

UNIVERSIDAD NACIONAL DE INGENIERÍA

Ingeniería de Software

Integración continua y dependencias

El build · GitHub Actions · Protección de rama · Semver y lockfile · Cadena de suministro

Gerson Garrido

Semana 12

Hoy

Sesión A

1. Integración continua: una práctica
2. El build, sin compilación
3. Presubmit y post-submit
4. GitHub Actions y su seguridad
5. Un pipeline que impide algo

Sesión B

6. Semver y lockfile
7. Dependencias transitivas
8. Evaluar y atacar una dependencia
9. Paquetes que no existen
10. Leer `npm audit`

Lecturas

SWEG, caps. 23 *Continuous Integration* y 18 *Build Systems and Build Philosophy*, hasta *Modern Build Systems*.

SWEG, cap. 21 *Dependency Management*. Russ Cox, *Our Software Dependency Problem*.

Complementarias: Fowler, *Continuous Integration* (2024); GitHub, *Actions*, *Workflow syntax*, *Secure use reference*; Versionado semántico 2.0.0; npm, `npm ci`, `package-lock.json`, `npm audit`; Spracklen y otros (USENIX Security 2025); SLSA; OpenSSF Scorecard.

La idea de hoy

Un build solo informa si sus entradas están bajo control.

- Las entradas de un build: su código y todo el código de otros que entra con él.
- Semana 11: `make check` encuentra errores. Pero alguien tiene que correrlo.
- Hoy instalarán seis paquetes para revisar Kallpa. Llegarán cien.

Salidas reales del contenedor, 2 de noviembre de 2026. Las que usan el registro de npm cambian con el tiempo.

SECCIÓN 01

Integración continua

01



Integración continua

Integración continua ***(continuous integration)***

Cada integrante integra su trabajo a la rama principal al menos una vez al día, y cada integración se verifica con un build automático que incluye las pruebas.

Integrar seguido
Una decisión del equipo.

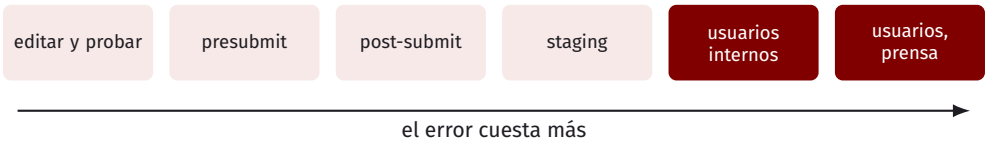
Verificar cada integración
Un servidor que corre un build.

Un servicio de CI sobre feature branches

Fowler: *"yes – but you're not doing Continuous Integration"*. Un pipeline perfecto con ramas de tres semanas tiene un build automático, no integración continua.

Fowler, 2024

Esperar sale caro



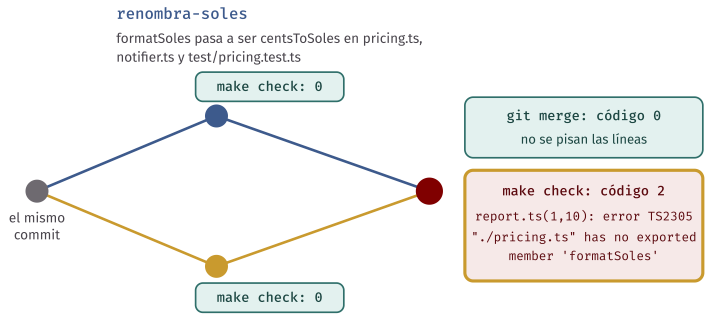
- Lo investiga alguien que no conoce el cambio.
- Quien lo escribió ya no se acuerda.
- Ya afecta a otras personas.

Lo mismo vale para el tiempo entre integraciones

Integrar cada dos horas: dos horas de cambios sospechosos. Cada dos semanas: dos semanas.

SWEG, cap. 23

Dos ramas sin una línea en común



lo que está roto es la combinación, y nadie la había probado

Cada una por separado está perfecta

Las dos revisiones pudieron aprobar con razón. Los dos pipelines salen en verde con razón.

Un merge en rojo

code/week-12

```
--- rama renombra-soles
make check: código de salida 0
--- rama linea-de-caja
make check: código de salida 0
--- las dos juntas

$ git merge --no-edit linea-de-caja
Merge made by the 'ort' strategy.
...
$ make check
tsc -p .
src/domain/report.ts(1,10): error TS2305: Module '"./pricing.ts"' has no exported member 'formatSoles'.
make: *** [Makefile:28: typecheck] Error 2
[código de salida: 2]
```

- Fowler: conflicto semántico. SWEG: mid-air collision.
- Lo roto es la combinación, y nadie la había probado. Con ramas cortas, hay pocas.

`make colision`

Las reglas que la hacen funcionar

Práctica	En Kallpa
Una rama principal, con todo adentro	`main`, con configuración y lockfile
Build automático y *self-testing*	`make check` corre `node --test`
Todos integran a la principal cada día	Pull requests chicos (semana 11)
Cada push dispara un build	El workflow de hoy
Arreglar el build de inmediato	Arreglar o revertir
Un build rápido (guía de XP: 10 minutos)	40 pruebas en décimas de segundo
Todos ven el resultado	La pestaña *Checks*

Kent Beck, citado por Fowler

"Nobody has a higher priority task than fixing the build." Muchas veces lo mejor es revertir (también SWEG).

Si tarda una hora

Nadie espera el resultado, y el rojo llega cuando la persona ya no se acuerda del cambio.

Fowler, 2024

SECCIÓN 02

El build

02

¿Qué es el build, si no hay compilación?

