `company-auth` 99.0.0 to PyPI, your build installs the PyPI one and you will not see a single warning.

```
# Dangerous: pip queries both indexes and takes the greater version
pip install --extra-index-url https://pypi.company.com/simple -r requirements.txt

# Safe: a single index, a proxy that mirrors PyPI and serves internal packages
pip install --index-url https://pypi.company.com/simple -r requirements.txt
```

`uv` fixed the default: its strategy is *first-index*, meaning it stops at the first index that offers the package and does not look at the others. You can also mark an index as explicit, so that it is used only for packages that ask for it by name.

```
# pyproject.toml
[[tool.uv.index]]
name = "internal"
url = "https://pypi.company.com/simple"
explicit = true          # only for packages that reference it explicitly

[tool.uv.sources]
company-auth = { index = "internal" }
```

```
# And do not go back to the unsafe strategy for convenience:
# UV_INDEX_STRATEGY=unsafe-best-match reintroduces the 2021 bug exactly
```

The npm equivalent is binding the scope to the internal registry, so that `@company/*` never resolves against the public one:

```
# .npmrc, committed to the repository
@company:registry=https://npm.company.com/
//npm.company.com/:_authToken=${NPM_INTERNAL_TOKEN}
ignore-scripts=true
```

And one cheap measure that is still not applied often enough: defensively register your internal names and scopes on the public registry, even as empty packages. It takes ten minutes and closes the door for good.

Typosquatting and slopsquatting: when a model writes the name

Classic typosquatting — `reqeusts` instead of `requests` — has been around for years and is reasonably well handled by the near-name detection PyPI and npm run today. The new variant is more uncomfortable. Coding assistants invent package names that do not exist, and they do it consistently: the same names, over and over, for similar prompts. An attacker only has to register them and wait.

The case that made it obvious: in 2024, Bar Lanyado noticed several models recommending `pip install huggingface-cli`, when the real tool installs as `pip install -U "huggingface_hub[cli]"`. He published an empty package under that name to measure the effect. It received more than 30,000 genuine downloads in three months. A 2026 study across nearly 200,000 code-generation responses measured package-name hallucination rates between 4.62% and 6.10% across five frontier models, and found that 53 of those names — 41 on PyPI and 12 on npm — were still free to register.

The defence is not to stop using assistants: it is to stop letting a package name reach the lockfile without somebody having looked at it. A pull request check that surfaces the data that matters turns an invisible decision into a deliberate one.

```
# scripts/new-dependencies.sh — run it in CI against the PR diff
#!/usr/bin/env bash
set -euo pipefail

base="${1:-origin/main}"

# Names present in package.json on this branch but not on the base
comm -13 \
  <(git show "$base:package.json" | jq -r '.dependencies + .devDependencies | keys[]' | sort) \
  <(jq -r '.dependencies + .devDependencies | keys[]' package.json | sort) \
| while read -r pkg; do
    meta=$(npm view "$pkg" --json 2>/dev/null || echo '{}')
    if [ "$meta" = "{}" ]; then
      echo "::error::$pkg does not exist on the registry. Hallucinated name?"
      continue
    fi
    created=$(echo "$meta" | jq -r '.time.created')
    repo=$(echo "$meta" | jq -r '.repository.url // "no repository"')
```

```
    maint=$(echo "$meta" | jq -r '.maintainers | length')
    age=$(( ( $(date +%s) - $(date -d "$created" +%s) ) / 86400 ))
    echo "::notice::$pkg — created $age days ago, $maint maintainer(s), $repo"
    [ "$age" -lt 90 ] && echo "::warning::$pkg is less than 90 days old. Manual review required."
  done
```

It blocks nothing on its own, and that is the point: it puts the package's age, its maintainer count and whether it points at a real repository into the pull request thread. Most malicious packages in this family are two days old, have one maintainer and no repository. Seeing those three numbers together is enough to make a reviewer ask.

How Sigstore actually works (and why it changes what you verify)

We used `cosign` earlier as if it were magic. It is worth opening the box, because understanding the three pieces explains why correct verification checks an identity rather than a key, and why that is an improvement rather than a quirk.

Fulcio is a certificate authority that does not issue certificates to people but to OIDC identities. You present a token from your provider — the token GitHub Actions mints for your workflow, or your Google account — plus an ephemeral public key you just generated. Fulcio returns a very short-lived certificate, on the order of ten minutes, binding that key to that identity. You sign, and you throw the private key away. There is nothing to rotate and nothing to steal, because ten minutes later there is nothing left.

