

Security review: Invoicer (demo app)

29 October 2026 · Reviewed by Bardstar · bardstar.com

Reviewed and signed off by: Bardstar reviewer (named on the client's copy of the agreement)

Example review of a demo app Bardstar built and owns. Not a client. Every finding in this report is real on this demo app and can be reproduced with the included script.

This is the kind of report Bardstar delivers after a review. We ran it against a small invoicing app we built on purpose, so we could publish the whole review: the testing, the proof and the fixes.

1. Summary

We tested a private copy of Invoicer for the kinds of mistakes apps built with AI coding tools usually ship with. We found **10 problems: 6 critical, 2 high, 1 medium, 1 low**. The worst let anyone on the internet read every customer's invoices and uploaded ID documents, unlock the paid plan without paying, use a master key that was sitting inside the website's own code, and forge a login as any user or the admin using a signing secret left in the code. We wrote a fix for each problem, applied them, and tested again: **all 10 are now blocked**. Automated scanners flagged a couple of key-like strings but identified none of these problems as the problems they are; every one was proven by testing the running app.

APP Invoicer, a demo invoicing app for freelancers (built and owned by Bardstar).

SCOPE A test copy on `127.0.0.1` only — source code, full git history, and the running app. No public site, no third-party accounts, no real users.

METHOD Automated scanners + specialist AI reviewer agents + manual proof of every finding. Each finding was proven by a real request and is reproduced by `demo-app/prove.sh`.

In plain terms, before the fixes

- Any account could read **every** other customer's invoices, and change them.
- A **master key** that turns off all the app's security rules was shipped inside the website's downloadable code.
- The secret that **signs login sessions** was left in the code and git history, so anyone could forge a login as any user or the admin.
- Anyone could send a fake "payment complete" message and flip any account to the paid plan.
- Anyone could open any customer's uploaded **ID document** by guessing a short web address — no login.

How we rate severity

Critical anyone can, with no special access — including any **signed-up** user (sign-up is open), or a secret that unlocks the back end for anyone holding the code/history. **High** needs a non-public account, or hits one feature or a single targeted record. **Medium** needs a second weakness or many tries. **Low** hygiene.

Honest note on the demo. So this report can be published and reproduced by anyone, Invoicer runs on a **local stand-in** that copies Supabase's API shape (REST, auth, storage, functions) on local SQLite, with a deterministic **mock** in place of a real LLM. The bugs are the same kind we look for in real Supabase apps; they are not identical software. So each finding gives two fixes — the one we shipped on the demo, and the one you would ship on **real Supabase** — clearly labelled. Where the mock matters (F-07) we say so and prove the issue at the tool boundary.

2. Findings

ID	Severity	Area	What a stranger could do	Status
F-01	Critical	Private data	Read and change every user's invoices from any signed-up account	Fixed
F-02	Critical	Secret keys	Use a master key shipped in the website code to bypass all rules	Fixed
F-03	Critical	Payments	Fake a payment to unlock the paid plan for free	Fixed
F-05	Critical	File uploads	Open any customer's uploaded ID document with a guessable link	Fixed
F-06	Critical	Admin pages	Read every user's email and the revenue total with no login	Fixed
F-10	Critical	Sessions / keys	Forge a login as any user or the admin using a signing secret left in the code	Fixed
F-04	High	Payments	Set their own price at checkout	Fixed
F-07	High	AI features	Make the assistant fetch a named user's invoices	Fixed
F-08	Medium	Logins	Guess passwords without limit; reuse a reset link that never expires	Fixed
F-09	Low	Private data	See internal error details (stack traces)	Fixed

Totals: 6 Critical, 2 High, 1 Medium, 1 Low. 10 found, 10 fixed, 0 open.

Reproduce any proof below. In `demo-app/` : `ANON` is the public anon key (`grep VITE_SUPABASE_ANON_KEY .env`), sent as the `apikey` header; `$ALICE` / `$BOB` are session tokens from logging in the seeded demo users (passwords in `demo-app/README.md`). The one-command version is `./prove.sh` (appendix).

F-01 **Critical** Private data — any signed-up user reads and edits everyone's invoices

STRANGER Sign up for a free account, then read and change every other customer's invoices.

