ALEXIS VIETO

SECOND OPINION · AUDITORÍA INDEPENDIENTE DE SOFTWARE

AUDITORÍA DE SEGURIDAD DE CÓDIGO

AUDITORÍA DE SEGURIDAD DE [redacted]

Revisión del código y pruebas de ataque sobre una copia aislada de la aplicación, un gestor de finanzas personales que corre entero en la Mac del usuario.

[redacted] NOMBRE DE LA APLICACIÓN RESERVADO POR CONFIDENCIALIDAD. EL RESTO DEL INFORME ESTÁ COMPLETO Y SIN CAMBIOS.

PROYECTO	AUDITÓ	MÉTODO	HALLAZGOS	REVISIÓN
AUDITORÍA · [redacted]	A. VIETO	CAJA BLANCA · OWASP	9 · 0 CRÍTICOS	2026.10

OBJETIVO	COMMIT	FECHA DEL INFORME
[redacted] 0.1.0 (Node.js)	d14c1ee	25 de octubre de 2026

MÉTODO	ELABORADO POR
Caja blanca, OWASP	Alexis Vieto · Second Opinion

VEREDICTO

FAVORABLE.

El cifrado de las bóvedas, el aislamiento entre usuarios y las consultas SQL resistieron todas las pruebas. No encontramos inyección SQL, XSS explotable ni fallas en el cifrado.

Los nueve hallazgos están en controles de apoyo. Los cuatro de severidad media son una carrera en el registro que pierde cuentas, un bloqueo por intentos fallidos que se salta con peticiones en paralelo, mensajes de error internos que llegan al navegador y una dependencia con una vulnerabilidad publicada.

Recomendamos corregir los cuatro (**SO-01 a SO-04**) en el próximo ciclo y antes de dejar [redacted] publicada en Tailscale de forma permanente.

0	0	4	5	99/100
Críticos	Altos	Medios	Bajos	Pruebas del proyecto en verde (1 omitida)

01

ALCANCE.

Revisamos los 29 módulos de `src/`, el esquema SQL, la interfaz de `public/`, los envoltorios nativos de `mac/` y las dependencias.

[redacted] sirve su interfaz desde un servidor Node.js enlazado a `127.0.0.1` y la muestra en una ventana nativa de macOS. El acceso desde el iPhone es opcional y pasa por la red privada Tailscale. No tiene servidor remoto, cuenta en internet ni telemetría.

ÁREA	ARCHIVOS	QUÉ REVISAMOS
Criptografía	`boveda.js`	AES-256-GCM, derivación con scrypt, doble envoltura de la llave, códigos de recuperación
Autenticación y sesión	`usuarios.js`	Sesiones, cookies, intentos fallidos, recuperación, enumeración de usuarios
Datos y aislamiento	`db.js`, `esquema.sql`	Inyección SQL, aislamiento por usuario con AsyncLocalStorage
Capa HTTP	`app.js`, `servidor.js`	Rutas, autorización, control de origen, CSRF, cabeceras
Archivos de entrada	`importador.js`, `pdf.js`, `facturas.js`	Lectores de XLSX, CSV, HTML y PDF; `execFile`; ReDoS; consumo de recursos
Integración externa	`iphone.js`	CLI de Tailscale, inyección de argumentos
Interfaz	`public/app.js`, `sw.js`, `index.html`	XSS en el DOM, service worker, almacenamiento local
Dependencias	`package-lock.json`	Vulnerabilidades publicadas (CVE y avisos de GitHub)

02

MÉTODO.

Combinamos lectura del código con ataques reales contra una instancia en vivo, aislada en directorios temporales y el puerto 4611.

Lectura del código. Los 29 módulos del servidor, el esquema SQL y las 2 127 líneas de la interfaz. Seguimos cada dato desde la petición HTTP hasta donde se usa: SQL, sistema de archivos, `execFile` e `innerHTML`.

Dependencias. Instalación reproducible con `npm ci` y `npm audit` sobre el árbol bloqueado de 118 paquetes.

Pruebas del proyecto. Las 100 pruebas de `node --test` : 99 pasan y 1 está omitida.