> **Un buen build**
> **Rápido**: un comando, un resultado pronto. **Correcto**: mismas entradas, mismo resultado.

Node quita los tipos y corre: nada genera JavaScript. Pero romper el build incluye romper las pruebas. En la semana 13 se agrega la imagen.

Makefile de Kallpa, resumido

```
check: typecheck lint test

typecheck:  tsc -p .
lint:       eslint .
test:       node --test "test/**/*.test.ts"
```

SWEG, caps. 18 y 23

Las pruebas pasan, `tsc` no

code/week-12

```
$ make test | tail -8
ℹ tests 40
ℹ pass 40
ℹ fail 0
...
$ make typecheck
tsc -p .
src/notifications/notifier.ts(21,146): error TS2345: Argument of type 'string' is not assignable to parameter of type 'number'.
make: *** [Makefile:28: typecheck] Error 2
[código de salida: 2]
```

`make check` es el compilador que no tienen

Node quitó los tipos sin mirarlos, y `cents / 100` convierte el texto en número. Sin `tsc`, los tipos son comentarios.

`make tipos`: `formatSoles(String(c.priceCents))`

Reproducible y hermético

Build reproducible

Con las mismas entradas da el mismo resultado, en cualquier máquina y en cualquier momento.

Build hermético

No depende de nada que no haya declarado: lo instalado en la máquina, la zona horaria, la red, otro servicio.

- SWEG, cap. 18: las **herramientas son dependencias**; las acciones corren aisladas; lo externo se fija con un hash.
- *"A build might work one day and fail the next for no obvious reason."*
- SWEG usa *hermético* sobre todo para pruebas (cap. 23).

“En mi computadora pasa”

code/week-12

```
$ TZ=America/Lima node --test ejemplos/fecha-local.check.ts | grep -E '^(✓|✗) la fecha|^i (pass|fail)'
✓ la fecha según la máquina (0.489541ms)
✓ la fecha en Lima (0.185209ms)
i pass 2
i fail 0

$ TZ=UTC node --test ejemplos/fecha-local.check.ts | grep -E '^(✓|✗) la fecha|^i (pass|fail)' | head -4
✗ la fecha según la máquina (1.120167ms)
✓ la fecha en Lima (0.20975ms)
i pass 1
i fail 1
```

- Las 21:00 del viernes en Lima ya son sábado en UTC. Los runners de GitHub están en UTC.
- La zona horaria era una entrada no declarada.
- Kallpa ya lo evita: `limaTime`, reloj fijo en pruebas, base de datos `:memory:`.

```
make zona
```

Las entradas del build de Kallpa

Entrada	¿Fijada?	Cómo
El código	Sí	El commit
Las seis herramientas	Desde hoy	Versión exacta en `package.json`
Los otros 94 paquetes	Desde hoy	Versión y hash en el lockfile
Hora, zona, base de datos	Sí	`fixedClock`, `limaTime`, `:memory:`
Versión de Node	A medias	`node-version: 26` acepta cualquier 26.x
Sistema del runner	A medias	`ubuntu-latest` se mueve: a Ubuntu 26.04 en noviembre de 2026
Registro de npm	No	`npm ci` lo necesita disponible

No todo se fija. Hay que saber qué no

Si el pipeline se pone rojo un lunes sin que nadie haya tocado nada, la lista de sospechosos ya está escrita.

Tareas y artefactos

Basado en tareas

Ant, Maven, Gradle, Make.

- Scripts que dependen de otros.
- Dicen *cómo* construir.
- Pueden hacer cualquier cosa: el sistema no sabe qué leen ni qué escriben.
- Difícil de paralelizar y de hacer incremental.

Basado en artefactos

Blaze, Bazel, Pants, Buck.

- Declaran *qué* construir, y a partir de qué.
- El sistema decide los pasos.
- Conoce todas las entradas: puede aislar, repetir y guardar.

> **Es una necesidad de escala**
>
> El `Makefile` de Kallpa no lo necesita. Las lecciones sí se trasladan: declarar herramientas, aislar, nada de "última versión".

SWEG, cap. 18, *Modern Build Systems*

La regla

El pipeline llama a lo mismo que corre una persona en su computadora,
no a una copia con otros comandos.

- Dos definiciones de “está bien” terminan discrepando.
- `make check` en los dos lados: un paso nuevo se agrega en un solo lugar.
- El costo: en el contenedor, `tsc` está en el `PATH`. En un runner, no.

SECCIÓN 03

Presubmit y post-submit

03



Qué corre antes de integrar

Presubmit

Antes de integrar. Solo pruebas rápidas y confiables.

Post-submit

Después de integrar. Pueden ser más lentas o menos estables, con mecanismos para manejarlas.

¿Por qué no todo antes?

1. Es caro: las pruebas pasan mucho más seguido de lo que fallan.
2. La inestabilidad bloquea a quien no hizo nada mal.
3. No alcanza: el repositorio cambia mientras corren.

En GitHub: `pull_request` (presubmit) y `push` a `main` (post-submit).

SWEG, cap. 23

Pruebas inestables

Una **flaky test** a veces pasa y a veces falla sin que nada haya cambiado.

$$P(\text{rojo sin razón}) = 1 - (1 - p)^n$$

- 40 pruebas con 1 % de falla: 33.1 % de los builds.
- Una sola prueba con 5 %, en 20 builds: 64.2 % de ver un rojo falso.
- El equipo aprende a volver a correr hasta que pase. También el día que el rojo es real.

Reintentos

Esconden el síntoma y se tragan un error intermitente real. Primero, sacarla de presubmit con una tarea; después, encontrar la entrada no declarada.

