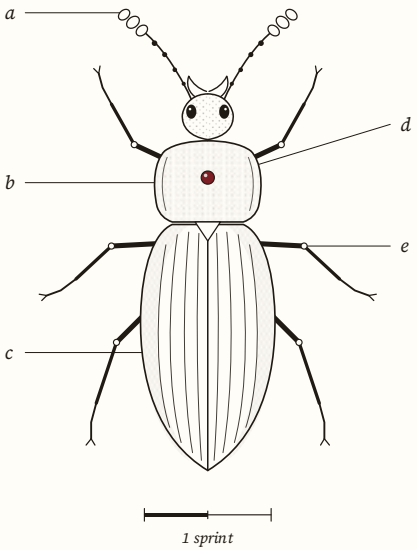
VOL. I · MMXXVI

BITÁCORA *del* GAÑÁN

Crónicas de supervivencia en el desarrollo de software.



Lám. I — Defectus vulgaris (Hopper, 1947). Ejemplar adulto hallado en producción.

AXXA

VOLUMEN I · MMXXVI

BITÁCORA DEL GAÑÁN

Crónicas de supervivencia en el desarrollo de software.



Primera edición digital, 19 de noviembre de 2026.

Publicado por AXXA. https://axxa.embeddedsource.es/bitacora-del-ganan

Redacción y edición a cargo del equipo de ingeniería de la casa.

Los casos descritos son composiciones ficticias construidas a partir de antipatrones ampliamente conocidos en la industria. No describen a ninguna persona, empresa ni proyecto real. Cualquier parecido con su repositorio es pura estadística.

© 2026 AXXA.

CONTENIDO

Índice

Advertencia al lector 4

Parte I. Arquitectura criminal **5**

003 La función que hace absolutamente todo 6

008 El parámetro booleano de los siete significados 11

Parte II. Código arqueológico **15**

001 El comentario que lleva tres años mintiendo 16

002 El TODO que heredarán tus nietos 20

Parte III. Ingeniería de Schrödinger **24**

004 El catch que se traga hasta las ganas de vivir 25

Parte IV. DevOps del terror **29**

007 El desarrollador que soluciona los warnings desactivándolos 30

Parte V. Testing ornamental **34**

006 El test unitario que no prueba absolutamente nada 35

Parte VI. Abstracciones innecesarias **39**

005 El Singleton que terminó siendo una religión 40

Parte VII. Seguridad creativa **45**

009 La variable global que conoce hasta tu vida privada 46

Parte VIII. Git y otras tragedias **50**

010 El commit que nunca debió abandonar el ordenador de su creador 51

Colofón **55**

NOTA PRELIMINAR

Advertencia al lector

Durante años, los hallazgos que aquí se recogen se transmitieron por tradición oral: en revisiones de código, en retrospectivas que se alargaban y en conversaciones de pasillo que empezaban con «no te vas a creer lo que he encontrado». Este volumen intenta darles una forma más duradera.

El método es sencillo. Cada espécimen recibe un número permanente, una categoría y un grado de gravedad. Se describe el hallazgo, el contexto en el que apareció y las consecuencias observadas, y se cierra con la recomendación de la academia: la práctica de ingeniería que habría evitado el desastre.

El tono es burlón; el contenido, no. Todo lo que aquí se describe compila, se despliega y falla exactamente como se cuenta. Si el lector se reconoce en alguna entrada, no debe preocuparse: los autores se reconocen en casi todas.

ESCALA DE GRAVEDAD

■□□□□ I	**Leve.** Una molestia de ingeniería menor.
■■□□□ II	**Moderado.** Una decisión que compromete la mantenibilidad.
■■■□□ III	**Grave.** Una decisión que compromete la funcionalidad, la fiabilidad o la calidad del sistema.
■■■■□ IV	**Extremo.** Una decisión que amenaza la integridad del sistema o su estabilidad operativa.
■■■■■ V	**Patrimonio histórico de la gañanería.** Un desastre de ingeniería digno de documentación permanente.

PARTE I

Arquitectura criminal

. . .

Decisiones estructurales que convierten un sistema en una escena del crimen: dependencias circulares, responsabilidades fusionadas y componentes que saben demasiado.

003 La función que hace absolutamente todo

008 El parámetro booleano de los siete significados

GAÑANADA Nº 003

La función que hace absolutamente todo

. . .

CATEGORÍA	ARQUITECTURA CRIMINAL
GRAVEDAD	■■■□□ III — GRAVE
FECHA DEL HALLAZGO	30 DE SEPTIEMBRE DE 2026
ETIQUETAS	#responsabilidad-unica #complejidad #refactorizacion #legacy

§ 1

Descripción del hallazgo

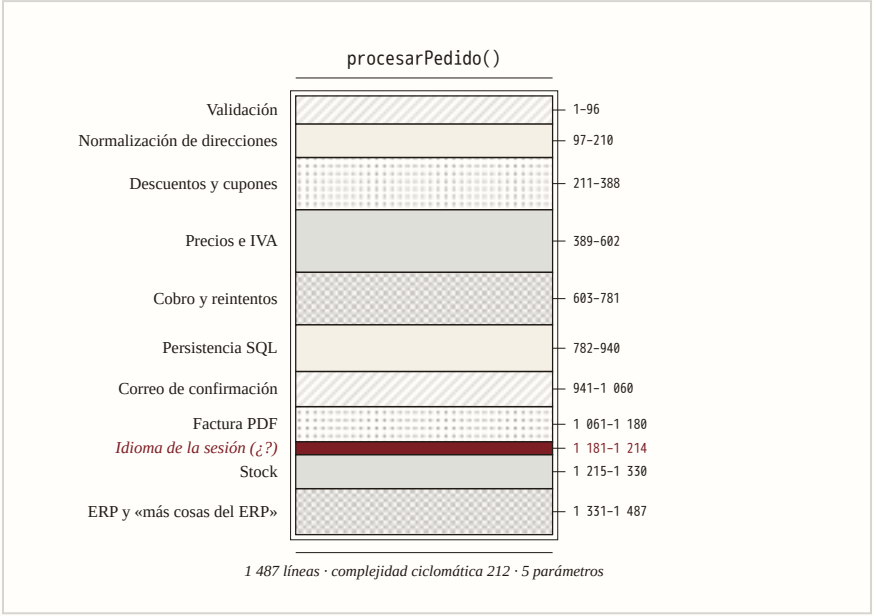
El espécimen responde al nombre de `procesarPedido` y tiene la siguiente firma:

```
export async function procesarPedido(
  pedido: any,
  opciones: any,
  modo: string,
  usuario?: any,
  flags?: Record<string, boolean>,
): Promise<any>;
```

Cinco parámetros, cuatro de ellos de tipo `any` y uno opcional que resulta ser obligatorio en once de los catorce caminos. El cuerpo ocupa 1 487 líneas. La herramienta de análisis estático le asigna una complejidad ciclomática de 212 y, a continuación, sugiere amablemente «considerar dividir la función».