PROOF

```
$ curl "http://127.0.0.1:8787/rest/v1/invoices?select=id,owner_id,description" \
-H "apikey: $ANON" -H "authorization: Bearer $ALICE"
[{"id":1,"owner_id":"1111...", "description":"Logo design - Alice"},
{"id":3,"owner_id":"2222...", "description":"Promo video - Bob (confidential)"},
{"id":5,"owner_id":"3333...", "description":"Brand package - Carol"}]

$ curl -X PATCH "http://127.0.0.1:8787/rest/v1/invoices?id=eq.3" \
-H "apikey: $ANON" -H "authorization: Bearer $ALICE" \
-H 'content-type: application/json' -d '{"status":"cancelled"}'
{"updated":1}
```

WHY The invoices rule let any signed-in user read all rows (the `using (true)` pattern), and the update path never checked ownership (an IDOR). The check lived in the screen, not the server.

FIX (DEMO)

```
- select: (ctx) => ctx.role === 'authenticated' ? { allowed: true, filter: null } : ...
- update: (ctx) => ctx.role === 'authenticated' ? { allowed: true, filter: null } : ...
+ select: ownBy('owner_id'),
+ update: ownBy('owner_id'),
```

ON REAL SUPABASE

```
alter table invoices enable row level security;
create policy invoices_owner on invoices for all
  using (auth.uid() = user_id) with check (auth.uid() = user_id);
```

WHY THIS SEVERITY Critical — sign-up is open to the public, so "any logged-in user" is effectively anyone, and this exposes and alters every user's data.

HUMAN-REPRODUCED yes, 29 Oct 2026.

RETEST **BLOCKED (29 Oct 2026).** `git checkout fixed && ./prove.sh` → F-01a ...

BLOCKED , F-01b ... **BLOCKED** . Alice now lists only her own rows (`[{"id":1,...}, {"id":2,...}]`); her **PATCH** of Bob's #3 returns `{"updated":0}` .

F-02 Critical Secret keys — master key shipped in the website code

STRANGER Read a `service_role` master key out of the downloaded JavaScript and use it, with no login, to read or change any data.

PROOF

```
$ curl "http://127.0.0.1:8787/rest/v1/profiles?select=email,role" \
-H "apikey: eyJhbG...[redacted service_role key]"
[{"email":"alice@example.test","role":"user"},
{"email":"bob@example.test","role":"user"},
{"email":"admin@example.test","role":"admin"}, ...]
```

WHY The key was put in a `VITE_`-prefixed variable, so it was baked into the browser bundle at build time — and the API accepted it over the public internet.

FIX (DEMO) Stop shipping it, and stop honouring it on the public API:

```
- const SERVICE_ROLE_KEY = import.meta.env.VITE_SUPABASE_SERVICE_ROLE_KEY; // frontend:
  removed
-   return { role: 'service_role', userId: null }; // server honoured it
+   return { role: 'anon', userId: null };         // treated as an ordinary anon request
```

ON REAL SUPABASE Supabase *a/ways* accepts the `service_role` key — so you cannot stop honouring it. **REMOVE IT FROM THE FRONTEND, ROTATE IT** (it was public), and move any admin call that needed it into an **EDGE FUNCTION** that holds the key server-side. The demo's extra server-side rejection is a demo-only belt-and-braces.

WHY THIS SEVERITY Critical — a secret that grants full back-end access to anyone who downloads the shipped code.

HUMAN-REPRODUCED yes, 29 Oct 2026.

RETEST **BLOCKED (29 Oct 2026).** `./prove.sh` → F-02 ... BLOCKED . The rebuilt bundle contains only the public `anon` key; the request now returns `{"message":"not allowed by policy"}` . In production the key must still be rotated, because it was public.

F-03 Critical Payments — fake a payment to unlock the paid plan

STRANGER Send one request that looks like a "payment complete" message and flip any account to paid, having paid nothing.