`flaky.ts`. Supone fallas independientes

Con IA: “haz que el pipeline pase”

- Ilustración: cambios como los que suele proponer un agente.
  - Subir un timeout.
  - Agregar un reintento.
  - Marcar la prueba con `skip`.
  - Cambiar lo que comprueba.
- En el diff parecen inofensivos.

La pregunta de la revisión

No es si ahora pasa. Es por qué fallaba, y dónde está la explicación.

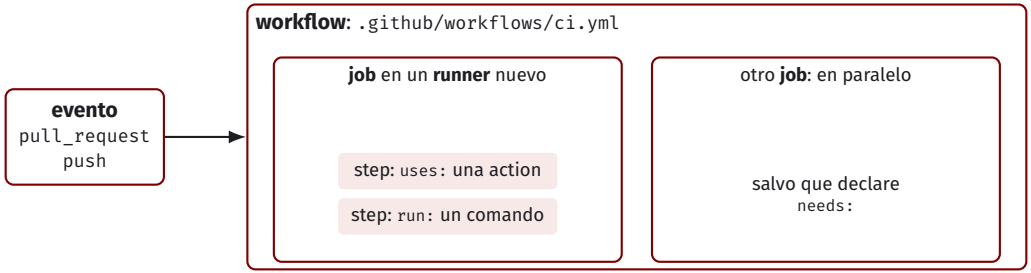
SECCIÓN 04

GitHub Actions

04



Las piezas



Todo descansa en una convención

Los steps corren en secuencia. Un step falla si su comando sale con un código distinto de cero, y los siguientes no corren.

Un primer intento

```yaml
name: CI

on:
  pull_request:
  push:
    branches: [main]

permissions:
  contents: read
```

```yaml
jobs:
  check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v7
      - uses: actions/setup-node@v7
        with:
          node-version: 26
      - run: make check
```

La forma está bien, y en un runner no llega a correr nada. ¿Por qué?

`ejemplos/workflow-incompleto.yml`

Línea por línea

- `on`: por omisión, cuando el pull request se abre, se sincroniza o se reabre; y cada push a `main`.
- `permissions`: limita el `GITHUB_TOKEN`, la credencial temporal del workflow.
- `check`: un id, y el nombre del check que elegirán en la protección de rama.
- `actions/checkout`: un runner nuevo está vacío.
  - En un `pull_request` se prueba el commit de merge (`refs/pull/N/merge`): la rama junto con `main`, como estaba cuando corrió.
- `actions/setup-node`: el runner trae otro Node; aquí se pide el 26.
- `make check`: el mismo comando de siempre.

No llega a correr nada

La imagen del runner trae Git y `make`, pero no `tsc` ni ESLint.

Simular el PATH del runner

contenedor

```
$ env PATH=/usr/local/bin:/usr/bin:/bin make check
tsc -p .
make: tsc: No such file or directory
make: *** [Makefile:28: typecheck] Error 127
[código de salida: 2]

$ env PATH="$PWD/node_modules/.bin:/usr/local/bin:/usr/bin:/bin" make check | tail -9
...
ℹ tests 40
ℹ suites 6
ℹ pass 40
ℹ fail 0
...
```

Faltan dos cosas:

1. Instalar las herramientas: declararlas en `package.json` y correr `npm ci`.
2. Que `make` las encuentre: `$GITHUB_PATH` agrega al `PATH` de los siguientes steps.

Un job colgado no falla

- Un step falla si sale con un código distinto de cero. Un proceso colgado no sale.
- Por omisión, GitHub cancela el job a los 360 minutos.
- Pasa de verdad: `tsc` se colgó 2 veces en 1200 corridas sobre Kallpa, con la máquina saturada (bug de Node al salir, `nodejs/node#54918`).

```yaml
jobs:
  check:
    runs-on: ubuntu-latest
    timeout-minutes: 10
```

Un límite bastante mayor que lo normal: raro, pero sin él, caro.

Lo que el contenedor esconde

Sin declarar `@eslint/js`, que `eslint.config.js` importa:

contenedor del curso

```
$ make check | tail -8
ℹ tests 40
ℹ suites 6
ℹ pass 40
ℹ fail 0
...
```

imagen oficial de Node

```
$ make check
tsc -p .
eslint .

Oops! Something went wrong! :(
...
Error [ERR_MODULE_NOT_FOUND]: Cannot find package '@eslint/js'
  imported from /tmp/kallpa/eslint.config.js
...
make: *** [Makefile:31: lint] Error 2
```

El pipeline da información que el contenedor no puede dar

Los `import` encuentran los paquetes de la imagen. El pipeline es la primera vez que el proyecto se construye solo con lo que declaró.

`make falta; sh runner-limpio.sh @eslint/js`

El código de salida manda

Un `run:` sin `shell:` corre con `bash -e {0}`. Con `shell: bash`, con `-eo pipefail`.

contenedor

```
$ bash -e -c 'false | tee /tmp/log; echo sigue'
sigue
[exit 0]

$ bash -eo pipefail -c 'false | tee /tmp/log; echo sigue'
[exit 1]
```

Pipelines que no impiden nada

`run: make check | tee check.log` sin más: .
`continue-on-error: true` en el job principal. `npm test || true`.