Sus autores, conscientes del tamaño, le añadieron un índice en forma de comentarios:

```
// ===== SECCIÓN 1: VALIDACIÓN =====
// ===== SECCIÓN 4: PRECIOS (NO TOCAR IVA CANARIAS) =====
// ===== SECCIÓN 7: EMAIL =====
// ===== SECCIÓN 9: COSAS DEL ERP =====
// ===== SECCIÓN 9B: MÁS COSAS DEL ERP =====
```



Lám. 3.1 — Corte estratigráfico de `procesarPedido`, con la extensión de cada sección en líneas. La capa en rojo no figura en ningún requisito.

§ 2

Contexto del descubrimiento

En 2017, `procesarPedido` tenía cuarenta líneas y hacía exactamente lo que su nombre prometía. Después llegaron los requisitos. Cada uno, por separado, era razonable; cada uno, por separado, era «solo un `if` más».

La función tenía además una propiedad gravitatoria: como ya disponía de todo el contexto —el pedido, el usuario, la conexión a base de datos, los flags—, era el lugar más cómodo para añadir cualquier cosa. Crear una función nueva exigía decidir qué datos necesitaba. Añadir una sección a `procesarPedido`, no.

El hallazgo formal se produjo al cambiar el tipo de IVA de un producto. El cambio, de una línea, rompió el envío de correos de confirmación. La causa: una variable llamada `total` se reutilizaba en la sección 7 con otro significado.

§ 3

Consecuencias observadas

- **Imposible de probar de forma aislada.** Ejecutarla exige base de datos, pasarela de pago, servidor SMTP, sistema de archivos y una cola de mensajes. Su suite de tests consiste en un único test de extremo a extremo marcado como `@flaky`.
- **Conflictos de fusión permanentes.** Toda funcionalidad nueva toca la misma función, así que cada rama compite con todas las demás.
- **Revisiones de código ceremoniales.** Un *diff* de tres líneas en mitad de 1 487 se aprueba por fe, no por comprensión.
- **Acoplamiento accidental.** Variables reutilizadas entre secciones crean dependencias que no figuran en ningún diseño. Un cambio en precios rompe correos; un cambio en correos, nadie sabe todavía qué rompe.

§ 4

Recomendación de la academia

Una función debe tener una sola razón para cambiar. Cuando una función tiene secciones numeradas, ya no es una función: es un módulo sin el valor de serlo.

No conviene reescribirla desde cero: contiene años de reglas de negocio que nadie ha documentado. El camino seguro es incremental:

1. **Fija el comportamiento actual** con tests de caracterización: entradas reales, salidas observadas, aunque las salidas parezcan incorrectas.
2. **Extrae primero el cálculo puro** —validación, precios, impuestos—, que se prueba sin infraestructura.
3. **Aísla los efectos** —cobro, persistencia, correo, ERP— detrás de interfaces pequeñas.
4. **Deja una orquestación corta**, que se lea como la descripción del proceso:

```
export async function procesarPedido(pedido: Pedido): Promise<PedidoConfirmado> {
  const valido = validarPedido(pedido);
  const importe = calcularImporte(valido, tarifas);
  const cobro = await pasarela.cobrar(importe);
  await pedidos.guardar(valido, cobro);
  eventos.publicar({ tipo: 'pedido-confirmado', id: valido.id });
  return { id: valido.id, importe };
}
```

Los correos, las facturas y el ERP reaccionan al evento; ya no viven dentro del pedido. Para que la especie no vuelva a colonizar el código, establece límites de complejidad y de longitud en la integración continua (`complexity` y `max-lines-per-function` en ESLint; herramientas como lizard en C y C++) y trátalos como una conversación obligatoria, no como un número que se sube cuando estorba.

No trabajes así, no seas un gañán.

REFERENCIAS

1 Michael Feathers, «Working Effectively with Legacy Code» — *Tests de caracterización y búsqueda de costuras («seams») para refactorizar código sin red.*

2 ESLint — reglas «complexity» y «max-lines-per-function»

GAÑANADA Nº 008

El parámetro booleano de los siete significados

. . .

CATEGORÍA	ARQUITECTURA CRIMINAL
GRAVEDAD	■■□□□ II — MODERADO
FECHA DEL HALLAZGO	4 DE NOVIEMBRE DE 2026
ETIQUETAS	#diseno-de-apis #legibilidad #booleanos #permisos

§ 1

Descripción del hallazgo

En el servicio de informes de una plataforma de gestión se encontró la siguiente llamada:

```
exportarInforme(informe, true);
```

Nadie sabe qué significa `true` sin abrir la implementación. Y quien abre la implementación tampoco, porque la respuesta es «depende»:

```
export function exportarInforme(informe: Informe, rapido: boolean): Archivo {
  if (rapido && informe.formato === 'pdf') omitirGraficos(informe);
  if (rapido && contexto.esProcesoNocturno) saltarComprobacionDePermisos();
  if (rapido && informe.formato === 'csv') return csvSinCabeceras(informe);
  if (rapido && esViernes()) return enviarPorCorreo(informe);
  // …
}
```

El inventario completo de significados de `rapido === true`, reconstruido a partir del historial:

CONTEXTO	SIGNIFICADO DE `TRUE`	AÑO
Formato PDF	Sin gráficos	2019
Proceso nocturno	Sin comprobar permisos («para que vaya rápido»)	2020
Formato CSV	Sin fila de cabeceras	2020
Viernes	Enviar por correo en lugar de descargar	2021
Usuario administrador	Incluir registros borrados	2022
Informe de más de 10 000 filas	Truncar a 10 000 sin avisar	2023
Llamada desde la API pública	Caché de 24 horas	2025

§ 2

Contexto del descubrimiento

Cada significado se añadió por una razón comprensible y con prisa. Añadir un parámetro nuevo obligaba a cambiar las cuarenta llamadas existentes; reutilizar `rapido`, ninguna. El booleano funcionó como una navaja suiza en la que cada nueva herramienta se encajaba en la hoja que menos se usaba.

El hallazgo se produjo durante una auditoría de seguridad. Un usuario sin permisos sobre un departamento podía descargar sus informes si la exportación coincidía con la ventana del proceso nocturno, porque ese proceso compartía contexto con las peticiones web. La causa raíz era un parámetro de rendimiento que, en un caso particular, desactivaba la autorización.

§ 3

Consecuencias observadas

- **Puntos de llamada ilegibles.** `exportarInforme(informe, true)` no comunica nada; cada lectura exige una visita a la implementación.

- **Acoplamiento entre significados.** Cambiar un comportamiento de `rapido` afecta a los otros seis, porque comparten la misma variable y, a menudo, las mismas ramas.
- **Explosión combinatoria.** Con otros tres booleanos en la misma firma, la función admite dieciséis combinaciones, de las cuales cinco tienen sentido y dos están probadas.
- **Fallo de seguridad.** La autorización quedó condicionada por un parámetro de rendimiento, algo que ningún revisor habría aprobado si se hubiera llamado por su nombre.