Rekor is an append-only transparency log, the same family as Certificate Transparency. Every signature is recorded with its certificate and its timestamp. This solves the obvious problem with the previous paragraph: if the certificate expired a year ago, how do you verify today? Because Rekor attests that the signature existed while the certificate was valid. And as a bonus, anyone can audit the log: if somebody signed something as you, it would be written into a public log you can watch.

Cosign is the tool that coordinates the three. And here is the nuance that gets missed: since anyone can obtain a Fulcio certificate for their own identity, *anyone can sign your image*. A valid signature says nothing by itself. What says something is *who* signed and *from where*.

```
# Useless: checks that a valid signature from somebody exists. Anybody.
cosign verify ghcr.io/my-org/api@sha256:abc123...

# Correct: demand a specific identity and issuer
cosign verify ghcr.io/my-org/api@sha256:abc123... \
  --certificate-oidc-issuer=https://token.actions.githubusercontent.com \
  --certificate-identity-regexp='^https://github.com/my-org/api/.github/workflows/
release.yml@refs/tags/v.*$'

# Verify the SLSA provenance attestation, not just the signature
cosign verify-attestation ghcr.io/my-org/api@sha256:abc123... \
  --type slsaprovenance \
  --certificate-oidc-issuer=https://token.actions.githubusercontent.com \
  --certificate-identity-regexp='^https://github.com/my-org/' \
  | jq '.payload | @base64d | fromjson | .predicate.buildDefinition.externalParameters'
```

Look at `--certificate-identity-regexp`: it does not just bind the signature to your organisation, it binds it to the *specific workflow* and to the trigger being a version tag. With that, an image signed from a test branch in your own repository fails verification too. It is least privilege, applied to provenance.

For packages there are shortcuts that do not need cosign. On npm, `npm audit signatures` verifies registry signatures and provenance attestations across your whole dependency tree at once. For artifacts published from GitHub, `gh attestation verify` does the same against the attestations API:

```
# Verify registry signatures and provenance for the whole installed tree
npm audit signatures

# Verify a downloaded binary against the attestation published by the repository
gh attestation verify ./my-binary --repo my-org/my-project

# In an environment with no route to GitHub, using a previously fetched bundle
gh attestation verify ./my-binary --bundle bundle.jsonl --repo my-org/my-project
```

The SBOM people actually use: CycloneDX, VEX and the five-minute test

We already said an SBOM nobody consumes is an expensive JSON file. The test for whether yours earns its keep is concrete: **a critical CVE drops in a compression library at four in the afternoon. How long does it take you to say which services in production ship it, at which version, in which exact image?** If the answer is *we ask in Slack and somebody greps*, the SBOM is not doing its job.

Generating: which format, and at which moment

CycloneDX and SPDX coexist and both are fine. CycloneDX tends to be more comfortable for the security use case — it has VEX built in and a very direct *purl* model — while SPDX carries more weight in compliance and licensing. Pick one and be consistent; converting between them is possible but lossy at the edges.

The moment matters more than the format. An SBOM generated from the lockfile tells you what you asked for; one generated from the built image tells you what is actually inside, including the operating-system packages nobody declared. Generate the second, or both.

```
# From the final image: includes OS packages, not just application dependencies
syft ghcr.io/my-org/api@sha256:abc123... -o cyclonedx-json=sbom.cdx.json

# From the project dependency tree (richer licence metadata)
cdxgen -t js -o sbom.cdx.json .

# Attach it as a signed attestation on the digest, not as a loose file
cosign attest --predicate sbom.cdx.json --type cyclonedx \
  ghcr.io/my-org/api@sha256:abc123...

# Scan the SBOM, not the image: faster and works without registry access
grype sbom:./sbom.cdx.json --fail-on high
```

That last command is what turns the SBOM into something operational: scanning the document instead of the image means you can answer a new CVE without rebuilding or re-pulling anything, and that you can do it for artifacts that are no longer in any registry.