Por eso el paso 5 los hace romperlo a propósito.
GitHub, *Workflow syntax*

Un revisor de workflows

code/week-12

```
$ node revisar-workflow.ts ejemplos/workflow-incompleto.yml
error  npm-ci: Falta npm ci: en el runner no están las herramientas.
error  path: Falta agregar node_modules/.bin al PATH: make no encontraría tsc.
aviso  fijar: actions/checkout@v7 usa una etiqueta, que se puede mover; lo más seguro es el SHA del commit.
aviso  fijar: actions/setup-node@v7 usa una etiqueta, que se puede mover; lo más seguro es el SHA del commit.
2 errores, 2 avisos
```

- Análisis estático, con los límites de la semana 11.
- Revisa la forma del archivo. No dice si el workflow funciona: eso solo lo dice el runner.
- `ejemplos/workflow-peligroso.yml` junta lo que no hay que hacer: 11 errores.

`revisar-workflow.ts`

Permisos y secrets

Un pipeline corre, con credenciales, código que nadie aprobó todavía, y baja código de terceros.

Permisos mínimos

- `permissions: contents: read`.
- Desde un fork: sin secrets y token de solo lectura.
- `pull_request_target` corre con los permisos de la rama base. Con código del fork, es darle sus credenciales a un desconocido.

Secrets

- En la configuración de GitHub. Nunca en el código ni en el workflow.
- GitHub los tapa en los logs, pero la redacción no está garantizada: busca coincidencias exactas.
- Si uno llega a un log: se borra el log y se cambia el secret.

GitHub, *Secure use reference*

Una etiqueta se puede mover

`tj-actions/changed-files`, marzo de 2025

El 14 y 15 de marzo alguien movió las etiquetas `v1` a `v45.0.7` a un commit malicioso. Quien tenía `@v45` lo corrió sin cambiar nada, y los atacantes podían descubrir secrets leyendo los logs.

terminal

```
$ git ls-remote https://github.com/actions/checkout 'refs/tags/v7*'
3d3c42e5aac5ba805825da76410c181273ba90b1    refs/tags/v7
9c091bb21b7c1c1d1991bb908d89e4e9dddfe3e0    refs/tags/v7.0.0
3d3c42e5aac5ba805825da76410c181273ba90b1    refs/tags/v7.0.1
```

en el workflow

```
- uses: actions/checkout@3d3c42e5aac5ba805825da76410c181273ba90b1 # v7.0.1
```

CVE-2025-30066

SECCIÓN 05

Un pipeline que impide algo

05

Un pipeline decorativo

Sin configurar, el botón de merge funciona igual con el check en rojo.

Protección de rama, en *Settings* → *Branches* o *Rules*:

- **Require a pull request before merging**, con una aprobación.
- **Require status checks to pass before merging**, con el check de su job.

Por omisión, no se aplica a los administradores

Salvo con *Do not allow bypassing the above settings*. En un equipo de estudiantes, todos suelen ser admin.

Pública: gratis. Privada: GitHub Pro (gratis con GitHub Education), Team o Enterprise.

Estricto o relajado

Relajado

- La rama no tiene que estar al día.
- Menos builds.
- `make colision`: las dos entran en verde, y `main` queda en rojo.

Estricto: *Require branches to be up to date*

- Cuando entra la primera, la segunda queda desactualizada.
- Trae los cambios, su check corre de nuevo y sale en rojo antes del merge.
- Más builds por cada pull request abierto.

Una cola de merge es la misma idea

GitHub: *Require merge queue*. SWEG: los proyectos chicos evitan los choques en el aire integrando de a uno.

GitHub, *About protected branches*

Con IA: el workflow es parte del perímetro

- El pipeline atrapa lo que un agente rompió sin darse cuenta.
- Por eso un agente no recibe permiso para saltarse los checks ni para cambiar el workflow sin revisión.
- Con la meta “que quede en verde”, tiene incentivos para desactivar una prueba, y puede seguir instrucciones escondidas en lo que lee (semana 1).
- Un cambio en `.github/workflows/` se revisa como un cambio de seguridad. CODEOWNERS lo hace obligatorio.

Un pipeline da información. La protección de rama la vuelve regla

Sin *required status checks*, un check en rojo es una opinión.

SECCIÓN 06

Dependencias

06



Código del que responden

> **Una dependencia**
>
> *"Code you download from the internet [...] someone else on the internet, someone you often don't know."*

- Sus errores, vulnerabilidades y cambios son ahora de ustedes. Agregarla toma segundos; mantenerla, años.
- SWEG: comparada con el control de versiones, agrega complejidad en **tiempo** (el cambio ocurre afuera, sin aviso) y en **escala** (es una red, no un paquete).
- *"Prefer source control problems to dependency management problems."*
- Kallpa no tiene ninguna de producción, a propósito: `node:http`, `node:sqlite`, `node:test`.
- `devDependencies` no llegan a producción, pero corren en sus máquinas y en el runner. Riesgo distinto no es riesgo cero.

Russ Cox, 2019; SWEG, cap. 21

Versionado semántico

Sube	Cuando	Ejemplo
MAJOR	Se rompe la compatibilidad	`10.14.0` → `11.0.0`
MINOR	Se agrega algo compatible	`10.14.0` → `10.15.0`
PATCH	Se corrige algo, de forma compatible	`10.14.0` → `10.14.1`

Es una promesa, no una garantía

SWEG: una estimación con pérdida de cuánto riesgo cree una persona que tiene su cambio.

- **Promete de más**: ley de Hyrum. Un PATCH de typescript-eslint que arregla un falso negativo pone `make check` en rojo en un archivo que nadie tocó.
- **Restringe de más**: un MAJOR por una función que ustedes no usan bloquea la actualización.
- *"Testing and CI provide actual evidence."* Semver dice qué se espera; el pipeline, qué pasó.