Ataques con scripts propios contra la instancia viva:

- 24 cargas de inyección SQL (`' OR '1'='1` , `;DROP TABLE` , `UNION` y otras) en `buscar` , `mes` , `tipo` , `al` y los campos de cuenta.
- Registros simultáneos (6 y 3 a la vez), revisando después lo que quedó en disco.
- Bloqueo por intentos fallidos: 5 intentos seguidos contra 40 en paralelo.
- Enumeración de usuarios por tiempo de respuesta en `/api/sesion` y `/api/usuarios/recuperar` .
- Control de origen con un cliente HTTP crudo, con `Host` y `Origin` arbitrarios, para simular DNS rebinding y peticiones sin `Origin` .
- Archivos HTML grandes al importador, midiendo cuánto se demoran las demás peticiones mientras se procesan.
- XSS almacenado con `<img src=x onerror=…>` en nombres de cuenta.
- Cabeceras de respuesta, permisos y cifrado de lo que la aplicación escribe en disco.

está hecha para macOS. Las pruebas corrieron en Windows 11 con Node.js 24.15.0, PGlite 0.5.10 y Express 5.3.0.

03

HALLAZGOS.

Nueve hallazgos, ninguno crítico ni alto. Ocho los confirmamos con una prueba; el noveno es una observación de diseño.

ID	HALLAZGO	SEVERIDAD	CWE	ESTADO
SO-01	Carrera en el registro que pierde cuentas	Media	362	Confirmado
SO-02	El bloqueo por intentos se salta con peticiones en paralelo	Media	307	Confirmado
SO-03	Errores internos llegan al navegador	Media	209	Confirmado

ID	HALLAZGO	SEVERIDAD	CWE	ESTADO
SO-04	Dependencia vulnerable: `body-parser`	Media	1035	Confirmado
SO-05	La recuperación revela qué usuarios existen	Baja	208	Confirmado
SO-06	El control de origen se puede saltar	Baja	350	Confirmado
SO-07	Un archivo grande congela la aplicación	Baja	400	Confirmado
SO-08	Faltan cabeceras de seguridad HTTP	Baja	693	Confirmado
SO-09	Confianza en cabeceras de proxy en el enlace local	Baja	290	Observación

Las barras marcan la severidad en cuatro niveles: baja, media, alta y crítica.

04

DETALLE Y CORRECCIÓN.

Cada hallazgo trae la evidencia, la corrección en código y una prueba que falla hoy y debe pasar después de corregir.

Para correr las pruebas, levanta una instancia en un directorio desechable. No toca datos reales.

```
# Terminal 1: instancia de pruebas
DATOS_DIR=/tmp/[redacted]-sec PORT=4611 node src/servidor.js

# Terminal 2: cada prueba de abajo, con BASE = 'http://127.0.0.1:4611'
node prueba.mjs
```

Conviene pasarlas a archivos `*.prueba.js` del proyecto para que corran con `node --test` junto a las 100 actuales.

SO-01

Media

Carrera en el registro que pierde cuentas

CWE 362, condición de carrera DÓNDE `usuarios.js`, `leerRegistro` y `escribirRegistro`

IMPACTO Integridad y disponibilidad

HOY, 6 ALTAS SIMULTÁNEAS

~~1 de 6~~

quedan en el registro

META TRAS CORREGIR

6 de 6

QUÉ PASA

`registrar()` lee `usuarios.json`, agrega el usuario y vuelve a escribir el archivo, sin exclusión mutua. Entre la lectura y la escritura hay operaciones `await` (crear y cifrar la base del usuario), así que varias altas simultáneas leen el mismo estado y la última escritura borra a las demás.

Cada alta crea su base cifrada en disco y recibe su cookie de sesión, pero su entrada en el registro se pierde. La cuenta queda **huérfana**: nadie puede volver a entrar en ella. La misma carrera se salta la verificación de usuario repetido.

EVIDENCIA

Seis registros simultáneos (`carrera0` a `carrera5`) respondieron `200 OK`. El disco muestra la pérdida:

```
# bases creadas en disco
$ ls datos-poc/bases | wc -l
10
# el registro solo conservó una de las seis altas
$ node -e "...usuarios.json..."
usuarios en registro: 3
nombres: victima, carrera5, duplicado
# carrera0 a carrera4: base cifrada en disco, sin entrada para iniciar sesión
```

Tres altas simultáneas del mismo usuario (`duplicado`) también respondieron `200 OK` las tres.