PROOF

```
$ curl -X POST "http://127.0.0.1:8787/functions/v1/stripe-webhook" \  
-H 'content-type: application/json' \  
-d '{"type":"checkout.session.completed","data":{"object":{"client_reference_id":"2222...  
(Bob)"}}}' \  
{ "received":true,"upgraded":"2222..." } # Bob is now on the paid plan
```

WHY The webhook trusted the JSON body and never verified the signature against the raw body with the signing secret.

FIX (DEMO) Require the signing secret from the environment (no hardcoded fallback), verify the signature, and only upgrade when the plan and amount match the server price:

```
+ const secret = process.env.STRIPE_WEBHOOK_SECRET; // no fallback  
+ if (!verifySignature(bodyStr, signature, secret)) return 400; // forged -> rejected  
+ if (Number(session.amount_total) !== PRICES[plan]) return 400; // tampered amount ->  
  rejected
```

ON REAL SUPABASE In the webhook Edge Function, verify with the `RAW` body: `await stripe.webhooks.constructEventAsync(await req.text(), req.headers.get('stripe-signature'), Deno.env.get('STRIPE_WEBHOOK_SECRET'))`, then check `amount_total` and the plan against your price table.

WHY THIS SEVERITY Critical — anyone, with no account, can grant paid status to any account (financial impact).

HUMAN-REPRODUCED yes, 29 Oct 2026.

RETEST **BLOCKED (29 Oct 2026).** `./prove.sh` → F-03 ... **BLOCKED** : the forged event returns `{"message":"signature verification failed"}` (400). The script also sends a correctly-signed event with the right amount (C-2 ... OK (upgraded) — real payments still work) and one with a tampered amount (C-3 ... OK (rejected)) — tested both ways.

F-05 **Critical** File uploads — anyone can open any customer's ID document

STRANGER Open any customer's uploaded ID document by guessing a short, predictable web address. No login.

PROOF

```
$ curl "http://127.0.0.1:8787/storage/v1/object/public/id-documents/2.txt" (no
credentials)
DEMO ID DOCUMENT – Bob Demo – license no. B7654321 (fake)
```

WHY The bucket was public and file names were sequential (1.txt , 2.txt ...), so the link was the only protection — and links leak.

FIX (DEMO) Require the caller to be logged in and to own the file:

```
-function handleStorage(req, res, url) {
+function handleStorage(req, res, url, ctx) {
+  if (ctx.role !== 'authenticated' || ctx.userId !== row.owner_id)
+    return sendJson(res, 403, { message: 'not allowed' });
```

ON REAL SUPABASE Put ID documents in a **PRIVATE** bucket, scope `storage.objects` to the owner, and serve **SHORT-LIVED SIGNED URLS**:

```
create policy id_docs_owner on storage.objects for select
  using (bucket_id = 'id-documents'
        and (storage.foldername(name))[1] = auth.uid()::text);
-- serve via createSignedUrl(path, 60), never a public URL
```

WHY THIS SEVERITY Critical — anyone, unauthenticated, can read any user's identity document (sensitive PII) with a guessable URL.

HUMAN-REPRODUCED yes, 29 Oct 2026.

RETEST **BLOCKED (29 Oct 2026).** `./prove.sh` → F-05 ... BLOCKED , F-05b ... BLOCKED . The anonymous request returns 403; a logged-in **NON-OWNER** (Alice) fetching Bob's exact path also gets `{"message": "not allowed"}` (403); only Bob can read his own file.

F-06 Critical Admin pages — admin stats open to anyone

STRANGER Read every user's email, plan and the total revenue, with no login. The admin page was hidden in the menu only.

PROOF

```
$ curl "http://127.0.0.1:8787/rest/v1/admin_stats" (no credentials)
{"users":4,"paid_accounts":3,"total_revenue_cents":994900,"accounts":[ ...every email... ]}
```

WHY The endpoint checked nothing on the server; only the UI hid the link.

FIX (DEMO)

```
+ const me = db.prepare('SELECT role FROM profiles WHERE id=?').get(ctx.userId);
+ if (!me || me.role !== 'admin') return sendJson(res, 403, { message: 'admin only' });
```

ON REAL SUPABASE Enforce the role in RLS or an Edge Function — never in the UI — e.g. gate the policy on `(auth.jwt() ->> 'role') = 'admin'` or a membership row. Hiding the menu link is not a control.