Los rangos

Rango	Acepta	No acepta
`8.72.1`	Solo `8.72.1`	`8.72.2`
`~8.72.1`	`8.72.1` a `8.72.x`	`8.73.0`
`^8.72.1`	`8.72.1` a `8.x.y`	`8.72.0`, `9.0.0`
`^0.23.5`	`0.23.5` a `0.23.x`	`0.24.0`
`^0.0.3`	Solo `0.0.3`	`0.0.4`

`^` deja fijo el primer número distinto de cero

Antes de la 1.0, un MINOR puede romper cosas. No es de libro: `eslint` pide `@eslint/config-array@"^0.23.5"`.

`semver-range.ts`, comparado con el paquete `semver` en 5000 casos

El mismo archivo, dos resultados

code/week-12/build/rangos

```
$ npm install --no-audit --no-fund --before=2026-03-01
added 68 packages in 1s
$ npm ls eslint --depth=0
`-- eslint@10.0.2

$ npm install --no-audit --no-fund
added 77 packages in 1s
$ npm ls eslint --depth=0
`-- eslint@10.14.0
```

Con rangos, `package.json` describe un conjunto de instalaciones posibles

Nadie tocó el archivo, y hay otra versión de ESLint y nueve paquetes más.

`make rangos`: `"eslint": "^10.0.0"`, sin lockfile

El lockfile

Versión exacta y hash de cada paquete, también los transitivos:

package-lock.json

```
"node_modules/minimatch": {
  "version": "10.2.6",
  "resolved": "https://registry.npmjs.org/minimatch/-/minimatch-10.2.6.tgz",
  "integrity": "sha512-vpLQEs+VLCr1nU0BXS07maYoFwlDAH0gngQuuttxIwutDFEMHq2blX+8vpgxDdK3J1PwjCJiep77OitTZ4Ll1A==",
  "dev": true,
  "license": "BlueOak-1.0.0",
  ...
}
```

- Nadie pidió `minimatch`: lo piden con `^10.2.4`, `^10.2.2` y `^10.2.6`.
- `–save-exact` fija las seis directas. No fija las otras 94.
- El lockfile entra al repositorio; `node_modules/`, no.

`npm install` y `npm ci`

	`npm install`	`npm ci`
Para qué	Agregar o cambiar	Reproducir
¿Modifica el lockfile?	Puede	Nunca
Sin lockfile	Lo crea	Falla
No coincide con `package.json`	Resuelve y actualiza	Falla
`node_modules` existente	Lo actualiza	Lo borra antes

En el pipeline siempre va `npm ci`

`npm install` no actualiza todo cada vez: si el lockfile satisface a `package.json`, lo respeta. El peligro es que resuelve en silencio cuando no coinciden.

npm, `npm ci`

Tres fallas buenas

contenedor

```
--- alguien edita package.json y se olvida del lockfile
$ npm ci
npm error code EUSAGE
npm error 'npm ci' can only install packages when your package.json and package-lock.json or npm-shrinkwrap.json are in sync. ...
npm error Invalid: lock file's fast-check@4.10.2 does not satisfy fast-check@4.10.1
...
--- no hay lockfile: otro EUSAGE
--- el hash de ms en el lockfile no es el del paquete que llega
$ npm ci 2>&1 | grep -E 'code EINTEGRITY|integrity checksum' | cut -c1-110
npm error code EINTEGRITY
npm error sha512-iK2f+YrcmoeGqk6fA0ea2bptcu/itMIm4NfEozq6N25+aG6h7s5HZbB/k1aV7b5w5sFLMCbbtRUsTVR+BgC3xw== inte
```

El lockfile protege contra cambios, no contra malas decisiones

Lo instalado hoy es lo de la primera vez. No dice que eso fuera bueno. `event-stream` (2018): el atacante no pudo cambiar la 3.3.5, y publicó una 3.3.6.

`make lockfile`

Seis se vuelven cien

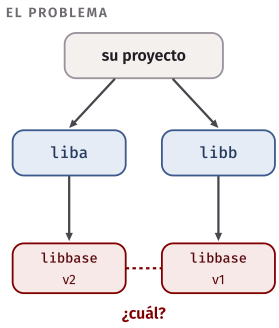
code/week-12

```
Paquetes instalados: 100
Directos y lo que traen por sus dependencies:
  @eslint/js@10.0.1: 0
  @types/node@26.6.6: 1
  eslint@10.14.0: 76
  fast-check@4.10.2: 1
  typescript@6.0.3: 0
  typescript-eslint@8.72.1: 26
Con más de una versión en el árbol:
  eslint-visitor-keys: 3.4.3, 5.0.1
  hookified: 1.15.1, 2.2.0
  ignore: 5.3.2, 7.0.14
...
Con script de instalación: 0
Sin hash: 0
Descargados de: registry.npmjs.org
```

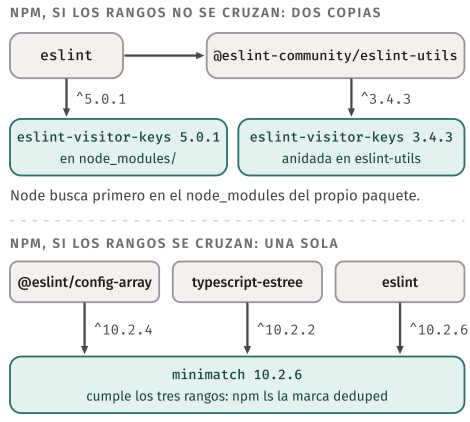
- 1399 líneas de lockfile y unos 50 MB.
- Los números no suman 100: un paquete viene por varios caminos.
- Su número puede ser otro: las transitivas se resuelven el día que instalan.
- ¿Quién lo pidió? `npm explain`.
- `left-pad` (2016): 8 líneas borradas rompieron los builds de la mayoría de los usuarios de Node (Cox).

```
make reporte
```

El problema del diamante



Si solo puede quedar una, una de las dos bibliotecas se queda sin la suya. En C++, tener dos da comportamiento indefinido.