CORRECCIÓN

Pasar todas las escrituras del registro por una cola de promesas, igual que ya hace `guardar()` con las bases. Leer el registro y revisar si el usuario existe *dentro* de la cola. Aplicar lo mismo a

`recuperar()` y `cambiarClave()`, que también leen, modifican y escriben `usuarios.json`. Crear la base del usuario solo después de tener su lugar en el registro, o borrarla si el registro falla.

```
// usuarios.js: una cola para el registro
let colaRegistro = Promise.resolve();
function conRegistro(fn) {
  const r = colaRegistro.then(fn, fn);   // en orden, aunque el anterior falle
  colaRegistro = r.catch(() => {});
  return r;
}

export async function registrar(datos) {
  const v = revisarRegistro(datos);
  if (v.error) return v;
  // derivar la llave, el código, etc. FUERA de la cola
  return conRegistro(async () => {
    const reg = leerRegistro();                          // leer DENTRO de la cola
    if (reg.usuarios.some((u) => u.usuario === usuario))
      return { error: `El usuario ${usuario} ya existe` };
    reg.usuarios.push(u);
    escribirRegistro(reg);                               // escribir antes de ceder el turno
    return { ok: true, /* ... */ };
  });
}
```

PRUEBA DE REGRESIÓN

```
// Deben quedar las 6 altas y tantas bases como entradas en el registro.
import fs from 'node:fs';
const post = (p, b) => fetch(BASE + p, { method:'POST',
  headers:{'content-type':'application/json'}, body: JSON.stringify(b) });

const rs = await Promise.all([0,1,2,3,4,5].map((i) =>
  post('/api/usuarios', { nombre:'R'+i, usuario:'carrera'+i, clave:'clave-carrera-'+i })
    .then((r) => r.status)));

const reg = JSON.parse(fs.readFileSync(process.env.DATOS_DIR + '/usuarios.json','utf8'))
  .usuarios.filter((u) => u.usuario.startsWith('carrera')).length;
const bases = fs.readdirSync(process.env.DATOS_DIR + '/bases').length;

console.assert(rs.filter((s) => s === 200).length === 6, 'las 6 deben responder 200');
console.assert(reg === 6, `el registro debe tener 6 (tiene ${reg})`);   // hoy: 1
console.assert(bases === reg, 'no deben quedar bases huérfanas');       // hoy: 10 contra 1

// Usuario repetido en paralelo: solo una alta debe pasar.
const dup = await Promise.all([0,1,2].map(() =>
  post('/api/usuarios', { nombre:'D', usuario:'dup', clave:'clave-dup-123'
}).then((r)=>r.status)));
console.assert(dup.filter((s) => s === 200).length === 1, 'solo una alta de "dup" debe pasar');
```

SO-02

Media

El bloqueo por intentos se salta con peticiones en paralelo

CWE 307, límite de intentos de autenticación

DÓNDE `usuarios.js`, `bloqueado` y `fallo`

IMPACTO Fuerza bruta de claves

HOY, 40 INTENTOS EN PARALELO

~~40 de 40~~

se evalúan

META TRAS CORREGIR

5 o menos

QUÉ PASA

La regla es 5 intentos y luego 60 segundos de espera. Cada petición revisa `bloqueado(usuario)` al empezar y suma un fallo al terminar. Como revisar y sumar no ocurren juntos y el servidor atiende peticiones en paralelo, un lote simultáneo pasa la revisión antes de que ninguno haya sumado. Cada lote trae todos los intentos que el atacante quiera.

EVIDENCIA

```
# En serie el bloqueo funciona: desde el 6.º intento responde 429.
bloqueo_serie: [401,401,401,401,401,429,429]

# En paralelo, 40 intentos simultáneos y ninguno bloqueado:
bloqueo_paralelo: { intentos: 40, evaluados_401: 40, bloqueados_429: 0 }
```

Lo que lo atenúa: cada verificación corre scrypt ($N=2^{15}$, unos 100 ms), la clave mínima es de 8 caracteres y el código de recuperación tiene unos 98 bits de entropía. La fuerza bruta sigue siendo cara, pero el control que la debía frenar no funciona.