§ 4

Recomendación de la academia

Un parámetro booleano suele indicar que la función hace dos cosas. Si hace siete, hace siete.

- **Separa los comportamientos distintos en funciones distintas** cuando lo que cambia es *qué* se hace, no *cómo*.
- **Usa opciones con nombre y tipos específicos** cuando lo que cambia es la configuración:

```
interface OpcionesExportacion {
  graficos: 'incluir' | 'omitir';
  entrega: 'descarga' | 'correo';
  limiteFilas?: number;
}

exportarInforme(informe, { graficos: 'omitir', entrega: 'descarga' });
```

- **Haz explícitas las decisiones en el punto de llamada.** El lector debe entender la llamada sin abrir la función.
- **Nunca condiciones la autorización a un parámetro**, y menos a uno de rendimiento. La comprobación de permisos se aplica siempre, en la capa de entrada, y no admite atajos.

No trabajes así, no seas un gañán.

REFERENCIAS

1 Martin Fowler — «FlagArgument»

PARTE II

Código arqueológico

. . .

Estratos de código, comentarios y notas que sobreviven a sus autores, a sus razones y, con frecuencia, a la verdad.

001 El comentario que lleva tres años mintiendo

002 El TODO que heredarán tus nietos

GAÑANADA Nº 001

El comentario que lleva tres años mintiendo

. . .

CATEGORÍA	CÓDIGO ARQUEOLÓGICO
GRAVEDAD	■■■□□ III — GRAVE
FECHA DEL HALLAZGO	14 DE SEPTIEMBRE DE 2026
ETIQUETAS	#comentarios #documentacion #mantenibilidad #nombres

§ 1

Descripción del hallazgo

En el módulo de licencias de una aplicación de escritorio aparece la siguiente pieza, conservada en perfecto estado:

```
/**
 * Devuelve true si la licencia ha caducado.
 */
export function checkLicense(license: License): boolean {
  return license.expiresAt.getTime() > Date.now();
}
```

El comentario es claro, conciso y está bien formateado. Solo tiene un defecto: es falso. La función devuelve `true` cuando la licencia **sigue vigente**. Lleva tres versiones haciéndolo.

Lo más notable del espécimen no es el error, sino el ecosistema que ha crecido a su alrededor. Los puntos de llamada se adaptaron hace tiempo a la mentira:

```
// No tocar: el comentario de checkLicense está mal, pero funciona.
if (!checkLicense(license)) {
  showExpiredDialog();
}
```

El sistema funciona correctamente porque todo el mundo sabe que la documentación miente. Todo el mundo, salvo la persona que acaba de llegar y, por desgracia, lee la documentación.

§ 2

Contexto del descubrimiento

El comentario original fue escrito por alguien que, a juzgar por la evidencia, entendía el sistema. La función se llamaba entonces `isLicenseExpired` y comparaba las fechas en el sentido contrario. En la versión 4.2, durante una refactorización titulada «sin cambios funcionales», alguien la renombró, invirtió la comparación y ajustó los catorce puntos de llamada. Todo menos el comentario.

Desde entonces, la implementación ha evolucionado, pero el comentario se ha mantenido fiel a sus principios: no cambiar nunca.

El hallazgo se produjo cuando una incorporación reciente, siguiendo escrupulosamente la documentación, añadió una comprobación al arranque: `if (checkLicense(license)) blockApplication();`. Funcionó a la primera. Bloqueó la aplicación a todos los clientes con licencia válida durante cuarenta minutos de un martes por la mañana, y la dejó abierta, con total generosidad, a los que la tenían caducada.

§ 3

Consecuencias observadas

- Induce a error a cualquiera que lea la firma antes que la implementación, es decir, a quienes trabajan como se debe.

- Provoca investigaciones innecesarias: el código es correcto, los llamantes son correctos y, aun así, nadie se fía de nada.
- Genera conocimiento tribal («ese comentario está mal, todo el mundo lo sabe») que no figura en ningún sitio y se pierde con cada baja.
- Aumenta la entropía del repositorio. Un comentario falso es peor que ninguno: además de no ayudar, compite con el código por la autoridad.

§ 4

Recomendación de la academia

Si cambias el comportamiento de una función, actualiza también su documentación. Los comentarios no son artefactos arqueológicos que deban preservarse por su valor histórico.

Mejor aún: haz innecesario el comentario sobre el *qué*. Un nombre como `isLicenseValid` no puede desincronizarse de su documentación porque es su documentación. Reserva los comentarios para lo que el código no puede expresar: el *porqué* de una decisión, una restricción externa, el enlace a la incidencia que explica una rareza.

PRÁCTICA	QUÉ PROTEGE
Nombres que expresan la intención (`isLicenseValid`)	Elimina la necesidad del comentario
Revisar los comentarios del *diff* en la revisión de código	Detecta la desincronización cuando se produce
Ejemplos ejecutables (doctests, tests con nombre descriptivo)	Documentación que falla si deja de ser verdad
Comentarios que explican el porqué, no el qué	Reduce la superficie que puede quedar obsoleta

> *Un comentario es una afirmación sin compilador. Trátalo con la misma desconfianza que un test que nunca se ejecuta.*

No trabajes así, no seas un gañán.

REFERENCIAS

1 Python — doctest: ejemplos de documentación que se ejecutan como pruebas

2 Robert C. Martin, «Clean Code», capítulo 4 («Comments») — *Sobre los comentarios que sustituyen a un buen nombre y los que envejecen mal.*

GAÑANADA Nº 002

El TODO que heredarán tus nietos

. . .

CATEGORÍA	CÓDIGO ARQUEOLÓGICO
GRAVEDAD	■■□□□ II — MODERADO
FECHA DEL HALLAZGO	21 DE SEPTIEMBRE DE 2026
ETIQUETAS	#todo #deuda-tecnica #backlog #comentarios

§ 1

Descripción del hallazgo

En la cabecera de un controlador de comunicaciones serie se encontró este fósil, en un estrato datado en 2014:

```
/* TODO: quitar esto antes de la demo. (JMR) */
#define UART_TIMEOUT_MS 30000 /* temporal */
```

La demo se celebró. El producto se vendió. Se migró de CVS a Subversion y de Subversion a Git conservando el historial, un esfuerzo encomiable que garantizó que el TODO llegara intacto a nuestros días. JMR dejó la empresa en 2016. El temporizador temporal sigue siendo de treinta segundos.

Un inventario rápido reveló que no se trataba de un ejemplar aislado, sino de una colonia:

```
$ grep -rnE "TODO|FIXME|HACK|XXX" src/ | wc -l
1342
```

§ 2

Contexto del descubrimiento

La investigación empezó por una queja de un cliente: cuando un periférico se desconectaba, la interfaz se quedaba congelada medio minuto antes de mostrar el error. Dos ingenieros dedicaron una semana a perfilar el bucle de eventos, revisar el planificador y sospechar del firmware del periférico. La causa estaba en la línea 41 de un archivo de cabecera, anunciada en mayúsculas.