SWEG, cap. 21

Cómo lo resuelve cada lenguaje

- Dos versiones incompatibles de lo mismo.
- Cada lenguaje lo tolera distinto. C++, casi nada. Node, mucho: cada paquete busca primero en su propia carpeta.
- npm instala las dos: `eslint-visitor-keys` 5.0.1 arriba, y 3.4.3 anidada en `eslint-utils`.
- Si los rangos se cruzan, una copia: `minimatch`.

> **El diamante vuelve cuando dos copias no sirven**
>
> Un plugin de ESLint tiene que enchufarse en el ESLint que está corriendo: `peerDependencies`.

`peerDependencies`: por qué seguimos en TypeScript 6

contenedor

```
$ npm view typescript-eslint@8.72.1 peerDependencies
...  typescript: '>=4.8.4 <6.1.0'
$ npm outdated
Package     Current  Wanted  Latest  Location                 Depended by
typescript    6.0.3   6.0.3   7.0.2  node_modules/typescript  auditar
```

Si igual actualizamos a la 7:

1. `npm install`: solo avisa (`ERESOLVE overriding peer dependency`) y sale con 0.
2. `make typecheck` pasa. `make lint`: `typescript-eslint does not support TS 7.0.`
3. `npm ci`: `npm error code ERESOLVE`.

Con IA

`–legacy-peer-deps` y `–force` apagan el reclamo sin arreglar nada: el `as any` de las dependencias.

`make auditar`, `make peer`

SECCIÓN 07

Evaluar y atacar

07

Antes de agregar una dependencia

Pregunta	Qué mirar	Cómo
¿Existe y es el que creo?	Nombre exacto y repositorio	`npm view`
Diseño	¿Documentación clara? ¿Promete compatibilidad?	README
Calidad del código	¿Querrían depurar esto?	Su repositorio
Pruebas	¿Tiene? ¿Pasan?	Su repositorio y su CI
Errores	¿Reportes viejos sin respuesta?	Sus issues
Mantenimiento	¿Quién? ¿Último release?	`maintainers`, `time`
Uso	Ayuda, no garantiza	npmjs.com
Seguridad	¿Entradas no confiables? ¿Historial?	NVD, `npm audit`
Licencia	¿Tiene? ¿Aceptable?	`license`
Dependencias	Cada una es otra inspección	`dependencies`

Y antes que todas: ¿unas pocas líneas o `node:` alcanzan? *"A little copying is better than a little dependency."*

Cox: nueve categorías. La primera fila es nuestra

“Existe” es el primer paso

contenedor

```
$ npm view fast-check name version license repository.url time.modified dependencies
name = 'fast-check'
version = '4.10.2'
license = 'MIT'
repository.url = 'git+https://github.com/dubzzz/fast-check.git'
time.modified = '2026-09-19T19:34:55.280Z'
dependencies = { 'pure-rand': '^8.0.0' }

$ npm view request name version license repository.url time.modified deprecated
name = 'request'
version = '2.88.2'
license = 'Apache-2.0'
repository.url = 'git+https://github.com/request/request.git'
time.modified = '2026-07-17T17:10:13.431Z'
deprecated = 'request has been deprecated, see https://github.com/request/request/issues/3142'
```

- `request` existe, fue muy popular y está deprecado. Hoy Node trae `fetch`.
- `time.modified` no es el último release: la 2.88.2 es de febrero de 2020.

`make paquetes`

Si la aceptan

- **Pruébenla**: lo que necesitan de ella, para cada actualización.
- **Abstráiganla**: un wrapper con solo lo que usan (semana 5).
- **Aíslenla**, si procesa algo peligroso.
- **Evítenla**, si nada de eso alcanza.
- **Actualícenla**: diez cambios chicos cuestan menos que uno grande.
- **Vigílenla**: la versión, lo nuevo que entra, si sigue viva.

Cox, 2019

Equifax, 2017

Struts publicó su corrección el 7 de marzo. Los atacantes entraron el 13 de mayo: datos de 148 millones de personas. Sus escaneos no encontraron esos servidores.

Déjenlo escrito

Cada dependencia de producción va con un ADR (semana 7), y P4 pide `docs/dependencias.md`.

La cadena de suministro

Cadena de suministro ***(supply chain)***

Todo lo que interviene para que su software llegue a producción: paquetes, herramientas, actions, imágenes, el registro, las cuentas de quienes publican.

- **Typosquatting**: un nombre casi igual a uno popular.
- **Dependency confusion**: un paquete público con el nombre de uno interno y una versión más alta (Birsan, 2021). Defensa: scopes ligados a su registro.
- **Mantenedor comprometido o malicioso**: el paquete es el correcto.

debug, septiembre de 2025

GitHub marcó la 4.4.2 como malware. Con un rango como `^4.3.1`, un `npm install` sin lockfile esos días podía bajarla; un `npm ci` con lockfile anterior, no.