CORRECCIÓN

Poner en fila los intentos de cada usuario para que se evalúen uno tras otro y el contador se actualice antes del siguiente. Agregar una espera creciente y un límite global por proceso, además del límite por usuario.

```
// usuarios.js: un turno por usuario para entrar y recuperar
const turnos = new Map();                          // usuario -> última promesa
function enFila(usuario, fn) {
  const previo = turnos.get(usuario) ?? Promise.resolve();
  const r = previo.catch(() => {}).then(fn);
  turnos.set(usuario, r.catch(() => {}));
  return r;
}

export function entrar(usuario, clave) {
  const nombre = limpio(usuario).toLowerCase();
  return enFila(nombre, async () => {            // bloqueado() y fallo() ya no se cruzan
    const espera = bloqueado(nombre);
    if (espera) return { error: `Demasiados intentos. Espera ${espera} segundos.`, estado: 429 };
    // ...resto igual...
  });
}
```

PRUEBA DE REGRESIÓN

```
// 40 intentos en paralelo: a lo sumo 5 se evalúan; el resto, 429.
const rs = await Promise.all(Array.from({length:40}, (_,i) =>
  post('/api/sesion', { usuario:'victima', clave:'mala'+i }).then((r)=>r.status)));
const pasaron = rs.filter((s) => s === 401).length;
console.assert(pasaron <= 5, `no deben pasar más de 5 intentos (pasaron ${pasaron})`);  // hoy: 40
console.assert(rs.includes(429), 'debe haber al menos un 429');                         // hoy: 0
```

SO-03

Media

Errores internos llegan al navegador

CWE 209, información en mensajes de error DÓNDE `app.js`, manejador de errores

IMPACTO Divulgación de información

QUÉ PASA

El manejador global devuelve el mensaje de la excepción tal cual: `res.json({ error: \`Error interno: ${err.message}\` })`. Las entradas que nadie valida llegan a PGlite, y el error del motor viaja hasta el navegador con tipos de datos, nombres de operaciones internas y la tecnología de la base.

No se filtran datos de otros usuarios; el aislamiento se mantiene. Sí se filtran detalles útiles para quien ataca.

EVIDENCIA

```
GET /api/patrimonio?al='                      → 500  "Error interno: invalid input syntax for type
date: \"'\""
GET /api/patrimonio/evolucion?meses=999999999 → 500  "Error interno: Invalid time value"
GET /api/cuentas/abc/valoraciones             → 500  "Error interno: invalid input syntax for type
integer: \"NaN\""
```

Estos casos muestran además que los parámetros `al`, `mes`, `meses` y los `:id` de ruta llegan al motor sin validar. Siempre llegan como parámetros ligados, nunca concatenados (ver sección 5).

CORRECCIÓN

Validar fechas y números en la ruta y responder `400` con un mensaje claro. En el manejador global, devolver texto solo para errores de negocio (los que traen `estado`); el resto va completo al log del servidor y el cliente recibe «Error interno».

```
// app.js: manejador de errores
app.use((err, req, res, _next) => {
  if (err.estado) return res.status(err.estado).json({ error: err.message });  // error de negocio
  console.error(err);                                                          // el detalle, solo
al log
  res.status(500).json({ error: 'Error interno' });                            // mensaje genérico
});

// Validación en la ruta (ejemplo con una fecha):
const fechaValida = (s) => /^\d{4}-\d{2}-\d{2}$/.test(String(s));
api.get('/patrimonio', async (req, res) => {
  const al = req.query.al ?? hoy();
  if (!fechaValida(al)) return res.status(400).json({ error: 'La fecha va como AAAA-MM-DD' });
  res.json(await patrimonio.patrimonio(al));
});
```

PRUEBA DE REGRESIÓN

```
// Una entrada inválida da 400 con mensaje genérico, nunca 500 con el error del motor.
const casos = ['/api/patrimonio?al=\'', '/api/cuentas/abc/valoraciones',
               '/api/patrimonio/evolucion?meses=999999999'];
for (const ruta of casos) {
  const r = await fetch(BASE + ruta, { headers:{ cookie } });
  const b = await r.json();
  console.assert(r.status !== 500, `${ruta} no debe dar 500 (dio ${r.status})`);          // hoy:
500
  console.assert(!/invalid input syntax|Invalid time value|NaN/.test(b.error ?? ''),
    `${ruta} no debe mostrar el error del motor`);                                       // hoy:
lo muestra
}
```