Consuming: an inventory, not a folder of build artifacts

The step almost everyone skips is publishing the SBOM somewhere queryable. Dependency-Track is the open-source reference: you push every SBOM on every build, it keeps the inventory per project and version, and when a new vulnerability appears it tells you which projects are affected without you asking.

```
# .github/workflows/release.yml — excerpt
      - name: Generate SBOM
        run: syft "$IMAGE_DIGEST" -o cyclonedx-json=sbom.cdx.json

      - name: Publish to the inventory
        env:
          DT_URL: ${{ vars.DEPENDENCY_TRACK_URL }}
          DT_KEY: ${{ secrets.DEPENDENCY_TRACK_API_KEY }}
        run: |
          curl -sSf -X POST "$DT_URL/api/v1/bom" \
            -H "X-Api-Key: $DT_KEY" \
            -F "projectName=api" \
            -F "projectVersion=${GITHUB_REF_NAME}" \
            -F "autoCreate=true" \
            -F "bom=@sbom.cdx.json"
```

With that, the four-o'clock question is answered with a query by *purl*. If you do not want to run Dependency-Track, the minimum viable version is storing SBOMs in a bucket with a path convention per service and digest, plus a script that runs `jq` across all of them. It is ugly, but it answers in five minutes, which is the actual requirement.

VEX: half the CVEs in your report do not affect you

Anyone who has run a scanner against a production image knows the feeling: two hundred vulnerabilities, the vast majority of them in code your application never calls. The problem is not that the scanner is lying; it is that without more context it cannot tell *present* from *exploitable*. VEX (*Vulnerability Exploitability eXchange*) is the document where you supply that context, machine-readably and auditably.

The four states are `not_affected`, `affected`, `fixed` and `under_investigation`. The valuable one is the first, because it must come with a justification drawn from a closed vocabulary: the component is not present in the executable, the vulnerable code is not reachable, it cannot be controlled by an adversary, or a mitigation is in place.

```
{
  "bomFormat": "CycloneDX",
  "specVersion": "1.6",
  "vulnerabilities": [
    {
      "id": "CVE-2026-11111",
      "source": { "name": "NVD" },
```

```
      "affects": [ { "ref": "pkg:npm/libcompression@3.2.1" } ],
      "analysis": {
        "state": "not_affected",
        "justification": "code_not_reachable",
        "detail": "The flaw is in the stream decoder. Our API only exposes synchronous compression
over in-memory buffers; the decoder is not linked into the production bundle (verified by
reachability analysis on 2026-08-14).",
        "response": [ "will_not_fix" ]
      }
    }
  ]
}
```

The detail that makes this work over time: the justification is written once and reused on every subsequent scan. Without VEX, your team re-investigates the same CVE every quarter, reaches the same conclusion and writes it down nowhere. With VEX, tomorrow's report already has those two hundred filtered down to the eight that matter, and every exclusion carries a name, a date and a reason.

It happened to you: a first-24-hours runbook

Everything above reduces the probability. It does not take it to zero. The difference between a scare and a serious incident usually comes down to the first hour, and that hour goes far better with a script written in advance than improvised at eleven at night.

Hour 0 — Contain

Before investigating anything, cut off propagation. The logic is simple: every new install can run the payload on one more machine, and every infected machine hunts for credentials to publish more packages.

```
# 1. Turn off lifecycle scripts across the organisation, now
npm config set ignore-scripts true --location=global

# 2. Freeze CI: disable the workflows that install dependencies
gh workflow list --repo my-org/api --json name,id \
  | jq -r '.[].id' \
  | xargs -I{} gh workflow disable {} --repo my-org/api

# 3. Block publishing from the organisation for the duration of triage
#    (npm: revoke publish tokens; with Trusted Publishing, disable the
#     package's trusted publisher in the registry web UI)
```

If you use Trusted Publishing this step is markedly cleaner: there are no tokens scattered across fifty repositories to hunt down, there is one per-package setting you switch off in a minute. It is one of those advantages that never shows up in the comparison table until you need it.

Hour 1 — Establish the blast radius

The question is which exact versions got in and where. Looking at `package.json` is not enough: what

matters is what is resolved in the lockfiles, including transitive dependencies nobody ever declared.

```
# scripts/find-compromised.sh
#!/usr/bin/env bash
# Usage: ./find-compromised.sh compromised.txt  (one "name@version" per line)
set -euo pipefail
list="$1"

find . -name 'package-lock.json' -o -name 'pnpm-lock.yaml' -o -name 'yarn.lock' \
  -o -name 'uv.lock' -o -name 'poetry.lock' | while read -r lock; do
  while read -r entry; do
    name="${entry%@*}"; version="${entry##*@}"
    if grep -qF "$name" "$lock" && grep -qF "$version" "$lock"; then
      echo "MATCH  $lock  ->  $entry"
    fi
  done < "$list"
done

# And inside a given repository, the exact path that pulled it in:
#   npm ls compromised-package
# which shows which of your direct dependencies drags the malicious one along.
```

In parallel, build the timeline: the publication date of the malicious version against the dates of your builds. Every build earlier than the publication is provably clean, and that cuts the work down enormously. CI logs and image timestamps in the registry hand you that list for free.