xz utils: más de dos años de paciencia



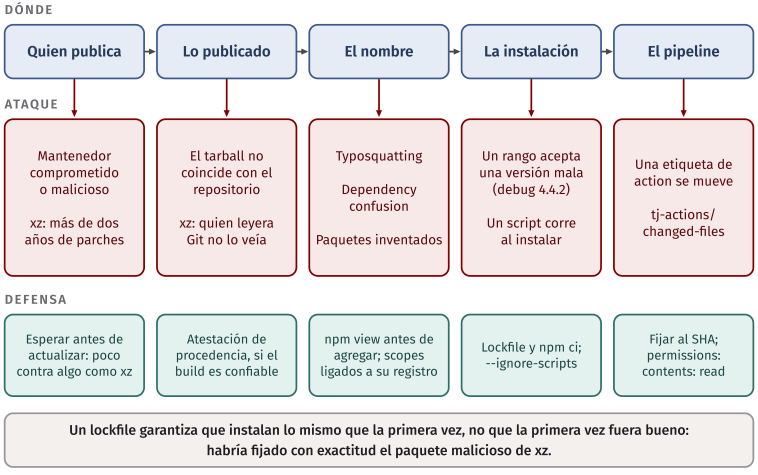
- El archivo que la activaba estaba en el tarball, no en el repositorio. Quien leyera Git no lo veía.
- Un lockfile habría fijado con exactitud el paquete malicioso.
- Esto es lo que ataca la procedencia: ¿lo publicado salió de ese código?

Russ Cox, *Timeline of the xz open source attack*; CVE-2024-3094

Defensas, y lo que no cubren

Defensa	Garantiza	No garantiza
Lockfile con hashes	Lo mismo que la primera vez	Que la primera vez fuera bueno
`npm audit`	Vulnerabilidades ya publicadas	Lo que nadie reportó
`npm audit signatures`	Firma del registro; procedencia, si hay	Que el código sea bueno
Actions fijadas al SHA	No cambian sin que lo sepan	Que ese commit sea bueno
SBOM	Saber qué tienen	Nada por sí solo: es una lista
SLSA	Niveles sobre cómo se construyó	Que la fuente sea buena
OpenSSF Scorecard	Revisión rápida de prácticas	Mide hábitos, no seguridad
`min-release-age`	Lo recién publicado espera	Contra ataques largos, como xz
`–ignore-scripts`	Instalar no ejecuta código	El código corre al usarse

Cada tramo, su ataque y su defensa



Firmas y SBOM en Kallpa

contenedor

```
$ npm audit signatures
audited 100 packages in 2s

100 packages have verified registry signatures

33 packages have verified attestations
...
$ npm sbom --sbom-format cyclonedx | jq -r '.bomFormat, .specVersion, (.components | length)'
CycloneDX
1.5
100

$ npm sbom --sbom-format cyclonedx --omit=dev | jq '.components | length'
0
```

- **Firma**: es el archivo que publicó el registro. **Atestación**: de qué repositorio y commit salió.
- Solo 33 de 100 la tienen: de los otros sabemos menos. A producción llegan cero.

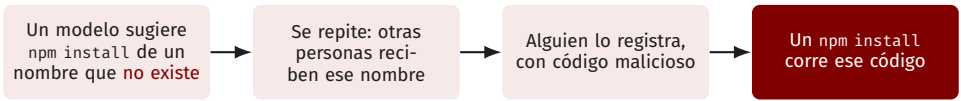
`make auditar`, `make sbom`

SECCIÓN 08

Paquetes que no existen

08

Slopsquatting



El término circula en la industria. No viene del artículo que vamos a leer, que habla de *package hallucinations*.

Lo que midió el estudio

- **Qué hicieron.** 576 000 muestras de código con 16 modelos, en Python y JavaScript. Compararon cada nombre con las listas de PyPI y npm del 10 de enero de 2024.
- **Qué encontraron.** De 2.23 millones de nombres, 440 445 (19.7 %) no existían: 205 474 nombres distintos. Al menos 5.2 % en modelos comerciales y 21.7 % en abiertos.
- **¿Se repiten?** Con 500 prompts, 10 veces cada uno: 43 % salió las 10 veces, 39 % ninguna, y 58 % más de una vez.

Sus límites

Porcentajes de los *nombres generados*, no de programas ni del uso real.
Modelos de 2023–2024; los comerciales son tres de la serie GPT. No describe una herramienta de hoy.
81 % de los nombres salió de un solo modelo. Mide el riesgo, no los ataques.

Spracklen y otros, USENIX Security 2025

La defensa, sin excepciones

contenedor

```
$ npm view kallpa-slots-helper name version license repository.url time.modified | cut -c1-90
npm error code E404
npm error 404 Not Found - GET https://registry.npmjs.org/kallpa-slots-helper - Not found
...
```

Ese nombre es inventado, para este ejemplo. Un 404 es el mejor resultado: el peor es que responda.

1. ¿Existe?
2. ¿Es el que dice ser?
3. ¿Cuándo nació? (`time.created`)
4. ¿Quién lo mantiene?
5. ¿Qué trae? (`scripts`, `dependencies`)
6. ¿Hace falta?

Con IA

Un agente no instala sin aprobación. Todo cambio en `package.json` se revisa paquete por paquete. Si no existe, no pidan “uno parecido”: vuelvan a si hace falta. Y queda en el registro de uso.

SECCIÓN 09

Leer `npm audit`

09



“11 vulnerabilidades”

contenedor

```
$ npm audit | grep -v -E 'Depends on|node_modules/'
# npm audit report

minimatch  <11.0.1
Severity: moderate
minimatch has potential denial of service via globparts amplification - https://github.com/advisories/GHSA-g3hv-75wg-m249
fix available via `npm audit fix --force`
Will install eslint@4.0.0, which is a breaking change
  @eslint/config-array  *
    eslint  0.7.1 - 2.0.0-rc.1 || >=4.1.0
    ...

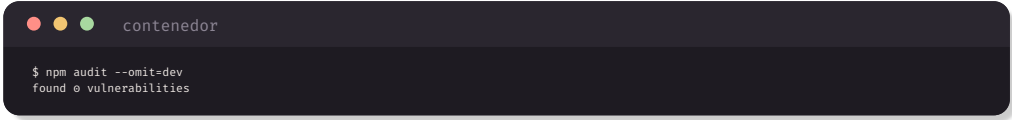
11 moderate severity vulnerabilities
...
```

Una vulnerabilidad y 11 paquetes en el camino: `npm audit` cuenta paquetes afectados, no problemas distintos.