SO-04

Media

Dependencia vulnerable: `body-parser`

CWE 1035 y 400 DÓNDE `body-parser` 2.3.0, vía Express 5.3.0

IMPACTO Denegación de servicio

HOY, `NPM AUDIT`

~~1 alta~~

vulnerabilidad publicada

META TRAS CORREGIR

0

QUÉ PASA

`npm audit` reporta el aviso GHSA-27r3-jxmh-p828, de severidad **alta**: denegación de servicio con bytes sobrantes después de un cuerpo comprimido, en `body-parser` 2.0.0 a 2.3.0. [redacted] tiene la 2.3.0, que trae Express 5.3.0, y acepta cuerpos JSON comprimidos.

Aquí queda en severidad media porque el servidor solo escucha en `127.0.0.1`. Lo alcanza un proceso local o, si [redacted] está publicada, cualquier equipo de la red de Tailscale.

EVIDENCIA

```
$ npm audit
body-parser  2.0.0 - 2.3.0   Severity: high
body-parser vulnerable to Denial of Service via trailing bytes after a compressed request body
fix available via `npm audit fix`
  node_modules/body-parser
1 high severity vulnerability

$ npm ls body-parser
[redacted]@0.1.0
└┬ express@5.3.0
  └── body-parser@2.3.0        # versión corregida: 2.4.0
```

CORRECCIÓN

```
$ npm audit fix          # sube body-parser a 2.4.0 o más, dentro del rango de Express 5
$ npm audit              # debe reportar 0 vulnerabilidades
```

PRUEBA DE REGRESIÓN

Agregar `npm audit` al flujo de integración para que falle si vuelve a aparecer una vulnerabilidad alta o crítica.

```
$ npm audit --audit-level=high
# Si Express todavía no sube el rango, fijar la versión en package.json:
"overrides": { "body-parser": ">=2.4.0" }
```

SO-05

Baja

La recuperación revela qué usuarios existen

CWE 208 y 203, canal lateral de tiempo DÓNDE `usuarios.js`, `recuperar()`

IMPACTO Revela qué cuentas existen

HOY, DIFERENCIA DE TIEMPO

~~55 ms~~

entre usuario existente e inexistente

META TRAS CORREGIR

< 25 ms

QUÉ PASA

`entrar()` está bien protegido: si el usuario no existe, deriva una clave contra `SAL_FALSA` para tardar lo mismo. `recuperar()` no lo hace. Si el usuario no existe, responde de inmediato (`u ? await desenvolver(...) : null`) sin correr scrypt, y esa diferencia de tiempo en `/api/usuarios/recuperar` dice qué usuarios existen.

EVIDENCIA

```
# Mediana de latencia según si el usuario existe:
/api/usuarios/recuperar  → existe: 58 ms   no existe: 3 ms   # se distingue a simple vista
/api/sesion (login)      → equilibrado por SAL_FALSA, no se distingue
```

CORRECCIÓN

Copiar en `recuperar()` lo que ya hace `entrar()`. El mensaje de error genérico ya está bien.

```
// usuarios.js: recuperar()
const u = reg.usuarios.find((x) => x.usuario === nombre);
if (!u) { await derivar(normalizarCodigo(codigo), SAL_FALSA); fallo(nombre);  // nuevo
  return { error: 'Usuario o código de recuperación incorrectos', estado: 401 }; }
const llave = await desenvolver(u.con_codigo, normalizarCodigo(codigo));
```

PRUEBA DE REGRESIÓN

```
// La mediana con usuario existente y sin él debe ser parecida.
const medir = async (u) => { const a = performance.now();
  await post('/api/usuarios/recuperar', { usuario:u, codigo:'AAAA-AAAA-AAAA-AAAA-AAAA',
nueva:'otra-clave-99' });
  return performance.now() - a; };
const med = (a) => a.sort((x,y)=>x-y)[a.length>>1];
const ex = [], no = [];
for (let i=0;i<5;i++){ ex.push(await medir('victima')); no.push(await medir('nadie'+i)); }
console.assert(Math.abs(med(ex) - med(no)) < 25,
  `la diferencia debe ser pequeña (fue ${Math.round(Math.abs(med(ex)-med(no)))} ms)`);  // hoy:
unos 55 ms
```

SO-06

Baja

El control de origen se puede saltar

CWE 350 y 352 DÓNDE `app.js`, middleware de origen

IMPACTO Una capa de defensa menos

QUÉ PASA

En las peticiones que cambian datos, el middleware compara el host de `Origin` con el `Host` de la petición. Tiene dos huecos:

- Si falta `Origin`, la petición pasa sin revisión (`if (!origen) return next()`).
- Acepta cualquier `Host` que coincida con el `Origin`, sin exigir `localhost` o `127.0.0.1`. Con **DNS rebinding**, un dominio del atacante que resuelve a 127.0.0.1 cumple la igualdad y pasa.