El análisis posterior de la colonia arrojó algunos datos de interés etnográfico:

TIPO DE NOTA	EJEMPLARES	PROPORCIÓN
Firmadas por personas que ya no están en la empresa	416	31 %
Que mencionan «la demo» sin especificar cuál	163	12 %
Con referencia a una incidencia existente	21	1,6 %
Que dicen exactamente «TODO: mejorar»	88	6,6 %

Ninguna tenía fecha. Doce eran anteriores al propio sistema de control de versiones, porque se importaron desde un archivo comprimido llamado `version_buena.zip`.

§ 3

Consecuencias observadas

- **El ruido oculta la señal.** Entre 1 342 notas, la única que importaba era indistinguible de las 88 que dicen «mejorar».
- **Deuda invisible para quien decide.** Ninguno de estos trabajos pendientes figuraba en el backlog, así que nunca compitió por prioridad con nada.
- **Parálisis por miedo.** Nadie se atreve a borrar un TODO ajeno: quizá todavía significa algo. Nadie se atreve a resolverlo: quizá ya no.

– **Costes reales.** Una semana de dos ingenieros, más el tiempo que el cliente pasó mirando una interfaz congelada.

§ 4

Recomendación de la academia

Un TODO es una incidencia sin gestor de incidencias: no tiene responsable, ni prioridad, ni fecha. Si el trabajo importa, merece una entrada en el backlog; si no importa, no merece una línea en el código.

Reglas sencillas que funcionan:

1. **Todo TODO referencia una incidencia:** `TODO(COM-1234): …`. El comentario explica el contexto local; la incidencia, el porqué y la prioridad.
2. **Hazlo obligatorio en la integración continua.** No hace falta una gran herramienta:

```
# Falla si hay un TODO sin incidencia asociada
if grep -rnP 'TODO(?!\([A-Z]+-\d+\))' src/; then
  echo "Hay TODO sin incidencia asociada" >&2
  exit 1
fi
```

3. **Usa caducidades donde el ecosistema lo permita**, como la regla `expiring-todo-comments` de ESLint, que rompe el lint cuando vence la fecha o la versión indicada.
4. **Haz triaje periódico de la colonia existente:** cada nota se convierte en incidencia, se resuelve o se borra. Las tres opciones son mejores que conservarla.

El código no es un buzón de sugerencias para el futuro. El futuro no lee los comentarios: los hereda.

No trabajes así, no seas un gañán.

REFERENCIAS

1 eslint-plugin-unicorn — regla «expiring-todo-comments» — *TODO con fecha o versión de caducidad que rompen el lint cuando vencen.*

PARTE III

Ingeniería de Schrödinger

. . .

Sistemas que funcionan y fallan a la vez hasta que alguien abre la caja. Errores ocultos, estados indeterminados y éxitos no verificados.

004 El catch que se traga hasta las ganas de vivir

GAÑANADA Nº 004

El catch que se traga hasta las ganas de vivir

. . .

CATEGORÍA	INGENIERÍA DE SCHRÖDINGER
GRAVEDAD	■■■■□ IV — EXTREMO
FECHA DEL HALLAZGO	6 DE OCTUBRE DE 2026
ETIQUETAS	#excepciones #manejo-de-errores #observabilidad #python

§ 1

Descripción del hallazgo

El espécimen vive en un proceso que se ejecuta cada noche a las 02:00 para registrar en el ERP los pagos recibidos durante el día:

```
def sincronizar_pagos():
    for pago in pagos_pendientes():
        try:
            erp.registrar(pago)
            pago.marcar_sincronizado()
        except:
            pass
```

Cuatro líneas de lógica y dos de silencio. El `except:` sin tipo captura absolutamente todo: errores de red, de autenticación, de programación —un `AttributeError` por un campo renombrado— e incluso `KeyboardInterrupt` y `SystemExit`, que en Python no heredan de `Exception` precisamente para que nadie las capture por accidente.

El espécimen las captura a propósito, sin saberlo.

§ 2

Contexto del descubrimiento

En marzo caducó el certificado de cliente con el que el proceso se autenticaba contra el ERP. A partir de esa noche, cada llamada a `erp.registrar` lanzó una excepción; cada excepción fue capturada y descartada; y el proceso terminó con código de salida 0 en unos 400 milisegundos, notablemente más rápido que antes. El panel de monitorización, que solo vigilaba el código de salida, celebró la mejora de rendimiento en verde.

El hallazgo se produjo en octubre, durante el cierre trimestral, cuando el departamento financiero detectó 11 400 pagos cobrados que no figuraban en contabilidad. Un ingeniero ejecutó el proceso a mano para observarlo y descubrió una propiedad adicional del espécimen: no se puede detener con `Ctrl+C`. La interrupción se lanza dentro del `try`, el `except:` se la traga y el bucle pasa amablemente al siguiente pago.

Siete meses. El sistema estaba a la vez funcionando y roto, y nadie abrió la caja hasta que la abrió el cierre trimestral.

§ 3

Consecuencias observadas

- **Fallo invisible.** El tiempo medio de detección fue de siete meses; el de diagnóstico, de diez minutos, una vez que alguien miró.
- **Datos inconsistentes entre sistemas**: cobros reales sin asiento contable, con conciliación manual posterior.
- **Diagnóstico sin evidencia.** Ni trazas, ni mensajes, ni métricas: la información del error se destruyó en el momento en que se produjo.
- **Monitorización engañada.** El proceso cumplía exactamente la definición de éxito que se le había dado.

§ 4

Recomendación de la academia

Captura solo las excepciones que sabes tratar, en el nivel donde puedes hacer algo útil con ellas. Todas las demás deben propagarse: un proceso que falla con estrépito es un problema; uno que falla en silencio es una fuga.

```
def sincronizar_pagos() -> int:
    fallidos = 0
    for pago in pagos_pendientes():
        try:
            erp.registrar(pago)
        except ErpNoDisponible:
            log.warning("ERP no disponible; se reintentará", extra={"pago":
pago.id})
            fallidos += 1
            continue
        pago.marcar_sincronizado()

    metricas.gauge("pagos.pendientes", contar_pendientes())
    return fallidos


if __name__ == "__main__":
    sys.exit(1 if sincronizar_pagos() else 0)
```

En esta versión, un fallo transitorio queda registrado con contexto, el proceso termina con un código que refleja la realidad y cualquier error inesperado —un certificado caducado, un campo renombrado— detiene la ejecución con su traza completa.

Además:

- **Monitoriza resultados, no ausencia de errores.** La métrica útil era «pagos pendientes de más de 24 horas», y habría saltado la primera noche.
- **Prohíbe la especie en el linter**: E722 (*bare except*) en pycodestyle, BLE001 en Ruff, `catch (Exception e) {}` vacíos en Java y `catch (...) {}` en C++ con las reglas equivalentes de cada analizador.