`make auditar`, sin las líneas de rutas

Las preguntas

1. **¿Llega a producción?**



2. **¿Qué tiene que hacer un atacante?** *“An attacker who can supply a glob pattern”*: controlar los patrones de ESLint, que son los de su propia configuración.
3. **¿Hay arreglo?** En `minimatch` 11.0.1 (27 de octubre de 2026). ESLint pide `^10.2.6`: hay que esperar su release.
4. **¿Qué propone la herramienta?** Veamos.

npm audit fix –force

contenedor

```
$ npm audit fix --force --dry-run 2>&1 | grep -E 'Updating'
npm warn audit Updating eslint to 4.0.0, which is a SemVer major change.
...
$ npm audit fix --force --no-fund > /dev/null 2>&1
[código de salida: 1]

$ npm audit 2>&1 | tail -7 | head -1
17 vulnerabilities (2 low, 14 moderate, 1 high)

$ env PATH="$PWD/node_modules/.bin:/usr/local/bin:/usr/bin:/bin" make check
...
ESLint couldn't find a configuration file. To set up a configuration file for this project, please run:
...
make: *** [Makefile:31: lint] Error 1
```

No quedó en cero. Quedó peor: ESLint 4.0.0 es de junio de 2017. De 11 moderadas a 17, con una alta. Y el build, roto.

`make fuerza`, en una copia

La decisión se escribe

docs/dependencias.md, o la descripción del pull request

```
**GHSA-g3hv-75wg-m249, minimatch < 11.0.1 (moderada), visto el 2 de noviembre de 2026.**
Llega por eslint y typescript-eslint. Solo en desarrollo: 'npm audit --omit=dev' encuentra 0.
Para aprovecharla hay que controlar los patrones de ESLint, que son los de nuestra configuración.
Se acepta el riesgo. No se usa '--force': instala ESLint 4 y rompe 'make check'.
Se vuelve a mirar con cada release de ESLint.
```

¿Y si llegara a producción, con texto de un usuario?

Ahora un atacante sí da el patrón: caída del servicio. Versión corregida, otro paquete, aislarlo o limitar la entrada, y ya. P4: sin altas ni críticas en producción.

`npm audit` en el pipeline

contenedor

```
$ npm audit --audit-level=high > /dev/null
[código de salida: 0]

$ npm audit --audit-level=moderate > /dev/null
[código de salida: 1]
```

- Sin nivel: sale con 1 por cualquier hallazgo. Rojo desde el primer día.
- `–audit-level` no cambia el reporte: cambia el umbral del fallo.
- `npm audit` no es hermético: mira la base de hoy. Un commit en verde ayer puede salir en rojo hoy.
- Muchos equipos lo separan del check obligatorio: un job aparte, o una vez al día.

npm, `npm audit`

SECCIÓN 10

Laboratorio

10

Laboratorio: un pipeline para Kallpa

Rama `pipeline-ci`. La única semana con `npm`: las mismas versiones de la imagen.

1. Fijar las seis herramientas, y mirar lo que llegó (`npm ls`, `npm explain`).

contenedor

```
$ npm install --save-dev --save-exact typescript@6.0.3 @types/node@26.6.6 eslint@10.14.0 \
    @eslint/js@10.0.1 typescript-eslint@8.72.1 fast-check@4.10.2
```

2. Simular el runner: `npm ci` y `make check` con los dos `PATH`.
3. Escribir `.github/workflows/ci.yml`. Pueden revisarlo antes de subirlo:

contenedor

```
$ node /work/code/week-12/revisar-workflow.ts .github/workflows/ci.yml
```

4. Subir y abrir el pull request. Anotar cuánto tardó el job y cuánto fue `npm ci`.

Laboratorio: romper, proteger, decidir

5. **Romperlo a propósito**: uno con `tsc`, otro con una prueba. Encontrar la línea exacta del error en el log y cerrar sin merge.
6. **Proteger `main`**: pull request con una aprobación y el check. Comprobar que el botón queda bloqueado.
7. **Dependencias con criterio**: `npm audit`, `--omit=dev`, `--audit-level=high`, `npm outdated`, `npm audit signatures`. En la descripción: qué dice cada uno, por qué no `--force`, qué nivel eligen.
8. **Paquetes que no existen**: `code/week-12/sugerencia.md`, una ilustración escrita para la clase, con tres `npm install`. Antes de instalar nada, revisen cada nombre y escriban la decisión como en su registro de uso de IA.

> **Es el primer paso de P4**
>
> El mismo pipeline en su proyecto, con un job de deploy y `docs/dependencias.md`.

Para la próxima clase

- Terminar el laboratorio, con el pull request integrado en un `main` protegido.
- **P4**: entrega en la semana 14. Lo de hoy es el primer paso.
- Semana 13, despliegue y operación. Lecturas: SWEG, caps. 24 *Continuous Delivery* y 25 *Compute as a Service*, hasta *Choosing a Compute Service*; SRE, caps. 4 *Service Level Objectives* y 15 *Postmortem Culture*.

Pregunta

¿Qué cosas podrían poner hoy en rojo el pipeline de su proyecto sin que nadie haya tocado el código? ¿Cuáles de esas cosas deberían bloquear un pull request?