Hours 2 to 6 — Rotate, in the right order

The classic mistake is rotating credentials from a machine that is still compromised, or rotating in an order that lets the attacker use one credential to renew another. The order that works runs from most powerful to least:

1. **Identity credentials**: GitHub tokens (PATs, app OAuth tokens, deploy keys), because everything else can be obtained with them. Revoke first, reissue afterwards.
2. **Long-lived cloud credentials**: IAM access keys, service accounts with JSON keys. If any of them is unnecessary because you could already be using OIDC, this is the moment not to reissue it.
3. **Package registry tokens** and any signing credentials that are not ephemeral.
4. **Application secrets**: third-party API keys, connection strings, session and webhook signing keys.

And one check that is always forgotten: forks and team members' personal repositories. Worms in this family use stolen credentials to create new repositories — often with a fixed, recognisable name — and to register self-hosted runners that survive secret rotation. Go looking for both explicitly.

```
# Repositories created within the incident window across the organisation
gh api "orgs/my-org/repos?sort=created&direction=desc&per_page=50" \
  --jq '.[] | select(.created_at > "2026-08-10") | [.name, .created_at, .private] | @tsv'

# Registered self-hosted runners: persistence that survives rotating secrets
gh api orgs/my-org/actions/runners --jq '.runners[] | [.name, .status, .os] | @tsv'
```

```
# New or modified workflows in repositories that should not have them
gh api "search/commits?q=org:my-org+path:.github/workflows+committer-date:>2026-08-10" \
  --jq '.items[] | [.repository.full_name, .commit.author.date, .commit.message] | @tsv'
```

Hours 6 to 24 — Rebuild and close out

Rebuild from clean environments, not from the affected machines: an `npm ci` on a compromised laptop does not return you to a good state. Regenerate lockfiles pinned to known-good versions, rebuild the images, and verify the new attestations before deploying — this is exactly the moment the whole provenance machinery proves what it was for.

The postmortem has one question that matters more than the rest, and it is not *how did it get in*. It is *how long did it take us to find out, and why that long?*. Detection almost always arrives from outside — a registry notice, a bulletin, a customer — and that delay is the variable you can genuinely move with everything we have covered: cooldowns, a queryable inventory and verification at admission.

Governing this across fifty repositories: Scorecard, SLSA levels and policy as code

Applying all of the above in one repository is an afternoon. Applying it across a whole organisation and having it still apply six months later is a different problem, and it is solved with three things: a metric, a target and a mechanism that prevents regression.

The metric: OpenSSF Scorecard

Scorecard is an automated analysis that scores a repository from 0 to 10 across roughly twenty checks, several of which are precisely what we have been covering: whether actions are pinned by SHA, whether token permissions are scoped, whether code review is required, whether branches are protected, whether known-vulnerable dependencies are present. The interesting part is that you can also run it against your *dependencies*, not only your own code.

```
# .github/workflows/scorecard.yml
name: Scorecard
on:
  branch_protection_rule:
  schedule: [{ cron: '30 4 * * 1' }]
  push: { branches: [main] }

permissions: read-all

jobs:
  analysis:
    runs-on: ubuntu-latest
    permissions:
      security-events: write   # to upload the SARIF to code scanning
      id-token: write          # to publish the signed result
      contents: read
```

```
    steps:
      - uses: actions/checkout@11bd71901bbe5b1630ceea73d27597364c9af683 # v4.2.2
        with: { persist-credentials: false }
      - uses: ossf/scorecard-action@62b2cac7ed8198b15735ed49ab1e5cf35480ba46 # v2.4.0
        with:
          results_file: results.sarif
          results_format: sarif
          publish_results: true
      - uses: github/codeql-action/upload-sarif@v3
        with: { sarif_file: results.sarif }
```

One piece of advice on how to use the score: do not chase a 10. Chase having no individual check sitting at 0, and a trend that goes up. A dashboard with the score for all fifty repositories, sorted ascending, is a far better prioritisation tool than any meeting.