El daño real es limitado porque la defensa principal funciona. La cookie de sesión es `SameSite=Strict` y `HttpOnly` y pertenece al origen `127.0.0.1`; después del rebinding el navegador trata el dominio del atacante como otro origen y no envía la cookie de [redacted]. El atacante llega a las rutas que no piden sesión, como registrarse o intentar entrar con el límite de intentos, pero no a los datos del usuario. Aun así, el control no protege lo que parece proteger.

EVIDENCIA

```
# Cliente HTTP crudo (fetch no deja fijar Host; se usó el módulo net):
Host=Origin=banco-malo.com      → 401  pasa el control y llega al login   # rebinding
Host=banco-malo.com, sin Origin → 401  pasa el control                    # sin Origin
Host=127.0.0.1, Origin=otro.com → 403  "Pedido de otro origen"            # CSRF clásico: bloqueado
```

CORRECCIÓN

Validar `Host` contra una lista explícita (`localhost`, `127.0.0.1` y el sufijo `.ts.net` esperado) y exigir `Origin` en los métodos que cambian datos. `SameSite=Strict` sigue siendo el control principal.

```
// app.js: middleware de origen
const HOST_OK = (h) => /^(localhost|127\.0\.0\.1)(:\d+)?$/.test(h) || /\.ts\.net$/.test(h);

app.use('/api', (req, res, next) => {
  if (req.method === 'GET') return next();
  const host = req.get('x-forwarded-host') || req.get('host') || '';
  if (!HOST_OK(host)) return res.status(403).json({ error: 'Host no permitido' });  // frena el
rebinding
  const origen = req.get('origin');
  if (!origen) return res.status(403).json({ error: 'Falta el origen' });           // sin Origin
ya no pasa
  let oHost = null; try { oHost = new URL(origen).host; } catch {}
  if (oHost !== host) return res.status(403).json({ error: 'Pedido de otro origen' });
  next();
});
```

PRUEBA DE REGRESIÓN

```
// fetch() no deja fijar Host; se usa el módulo net para armar las cabeceras.
import net from 'node:net';
const crudo = (h) => new Promise((ok) => {
  const body = '{}'; const s = net.connect(4611, '127.0.0.1', () => s.write(
    ['POST /api/sesion HTTP/1.1', ...Object.entries(h).map(([k,v])=>`${k}: ${v}`),
     'Content-Type: application/json', `Content-Length: ${body.length}`, '', body].join('\r\n')));
  let d=''; s.on('data',(c)=>d+=c); setTimeout(()=>{s.end(); ok(d.split(' ')[1]);}, 300);
});
console.assert(await crudo({ Host:'banco-malo.com', Origin:'http://banco-malo.com' }) === '403');
// rebinding, hoy: 401
console.assert(await crudo({ Host:'banco-malo.com' }) === '403');
// sin Origin, hoy: 401
console.assert(await crudo({ Host:'127.0.0.1:4611', Origin:'http://127.0.0.1:4611' }) !== '403');
// lo legítimo sigue pasando
```

SO-07

Baja

Un archivo grande congela la aplicación

CWE 400 y 1333 DÓNDE `importador.js`, `tablaHtml` y `leerCsv`

IMPACTO La aplicación deja de responder mientras importa

HOY, OTRA PETICIÓN DURANTE LA IMPORTACIÓN

~~3 079 ms~~

de espera

META TRAS CORREGIR

< 500 ms

QUÉ PASA

El importador lee las tablas HTML con expresiones regulares y los CSV carácter por carácter, en el hilo principal. Un archivo grande, dentro del límite de 20 MB de `/api/importar/previa`, consume CPU en forma cuadrática y bloquea el bucle de eventos: el resto de la aplicación no responde hasta que termina. [redacted] es local y de un solo usuario, así que sobre todo se afecta a sí mismo.