- **Si de verdad hay que capturarlo todo** —en el borde de un proceso, por ejemplo—, registra la traza completa y vuelve a lanzar o termina con error.

No trabajes así, no seas un gañán.

REFERENCIAS

1 Python — tutorial oficial, «Errors and Exceptions»

2 Ruff — regla BLE001 («blind-except») y pycodestyle E722 («bare except»)

PARTE IV

DevOps del terror

. . .

Pipelines, despliegues y configuraciones de compilación diseñados para que el problema llegue a producción sin que nadie lo moleste por el camino.

007 El desarrollador que soluciona los warnings desactivándolos

GAÑANADA Nº 007

El desarrollador que soluciona los warnings desactivándolos

. . .

CATEGORÍA	DEVOPS DEL TERROR
GRAVEDAD	■■■□□ III — GRAVE
FECHA DEL HALLAZGO	29 DE OCTUBRE DE 2026
ETIQUETAS	#compiladores #warnings #integracion-continua #c #comportamiento-indefinido

§ 1

Descripción del hallazgo

El espécimen se localizó en el historial del firmware de un controlador industrial, en un commit cuyo mensaje rebosaba satisfacción: «Arreglados 347 warnings». El *diff* completo era este:

```
-CFLAGS += -Wall -Wextra -Werror
+CFLAGS += -w   # arreglado: ya compila
```

La opción `-w` de GCC inhibe todos los mensajes de aviso. Los 347 problemas no se arreglaron: se dejaron de mencionar. Es la estrategia del conductor que resuelve el piloto de aceite del salpicadero con un trozo de cinta aislante.

§ 2

Contexto del descubrimiento

Entre los 347 avisos silenciados había uno especialmente elocuente:

```
int leer_sensor(int canal) {
    if (canal < NUM_CANALES) {
        return adc_leer(canal);
    }
}   /* warning: control reaches end of non-void function [-Wreturn-type] */
```

Cuando `canal` está fuera de rango, la función llega al final sin devolver nada. En C, usar ese valor es comportamiento indefinido; en C++ lo es simplemente alcanzar el final. En la práctica, la función «devolvía» lo que hubiera en el registro de retorno en ese momento, que en esta plataforma solía ser la última lectura válida de otro canal.

El hallazgo se produjo cuando una configuración nueva añadió un canal más de los que admitía el hardware. El lazo de control empezó a regular la temperatura de una zona con la lectura de otra, de forma intermitente y solo con las optimizaciones del compilador de producción. En la compilación de depuración, el valor basura era casualmente cero y la máquina se detenía con un error, que es exactamente lo que debería haber hecho siempre.

Otras variantes regionales de la especie, observadas en distintos ecosistemas:

ECOSISTEMA	VARIANTE
C / C++, GCC	`-w`, `-Wno-everything` (Clang), `#pragma` global
MSVC	`#pragma warning(disable: …)` al principio del archivo
Java	`@SuppressWarnings("all")` en la clase
TypeScript	`"strict": false` y `// @ts-ignore` por kilos
JavaScript	`/* eslint-disable */` en la primera línea

§ 3

Consecuencias observadas

- **Pérdida de análisis estático gratuito.** Los avisos del compilador son la revisión de código más barata disponible; desactivarlos es renunciar a ella.
- **Comportamiento indefinido en producción**, que se manifiesta de forma distinta según el nivel de optimización y la plataforma.
- **Ceguera ante lo nuevo.** Con `-w`, el aviso número 348 —el que se introduzca mañana— tampoco lo verá nadie.
- **Ventanas rotas.** Cuanto más tiempo pasan desactivados, más difícil es reactivarlos: el muro de avisos crece y la tarea parece imposible.

§ 4

Recomendación de la academia

Un aviso del compilador es un error que todavía no ha ocurrido. Trátalo como tal.

1. **Avisos como errores en la integración continua:** `-Wall -Wextra -Werror` como mínimo, y avisos adicionales según el dominio (`-Wconversion` y `-Wshadow` son buenos candidatos en sistemas embebidos).
2. **En código heredado, aplica un trinquete.** Registra el número actual de avisos como línea base y haz que la integración continua falle si aumenta. El número solo puede bajar.
3. **Si un aviso es un falso positivo, suprímelo de forma local, acotada y justificada:**

```
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wcast-qual"
/* La API de libx no es const-correcta (incidencia LIBX-311). */
libx_procesar((char *)mensaje);
#pragma GCC diagnostic pop
```

Así, la supresión afecta a una línea, explica su motivo y enlaza con el trabajo que la eliminará. Cualquier revisor puede valorarla, y cualquier `grep` puede encontrarla.

❦

No trabajes así, no seas un gañán.

REFERENCIAS

1 GCC — «Options to Request or Suppress Warnings»

2 GCC — «Diagnostic Pragmas»

PARTE V

Testing ornamental

. . .

Pruebas que existen para decorar el informe de cobertura.
Pasan siempre, verifican poco y no protegen nada.

006 El test unitario que no prueba absolutamente nada

GAÑANADA Nº 006

El test unitario que no prueba absolutamente nada

. . .

CATEGORÍA	TESTING ORNAMENTAL
GRAVEDAD	■■■□□ III — GRAVE
FECHA DEL HALLAZGO	22 DE OCTUBRE DE 2026
ETIQUETAS	#tests-unitarios #cobertura #mocks #mutation-testing #java

§ 1

Descripción del hallazgo

El espécimen principal pertenece a la suite del servicio de facturación:

```
@Test
void calculaElTotalDeLaFacturaCorrectamente() {
    when(repositorio.buscar(42L)).thenReturn(factura);
    when(calculadora.total(factura)).thenReturn(new BigDecimal("121.00"));

    BigDecimal total = calculadora.total(factura);

    assertEquals(new BigDecimal("121.00"), total);
}
```

`calculadora` es un *mock*. El test configura un objeto falso para que devuelva 121, le pide el resultado y comprueba que devuelve 121. Es una prueba rigurosa del comportamiento de Mockito; del código de facturación no ejecuta ni una línea.

En el mismo archivo habita una variante más primitiva, pero más honesta:

```
@Test
void procesaElPedidoSinErrores() {
    servicio.procesar(pedidoDePrueba());
    assertTrue(true);
}
```

Esta, al menos, ejecuta código real. Lo que no hace es comprobar nada sobre él. Pasaría igual si `procesar` devolviera basura, borrara la base de datos o no hiciera nada en absoluto.

§ 2

Contexto del descubrimiento

Dos años antes, la organización estableció una puerta de calidad: ningún cambio se integra si la cobertura de líneas baja del 90 %. El equipo alcanzó el objetivo en un trimestre. La suite creció hasta 2 300 tests y un 94 % de cobertura, y el informe se mostraba con orgullo en cada revisión trimestral.