The target: what each SLSA level actually buys

SLSA is almost always explained badly because it is presented as a compliance ladder. Seen as a ladder of *properties* it is much more useful, because each level answers a different attack:

- **Build L1 — provenance exists.** You know which build produced this binary and from which inputs. It prevents nothing on its own, but it turns a two-day investigation into a query. It costs the five lines of `attest-build-provenance`.
- **Build L2 — provenance is signed by the build service.** It is no longer enough for somebody to write a plausible JSON: the attestation comes from the platform and is verifiable. This blocks forgery by a third party.
- **Build L3 — provenance is unforgeable even by whoever controls the repository.** The build runs in an isolated environment the build process itself cannot manipulate. On GitHub you get there by signing from a reusable workflow whose contents the project cannot alter. This is what stops the attacker who already has write access: they can add malicious code, but they cannot lie about where the binary came from.

For most teams, L2 everywhere and L3 on the artifacts other people consume is an honest target. L3 across every repository is a platform project, not a task.

The mechanism: the template nobody can edit

The way to get fifty repositories doing the same thing is not documentation: it is a reusable workflow that already has permissions, pinning, OIDC publishing and attestation solved, plus an organisation rule that requires its use.

```
# .github/workflows/release.yml in every consuming repository: this is all of it
name: Release
on:
  push:
    tags: ['v*']

permissions: {}          # nothing by default; the reusable workflow asks for its own
```

```yaml
jobs:
  publish:
    uses: my-org/platform/.github/workflows/publish-npm.yml@v3
    permissions:
      contents: read
      id-token: write      # OIDC for Trusted Publishing and for signing
      attestations: write
    with:
      node-version: '22'
    secrets: inherit
```

The consuming repository cannot get the permissions wrong or forget the attestation, because it does not write them. And when the next good practice comes along, you roll it out by bumping a tag in a single place. Combine it with organisation rulesets that require that workflow to run before publishing is allowed, and you have a control that survives team turnover.

What to measure to know this is still alive

Four numbers on a dashboard, reviewed once a month, catch erosion before it becomes a problem:

1. **Percentage of repositories publishing with OIDC** versus those still holding a token in secrets. The target is zero tokens.
2. **Percentage of third-party actions pinned by SHA** out of total usages. A single new `@v4` moves the number and stands out.
3. **Median age of dependencies at install time.** If it drops below your cooldown window, somebody has disabled it somewhere.
4. **Admission policy coverage**: what share of deployments passes through signature verification in *Enforce* mode rather than *Audit*. Staying in Audit for years out of inertia is extremely common.

None of these is a security indicator in the strict sense. All of them indicate whether the controls are still plugged in, which in practice is the question people get wrong.

The mistakes that undo everything above

I've watched all eight of these repeat in serious teams. Most of them break nothing visibly, which is precisely what makes them dangerous.

- **Verifying the signature without verifying the identity.** Said already, but it's the number one failure: `cosign verify` without `--certificate-identity` and `--certificate-oidc-issuer` accepts anyone's signature on anything.
- **Signing the tag instead of the digest.** The signature stays valid while the tag points at a different image.
- `npm install` **in CI.** It turns the lockfile into a suggestion.
- `ignore-scripts` **only in CI.** The developer laptop holds cloud credentials, npm tokens, and SSH keys. It's a better target than your runner. Set it in the user-level config too.
- `id-token: write` **at workflow level.** It grants test jobs the ability to obtain publishing credentials.

- **Cooldown in the installer but not in the bot.** You end up with a queue of pull requests that fail to install, and the team disables the gate out of frustration.
- **An SBOM nobody consumes.** A JSON file in the build artifacts is not a control. If you don't verify it at admission or cross-reference it against vulnerability advisories, it's paperwork.
- **An org-scoped publishing token.** If one package is compromised, every other package goes with it. Minimum scope, or better, no token at all.

If you only have one afternoon

Rollout order by real impact, not by elegance:

1. Add the cooldown (`min-release-age` or its equivalent) and configure it in Dependabot or Renovate too. Half an hour, and it would have protected you from most incidents of the last two years.
2. Set `ignore-scripts=true`, run `npm rebuild` for the short list that genuinely needs it, and add the audit script to the pipeline.
3. Replace `npm install` with `npm ci` across all workflows and declare `permissions: contents: read` at the top.
4. Run `pinact run` and enable the `github-actions` ecosystem in Dependabot.
5. Migrate publishing to Trusted Publishing and delete the repository token. Deleting it is part of the job: a configured but unused token is still a valid token.
6. Add `attest-build-provenance` to the release job. It's five lines.
7. Deploy the admission policy in Audit mode, and switch it to Enforce once the noise reaches zero.