EVIDENCIA

```
# Tiempo de proceso según el tamaño (etiquetas <tr sin cerrar):
 59 KB → 616 ms      117 KB → 1 164 ms      234 KB → 3 430 ms
# Mientras procesaba 234 KB, un GET /api/sesion tardó:
latencia_otro_pedido_ms: 3 079   # unos 3 s con el bucle de eventos bloqueado
```

CORRECCIÓN

Bajar el límite de subida a lo que pesa un estado de cuenta real, limitar filas y celdas, y ajustar las expresiones de `tablaHtml` para evitar el retroceso catastrófico. Para archivos grandes, procesarlos en un *worker* aparte.

```
// app.js: un estado de cuenta rara vez pasa de 2 a 3 MB
app.use('/api/importar/previa', express.raw({ type: () => true, limit: '4mb' }));

// importador.js: tope de filas
const MAX_FILAS = 10000;
return [...html.matchAll(/<tr[\s\S]*?<\/tr>/gi)].slice(0, MAX_FILAS).map(/* ... */);
```

PRUEBA DE REGRESIÓN

```
// Un archivo patológico no debe bloquear el bucle: otra petición responde rápido.
const ataque = fetch(`${BASE}/api/importar/previa?cuenta_id=${id}&nombre=x.html`,
  { method:'POST', headers:{ cookie }, body: '<table>' + '<tr'.repeat(200000) });
await new Promise((r) => setTimeout(r, 200));
const a = performance.now(); await fetch(BASE + '/api/sesion'); const lat = performance.now() - a;
await ataque;
console.assert(lat < 500, `otra petición no debe esperar (esperó ${Math.round(lat)} ms)`);  // hoy:
unos 3000 ms
```

SO-08

Baja

Faltan cabeceras de seguridad HTTP

CWE 693, falla de un mecanismo de protección DÓNDE `app.js`

IMPACTO Una capa de defensa menos

QUÉ PASA

Las respuestas no traen `Content-Security-Policy`, `X-Content-Type-Options: nosniff`, `X-Frame-Options` ni `Referrer-Policy`, y anuncian `X-Powered-By: Express`. La interfaz arma HTML por interpolación; hoy escapa bien (ver sección 5), y una CSP estricta sería la segunda barrera si algún día deja de hacerlo.

EVIDENCIA

```
# Única cabecera relevante en la respuesta:
cabeceras: { "x-powered-by": "Express" }   # sin CSP, sin nosniff, sin X-Frame-Options
```

CORRECCIÓN

```
// app.js: después de crear la app
app.disable('x-powered-by');
app.use((req, res, next) => {
  res.set('Content-Security-Policy',
    "default-src 'self'; img-src 'self' data: blob:; object-src 'none'; frame-ancestors 'none';
base-uri 'none'");
  res.set('X-Content-Type-Options', 'nosniff');
  res.set('Referrer-Policy', 'no-referrer');
  next();
});
```

Probar la CSP con la interfaz actual: usa `blob:` para la vista previa de fotos y SVG en línea. Si algo deja de cargar, ajustar `img-src` o `style-src`.

PRUEBA DE REGRESIÓN

```
const r = await fetch(BASE + '/');
console.assert(r.headers.get('content-security-policy'), 'debe haber CSP');
console.assert(r.headers.get('x-content-type-options') === 'nosniff', 'debe haber nosniff');
console.assert(!r.headers.get('x-powered-by'), 'no debe anunciar X-Powered-By');   // hoy:
"Express"
```

SO-09

Baja · observación

Confianza en cabeceras de proxy en el enlace local

CWE 290, suplantación DÓNDE `app.js`, `trust proxy` y `tailscale-user-login`

IMPACTO Una capa de defensa menos

QUÉ PASA

[redacted] define `trust proxy = 'loopback'` y decide si la petición llegó por HTTPS y Tailscale mirando `x-forwarded-proto`, `x-forwarded-host` y `tailscale-user-login`. Esas cabeceras solo son confiables si siempre las pone el proxy de Tailscale. Un proceso local que hable directo con `127.0.0.1` puede inventarlas para que la cookie salga marcada `Secure` o para influir en el control de origen.