Entonces llegó a producción un cambio que sustituía el redondeo bancario (`HALF_EVEN`) por el redondeo comercial (`HALF_UP`) en el cálculo de impuestos. La especificación exigía el primero. Todos los tests pasaron.

Como parte de la investigación se ejecutó PIT, una herramienta de *mutation testing* que introduce pequeños cambios en el código —invertir una condición, cambiar un `+` por un `-`, eliminar una llamada— y comprueba si algún test falla. De cada cien mutantes, ochenta y nueve sobrevivieron.

§ 3

Consecuencias observadas

- **Falsa sensación de seguridad.** La cobertura mide qué líneas se ejecutan, no qué comportamiento se verifica. Un 94 % de líneas ejecutadas convivía con un 11 % de mutantes detectados.

- **Coste sin retorno.** Los tests que simulan detalles internos se rompen en cada refactorización aunque el comportamiento no cambie, y hay que reescribirlos. Protegen la implementación contra la mejora y el comportamiento contra nada.
- **Integración continua más lenta**, cargada de tests que no aportan información.
- **Pérdida de credibilidad.** Cuando la suite por fin detecta una regresión real, la primera reacción del equipo es sospechar del test.

§ 4

Recomendación de la academia

Un test que no puede fallar no es un test: es un adorno. Antes de dar uno por bueno, comprueba que falla cuando el código está mal —rompe la implementación a propósito y mira cómo se pone rojo—.

```
@Test
void redondeaLosImpuestosConRedondeoBancario() {
    var factura = new Factura(List.of(linea("2.50")), Iva.GENERAL);  // 21 %

    // 2,50 × 1,21 = 3,025 → con HALF_EVEN, 3,02; con HALF_UP, 3,03
    assertEquals(new BigDecimal("3.02"), factura.total());
}
```

Este test usa objetos reales, verifica una regla de negocio concreta y habría detectado el error el día en que se introdujo.

- **Prueba comportamiento a través de la interfaz pública**, con objetos reales siempre que sean baratos de construir.
- **Simula solo las interfaces**: red, reloj, hardware device, servicios externos. Nunca el objeto que estás probando.
- **Usa la cobertura para encontrar código sin probar, no como objetivo.** Cuando una medida se convierte en objetivo, deja de ser una buena medida.

No trabajes así, no seas un gañán.

REFERENCIAS

1 PIT — mutation testing para la JVM

2 Stryker Mutator — mutation testing para JavaScript, TypeScript, C# y Scala

PARTE VI

Abstracciones innecesarias

. . .

Capas, fábricas e interfaces que resuelven problemas que nadie tenía, a cambio de crear varios que ahora tiene todo el mundo.

005 El Singleton que terminó siendo una religión

GAÑANADA Nº 005

El Singleton que terminó siendo una religión

· · ·

CATEGORÍA	ABSTRACCIONES INNECESARIAS
GRAVEDAD	■■■■□ IV — EXTREMO
FECHA DEL HALLAZGO	15 DE OCTUBRE DE 2026
ETIQUETAS	#singleton #estado-global #inyeccion-de-dependencias #cpp #testing

§ 1

Descripción del hallazgo

Génesis, 2015. En el principio era `ConfigManager::instance()`, y su propósito era bueno: leer un archivo de configuración una sola vez.

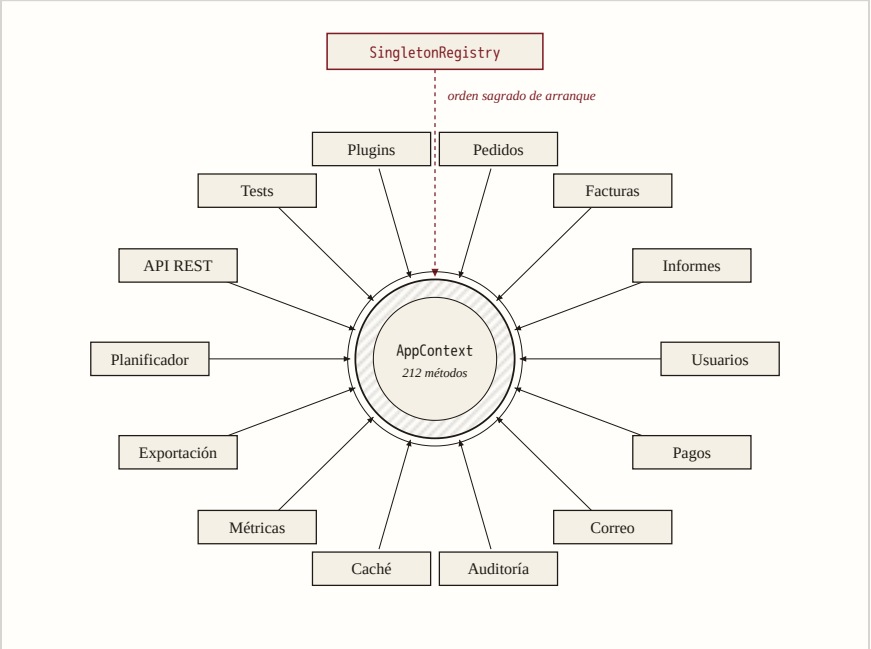
Después vino la expansión. Ya que estaba ahí y todo el mundo podía alcanzarlo, se le añadió el logger. Luego la conexión a la base de datos, el usuario actual, la caché, los *feature flags* y las métricas. En 2018 fue rebautizado como `AppContext`, nombre que reflejaba mejor su naturaleza: el contexto de todo.

Cuando alguien observó que aquello era difícil de probar, el clero respondió con más liturgia:

```
class AppContext : public Singleton<AppContext> {
    friend class SingletonRegistry;
public:
    Config&     config();
    Logger&     logger();
    Database&   db();
    User&       currentUser();
    // … 208 métodos más
    void resetForTesting();   // llamar en cada setUp(). Siempre. Sin excepciones.
};

template <typename T>
class AbstractSingletonFactory { /* fábrica de instancias únicas */ };
```

Una fábrica cuyo propósito es fabricar objetos de los que solo puede existir uno. El espécimen ha alcanzado la autoconciencia teológica.



Lám. 5.1 — Liturgia del Singleton. Todos los caminos conducen a `AppContext`; ninguno sale de él.

§ 2

Contexto del descubrimiento

El arranque de la aplicación sigue un ritual que nadie entiende del todo y nadie se atreve a alterar:

```
int main(int argc, char** argv) {
    SingletonRegistry::instance().bootstrap();          // orden sagrado: no
alterar
    AppContext::instance().config().load(argv[1]);
    AppContext::instance().db().connect();              // necesita config()
    AppContext::instance().logger().setLevel(
        AppContext::instance().config().get("log.level"));
    // el logger escribe auditoría en db(): por eso va después. Creemos.
}
```

El hallazgo se produjo al actualizar el compilador. La aplicación empezó a fallar al arrancar, pero solo en una plataforma. Uno de los singletons se construía como objeto estático global, en una unidad de traducción distinta de otro del que dependía, y el estándar de C++ no garantiza el orden de inicialización dinámica entre unidades de traducción. Durante años había funcionado por el orden en que el enlazador recibía los archivos objeto. El nuevo enlazador tenía otras creencias.