Production checklist

- Every CI install uses the lockfile strictly (`npm ci`, `--frozen-lockfile`, `--immutable`, `uv sync --locked`).
- Lifecycle scripts are off by default, with a short, reviewed exception list.
- A release cooldown is configured in both the installer and the dependency bot, with a documented way to bypass it for a CVE.
- Every third-party action is pinned to a full SHA, and CI fails if someone reintroduces a tag.
- Default workflow permissions are `contents: read`; `id-token` and `attestations` live only on the release job.
- No workflow executes external pull request code in a context that holds secrets.
- Publishing uses OIDC. No long-lived registry token remains in the secrets.
- Every artifact and image carries a signed provenance attestation and SBOM bound to its digest.
- The cluster rejects unsigned images at admission, checking identity and issuer, with `mutateDigest` enabled.
- The incident runbook answers, in under five minutes: which commit produced this binary, and what dependencies does it contain?

FAQ

Isn't a vulnerability scanner enough? No. A scanner compares your inventory against a database of *known* vulnerabilities. A package compromised two hours ago is in no database. Cooldowns and script blocking attack the window in which the scanner still knows nothing.

Isn't a lockfile enough for reproducibility? A lockfile pins *which* versions you install, not *what they do* when installed or who built them. These are orthogonal problems — which is why all five layers exist.

Cooldowns delay security patches. Isn't that counterproductive? That's the central trade-off, and it's why every implementation ships an escape hatch. In practice, the risk of consuming a freshly published malicious version is much higher than the risk of applying a legitimate patch twenty-four hours later. If a critical CVE genuinely affects you, you bypass the gate deliberately.

Does any of this matter if my code is private? Yes, and in some respects more. Your dependencies are still public and your CI still holds production credentials. On top of that, a private repository with cloud secrets is a more profitable target than an open source library.

What exactly do I gain at SLSA Build Level 3? Provenance that is unforgeable even by someone with write access to the repository, because the build runs in an isolated environment the build process itself cannot tamper with. On GitHub you get there by signing from a reusable workflow that the project's code cannot influence.

Where do I start with fifty repositories? With a template. A reusable workflow with permissions, pinning, and OIDC publishing already solved, plus a distributed `.npmrc` carrying the baseline configuration. Migrate the repositories that publish packages or images first; the consumer-only ones can wait for the second pass.

More frequently asked questions

Is a cooldown worth it if my team updates dependencies once a quarter? Even more so, for a non-obvious reason: a quarterly update pulls in hundreds of new versions at once, and some of them will be days old at install time. The cooldown costs nothing in that model — you will not even notice it — and it filters out precisely the risky scenario.

I use self-hosted runners. Does anything change? Quite a lot, and for the worse if you do not account for it. A persistent self-hosted runner shares state between jobs: package caches, Docker credentials, the home directory. A payload that runs during a pull request build can sit and wait for the next job. If you use them, make them ephemeral — a clean machine per job — and never expose them to pull requests from forks.

My organisation runs an internal npm and PyPI proxy or mirror. Does that protect me? It protects you from outages and from dependency confusion if it is configured correctly, but not from a legitimately published malicious package: the mirror will replicate the compromised version just as faithfully. Where it does help a great deal is as the single place to apply a cooldown and a blocklist for the whole organisation at once. That is the best reason to run one.

What about my dev dependencies' dependencies? There are hundreds. They are the most common vector, precisely because nobody looks at them and because they run with the same permissions as everything

else. The practical answer is not to audit them one by one, but to make sure the controls do not discriminate: `ignore-scripts`, cooldowns and a strict lockfile apply to the entire tree without you deciding anything package by package.

Can I verify signatures if my production nodes have no internet access? Yes. Both `cosign` and `gh attestation verify` accept a bundle containing the certificate and the Rekor inclusion proof, making verification fully offline. Fetch the bundle in the build step, which does have network, and publish it alongside the artifact.

My team is three people and none of us does security. What should I do? The first three items on the one-afternoon list: cooldowns, `ignore-scripts` and `npm ci`. They are configuration, not projects; they need no maintenance and they cover the way most real incidents of the last two years actually got in. The rest can wait until there is somebody to maintain it.

Closing

The lesson of the last two years isn't that we need more sophisticated tooling. It's that the most effective defenses are remarkably boring: a twenty-four hour delay, a switch that turns off scripts, forty characters instead of two, and one permission moved from one place to another. None of those lines will ever appear in an impressive demo. All of them would have stopped a phishing email to a maintainer from ending up on your machines.