El impacto es bajo: quien puede correr procesos en la Mac ya tiene mucho acceso al equipo.

CORRECCIÓN

No tomar decisiones de seguridad con cabeceras `X-Forwarded-*` o `tailscale-user-login` que un cliente local pueda fabricar. La lista de hosts de SO-06 cubre buena parte. Además, marcar la cookie `Secure` según la configuración del servidor y no según una cabecera entrante. Si se puede, servir la instancia de Tailscale en un puerto distinto del local.

```
// app.js: Secure por configuración, no por una cabecera del cliente
const SERVIDO_HTTPS = process.env.[redacted]_HTTPS === 'si';
const seguro = (req) => (req.secure || SERVIDO_HTTPS ? '; Secure' : '');
```

05

LO QUE RESISTIÓ.

Estas áreas las atacamos y aguantaron. Son decisiones de diseño que hay que conservar.

CONTROL	RESULTADO
✓ **Inyección SQL**	No se puede explotar. Todas las consultas usan parámetros ligados (`$1, $2…`). Los identificadores dinámicos (`cols` , `lado`) salen de listas que controla el servidor, nunca del texto del usuario. Las 24 cargas no rompieron ninguna consulta ni cambiaron datos; solo produjeron los errores de tipo de SO-03.
✓ **XSS almacenado**	No se puede explotar. La interfaz pasa todo por `esc()` antes de `innerHTML` , y `notificar()` usa `textContent` . Un nombre de cuenta con `<img onerror>` se guarda, pero se muestra escapado.
✓ **Cifrado de la bóveda**	Sólido. AES-256-GCM con IV aleatorio de 12 bytes por operación y etiqueta de autenticación verificada; scrypt $N=2^{15}$ con sal aleatoria de 16 bytes por envoltura; códigos de recuperación de unos 98 bits. La prueba unitaria confirma que una llave equivocada o un byte alterado fallan.

CONTROL	RESULTADO
✓ **Aislamiento entre usuarios**	Una base PGlite por usuario, contexto por petición con `AsyncLocalStorage` y rechazo explícito de consultas sin usuario en modo multiusuario. Los archivos subidos quedan ligados al `id` de quien los subió.
✓ **Inyección de comandos**	No encontramos ninguna. `pdf.js`, `facturas.js` e `iphone.js` llaman a `execFile` o `spawn` con rutas fijas y argumentos constantes; el contenido del usuario entra por la entrada estándar, no por la línea de comandos.
✓ **Sesiones**	Tokens aleatorios de 32 bytes, cookie `HttpOnly; SameSite=Strict`, vencimiento por inactividad (1 h) y absoluto (12 h), `Secure` sobre HTTPS. El registro se escribe de forma atómica con permisos `0600`.
✓ **Service worker**	Guarda solo la página (HTML, CSS, JS e iconos), nunca respuestas de `/api/` ni datos, y siempre consulta la red primero.

06

PLAN DE CORRECCIÓN.

Tres tandas, de lo que más pesa a lo que endurece.

CUÁNDO	QUÉ HACER	HALLAZGOS
Próximo ciclo	Poner en cola el registro de usuarios, poner en fila los intentos de entrada, ocultar los errores internos y correr `npm audit fix`.	01, 02, 03, 04
Mediano plazo	Igualar el tiempo de `recuperar()`, validar `Host` con una lista, agregar cabeceras de seguridad con CSP y limitar el tamaño y las filas de la importación.	05, 06, 07, 08

CUÁNDO	QUÉ HACER	HALLAZGOS
Endurecimiento	Dejar de confiar en cabeceras de proxy y validar todos los parámetros de fecha y número en la ruta.	09, 03

07

CONCLUSIÓN.

Las áreas donde una falla sería grave están bien resueltas y probadas: el cifrado, el aislamiento de datos y el manejo de SQL. Ningún hallazgo llega a severidad alta o crítica, y todos se corrigen con poco trabajo.

Con los cuatro hallazgos medios corregidos, [redacted] queda en buena posición incluso si se publica de forma permanente en Tailscale.

Confidencial. Probamos una copia aislada del proyecto, en un entorno controlado y sin acceso a datos reales.

Los resultados valen para el commit `d14c1ee`. Cualquier cambio posterior necesita una nueva revisión.