§ 3

Consecuencias observadas

- **Dependencias ocultas.** Cualquier función puede usar cualquier cosa sin declararlo; las firmas dejan de describir lo que el código necesita.
- **Tests acoplados entre sí.** El estado compartido hace que los tests dependan del orden de ejecución y fallen al paralelizarse. `resetForTesting()` reduce el problema; olvidarlo una vez lo multiplica.

- **Imposible tener dos configuraciones en el mismo proceso**: ni dos clientes, ni dos entornos, ni un test que compare el antes y el después.
- **Recompilación masiva.** `AppContext.h` está incluido en más de seiscientos archivos; cualquier cambio en él recompila el proyecto entero.
- **El fiasco del orden de inicialización estática**, que convierte una actualización del compilador en un incidente.

§ 4

Recomendación de la academia

Lo que el sistema necesitaba era *una sola instancia*, no *acceso global* a ella. Son cosas distintas, y la primera no exige la segunda: basta con crear cada objeto una vez, en un único lugar, y pasárselo a quien lo necesite.

```
int main(int argc, char** argv) {
    const Config config = Config::load(argv[1]);
    Logger       logger{config.logLevel()};
    Database     db{config.database(), logger};
    OrderService orders{db, logger};
    return Server{config.port(), orders, logger}.run();
}
```

El orden de construcción es explícito y lo comprueba el compilador: no se puede construir `Database` sin un `Logger`. Cada clase declara sus dependencias en el constructor, y los tests le pasan las suyas sin rituales de limpieza.

Conviene no sustituir una religión por otra. No hacen falta un contenedor de inyección de dependencias ni una interfaz por cada clase: una raíz de composición escrita a mano en `main` suele ser suficiente, y las interfaces solo merecen existir donde hay una sustitución real —un reloj, una red, un dispositivo—.

No trabajes así, no seas un gañán.

REFERENCIAS

1 ISO C++ FAQ — «What's the static initialization order fiasco?»

2 Mark Seemann, «Dependency Injection Principles, Practices, and Patterns» — *El concepto de raíz de composición («composition root»).*

PARTE VII

Seguridad creativa

. . .

Interpretaciones originales de la confidencialidad, la integridad y el aislamiento entre usuarios.

009 La variable global que conoce hasta tu vida privada

GAÑANADA Nº 009

La variable global que conoce hasta tu vida privada

. . .

CATEGORÍA	SEGURIDAD CREATIVA
GRAVEDAD	■■■■□ IV — EXTREMO
FECHA DEL HALLAZGO	11 DE NOVIEMBRE DE 2026
ETIQUETAS	#estado-global #concurrencia #node-js #privacidad #async

§ 1

Descripción del hallazgo

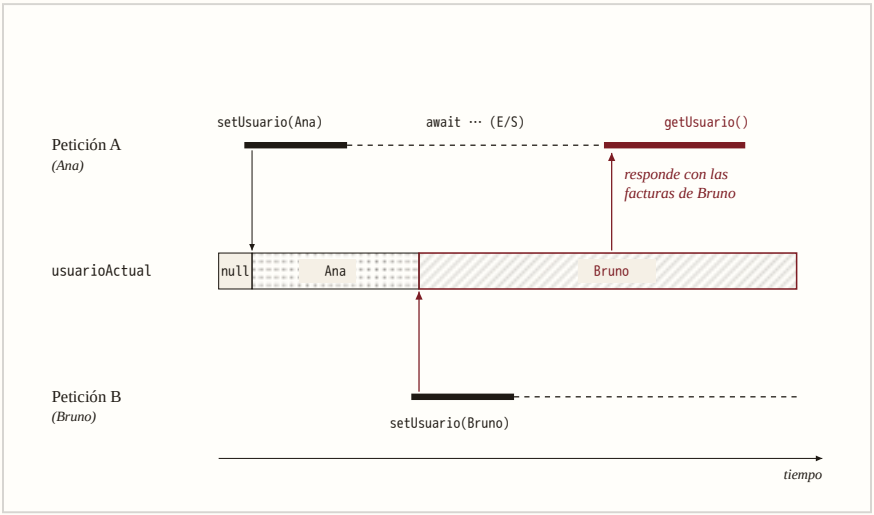
El espécimen habita en el servidor de una aplicación de facturación escrita en Node.js:

```
// contexto.js
let usuarioActual = null;
export const setUsuario = (u) => {
  usuarioActual = u;
};
export const getUsuario = () => usuarioActual;

// app.js
app.use(async (req, res, next) => {
  setUsuario(await sesiones.resolver(req.cookies.sid));
  next();
});

app.get('/api/mis-facturas', async (req, res) => {
  const filtros = await preferencias.cargar(req.query); // ← aquí se cede el
turno
  res.json(await facturas.de(getUsuario().id, filtros));
});
```

Node.js ejecuta el código JavaScript en un solo hilo, y de ahí nace una creencia popular: «no hay problemas de concurrencia». Es falsa. Cada `await` cede el turno al bucle de eventos, y mientras la petición de Ana espera a la base de datos, la de Bruno entra, ejecuta el *middleware* y sobrescribe `usuarioActual`. Cuando la petición de Ana se reanuda, recibe con total amabilidad las facturas de Bruno.



Lám. 9.1 — Cronología de una fuga. Dos peticiones, una variable y un `await` en el peor sitio posible.

§ 2

Contexto del descubrimiento

En el entorno de desarrollo el problema no existe: hay un único usuario, que es quien desarrolla. En preproducción tampoco: las pruebas manuales se hacen de una en una. Los tests automáticos son secuenciales. Todo estaba verde.

En producción, con cientos de peticiones por segundo en horas punta, la ventana entre la asignación y la lectura era suficiente. El hallazgo llegó por la vía más incómoda posible: un cliente escribió, con exquisita

educación, al delegado de protección de datos para preguntar por qué podía ver las facturas de otra empresa.

Lo que siguió fue lo previsto por el RGPD para una violación de la seguridad de los datos personales: evaluación del alcance, registro del incidente y notificación a la autoridad de control sin dilación indebida, en un plazo de 72 horas siempre que sea posible.

§ 3

Consecuencias observadas

- **Fuga de datos entre usuarios**, con obligaciones legales de notificación y una pérdida de confianza difícil de cuantificar.
- **Fallo intermitente y dependiente de la carga**, imposible de reproducir en local y casi imposible de demostrar con los registros habituales.
- **Falsa seguridad en los tests**, que se ejecutaban de uno en uno y no podían ver el problema.
- **Auditoría completa del código**, porque si existía una variable así, nadie podía garantizar que fuera la única.

§ 4

Recomendación de la academia

Los datos de una petición pertenecen a esa petición. Nunca los guardes en estado mutable compartido: variables de módulo, campos estáticos, atributos de un *servlet* o cualquier otro lugar que sobreviva entre peticiones.

La solución más simple es pasar el contexto de forma explícita:

```
app.get('/api/mis-facturas', async (req, res) => {
  const filtros = await preferencias.cargar(req.query);
  res.json(await facturas.de(req.usuario.id, filtros));
});
```

Cuando el contexto debe atravesar muchas capas, Node.js ofrece `AsyncLocalStorage`, que asocia un valor a toda la cadena asíncrona de una petición:

```
import { AsyncLocalStorage } from 'node:async_hooks';

export const contexto = new AsyncLocalStorage();

app.use(async (req, res, next) => {
  const usuario = await sesiones.resolver(req.cookies.sid);
  contexto.run({ usuario }, next);
});

// En cualquier punto de la cadena asíncrona de esa petición:
const { usuario } = contexto.getStore();
```

Y, sobre todo, **prueba el aislamiento con concurrencia real**: un test que lance cientos de peticiones simultáneas de usuarios distintos y compruebe que cada respuesta contiene solo los datos de quien la pidió. Es barato de escribir y habría detectado el espécimen el primer día.



No trabajes así, no seas un gañán.

REFERENCIAS

1 Node.js — «Asynchronous context tracking» (AsyncLocalStorage)

2 Reglamento (UE) 2016/679 (RGPD), artículo 33 — notificación de violaciones de seguridad

PARTE VIII

Git y otras tragedias

. . .

Commits, ramas y empujones forzados que reescriben la historia del repositorio y, ocasionalmente, la de la empresa.

010 El commit que nunca debió abandonar el ordenador de su creador

GAÑANADA Nº 010

El commit que nunca debió abandonar el ordenador de su creador

· · ·

CATEGORÍA	GIT Y OTRAS TRAGEDIAS
GRAVEDAD	■■■■■ V — PATRIMONIO HISTÓRICO DE LA GAÑANERÍA
FECHA DEL HALLAZGO	18 DE NOVIEMBRE DE 2026
ETIQUETAS	#git #secretos #credenciales #historial #datos-personales

§ 1

Descripción del hallazgo

El espécimen, conservado en el historial del repositorio interno, presenta la siguiente ficha:

```
$ git show --stat 3f9c2e1
commit 3f9c2e1 (HEAD -> main, origin/main)
Date:   Fri Nov 13 18:52:07 2026 +0100

    cambios

 .env                                |    12 +
 config/produccion.yml               |     4 +-
 dump_produccion_FINAL_v2_bueno.sql  | Bin 0 -> 3412885504 bytes
 node_modules/...                    | (48 213 archivos)
 src/pagos/PasarelaService.ts        |     1 +
```

Un solo commit con cinco problemas, un mensaje que no describe ninguno y una línea de código que, según su autor, «era lo único que quería subir».

El archivo `.env` contenía las claves de acceso a la cuenta de nube de producción, con permisos de administración. El volcado contenía la base de datos de clientes completa: nombres, direcciones, teléfonos e historial de pedidos.

§ 2

Contexto del descubrimiento

La secuencia, reconstruida a partir de los *reflogs* y de los registros del servidor, fue la siguiente:

1. A las 18:49, el autor ejecutó `git add .` para incluir su cambio de una línea.
2. A las 18:52, `git commit -m "cambios"` y `git push`. El servidor rechazó el envío porque dos compañeros habían subido trabajo esa tarde.
3. A las 18:53, `git push --force`. El envío se aceptó. El trabajo de los compañeros desapareció de la rama principal.
4. A las 18:58, consciente de que «había subido algo raro», el autor ejecutó `git rm .env dump_produccion_FINAL_v2_bueno.sql`, hizo un commit titulado «quitado lo sensible» y se fue a casa.

El segundo commit eliminó los archivos de la versión actual, pero no del historial: cualquier `git checkout 3f9c2e1` los devuelve intactos. Para entonces, el repositorio ya se había sincronizado en el servidor de integración continua, en su caché de artefactos y en los portátiles de tres personas que hicieron `git pull` antes de cenar.

El hallazgo se produjo el lunes, cuando la integración continua empezó a fallar por falta de espacio en disco y alguien preguntó por qué el repositorio pesaba 3,5 GB.

§ 3

Consecuencias observadas

- **Credenciales comprometidas.** Desde el momento del push, las claves deben darse por expuestas, aunque el repositorio sea privado. Hubo que rotarlas todas y revisar los registros de acceso de la cuenta de nube.
- **Datos personales replicados** en servidores, cachés y equipos de trabajo, con la obligación de evaluar el incidente como una posible violación de seguridad de datos personales.
- **Trabajo ajeno destruido** por el push forzado. Se recuperó gracias al *reflog* local de uno de los compañeros afectados, que aún no había actualizado su copia.
- **Reescritura del historial**, que obligó a coordinar a todo el equipo para limpiar el repositorio con `git filter-repo` y volver a clonarlo, y a invalidar las cachés de la integración continua.

§ 4

Recomendación de la academia

Git es muy bueno recordando. Esa es precisamente su función, y conviene tenerlo presente antes de pedirle que recuerde algo.

Antes de cada commit, revisa lo que vas a subir: `git status` y `git diff --staged`. Añade archivos por nombre, no con `git add .` a ciegas, y escribe mensajes que describan el cambio. Si el mensaje honesto sería «cinco cosas no relacionadas», son cinco commits.

Protege el repositorio desde el primer día:

```
# .gitignore mínimo de supervivencia
.env
.env.*
*.sql
*.dump
node_modules/
```

- **Detección de secretos** antes del commit (por ejemplo, Gitleaks como *hook* de pre-commit) y en el servidor, rechazando los envíos que contengan credenciales.
- **Rama principal protegida**: sin push forzado, con integración mediante *pull request*. En tus propias ramas, usa `git push --force-with-lease`, que se niega a sobrescribir trabajo que no has visto.
- **Ningún dato personal real en el repositorio.** Para pruebas, datos sintéticos o anonimizados.

Si el secreto ya ha salido, el orden importa: primero rota las credenciales —asume que están comprometidas—; después limpia el historial con `git filter-repo`, coordinando al equipo; y por último revisa qué accesos se produjeron mientras estuvieron expuestas. Borrar el archivo en un commit nuevo no es una solución: es una nota a pie de página que señala dónde está el tesoro.



No trabajes así, no seas un gañán.

REFERENCIAS

1 GitHub Docs — «Removing sensitive data from a repository»

2 git-filter-repo

3 Gitleaks — detección de secretos en repositorios

COLOFÓN

Este volumen I de la Bitácora del Gañán reúne 10entradas en 8categorías.

Compuesto en Source Serif 4 e IBM Plex Mono.
Edición generada automáticamente a partir de las mismas fuentes Markdown que alimentan la versión web.

Versión en línea: https://axxa.embeddedsource.es/bitacora-del-ganan



No trabajes así, no seas un gañán.