WHY THIS SEVERITY Critical — anonymous access to every user's email and the revenue total.

HUMAN-REPRODUCED yes, 29 Oct 2026.

RETEST **BLOCKED (29 Oct 2026).** `./prove.sh` → F-06 ... BLOCKED (returns `{"message":"admin only"}`, 403, unless signed in as admin). Control C-1 ... BLOCKED : a user cannot promote themselves by writing `profiles.role` — `role` is not a writable column.

F-10 **Critical** Sessions — the login-signing secret is left in the code

STRANGER Read the secret that signs login sessions out of the app's source (and git history), then forge a valid login as any user or the admin — no password.

PROOF The HS256 signing secret is committed in `server/config.js` (and in `.env` in git history). With it, mint a session for any user id; against an endpoint that needs a real session:

```
$ curl -X POST "http://127.0.0.1:8787/functions/v1/assistant" -H "apikey: $ANON" \
-H "authorization: Bearer <session forged for Bob with the leaked secret>" \
-d '{"message":"list my unpaid invoices"}'
{"reply":"You have 2 invoice(s).","used_user_id":"2222...(Bob)","data":[ ...Bob's 2 invoices...
]}
```

NOTE A token forged for the admin likewise opens the admin endpoint. On `master` that endpoint is also open via F-06; the point of F-10 is that the forged **SESSION** is trusted, which would defeat the F-06 fix too unless the secret is rotated.

WHY `config.js` set `jwtSecret` to a hardcoded literal fallback, and the same value is in `.env` in history. Sessions are signed **AND VERIFIED** with it, so the secret is the whole lock.

FIX (DEMO) Remove the committed secret; take it from the environment and rotate it, so tokens signed with the old secret stop verifying:

```
- jwtSecret: process.env.JWT_SECRET || 'super-secret-...[redacted committed signing secret]',
+ jwtSecret: process.env.JWT_SECRET || crypto.randomBytes(32).toString('hex'), // env-only, rotated
```

ON REAL SUPABASE Never put the project's JWT secret or `service_role` key in `VITE_`/client code or git. Keep them in Edge Function secrets / project config, and if one leaks,

ROTATE THE JWT SECRET IN THE DASHBOARD — that invalidates every existing token, including any forged.

WHY THIS SEVERITY Critical — a secret in the shipped code and public history that lets anyone forge any user or admin session (full back-end access).

HUMAN-REPRODUCED yes, 29 Oct 2026.

RETEST **BLOCKED (29 Oct 2026).** `./prove.sh` → F-10a ... **BLOCKED** (forged admin token → 403 at `admin_stats`), F-10b ... **BLOCKED** (forged Bob token → `{"message":"login required"}`). A genuine admin login still works.

Also assessed: the Stripe secret key (not a finding). `.env` also holds a `STRIPE_SECRET_KEY`. We checked it and did not file it: it is a test-mode `sk_test_...` placeholder containing `FAKE`, referenced by no code, and absent from the bundle — inert and not exploitable here. (On a real app we would rate a committed live `sk_live_...` key Critical and a committed `sk_test_...` key High, and rotate regardless.)

F-04 High Payments — the buyer sets their own price

STRANGER Choose their own price at checkout — for example, start a "pro" checkout for 1 cent.

PROOF

```
$ curl -X POST "http://127.0.0.1:8787/functions/v1/create-checkout" \
  -H "apikey: $ANON" -H "authorization: Bearer $ALICE" \
  -H 'content-type: application/json' -d '{"plan":"pro","amount":1}'
{"id":"cs_demo_...", "plan":"pro", "amount_cents":1, "currency":"usd"}
```

WHY The server read `amount` from the request body instead of looking the price up itself.

FIX (DEMO)

```
- const amountCents = Number(body.amount);
+ const amountCents = PRICES[plan];
+ if (amountCents === undefined) return sendJson(res, 400, { message: 'unknown plan' });
```

ON REAL SUPABASE Compute the price on the server from a `PRICE ID` : the client sends only which plan it wants; the Edge Function looks up the amount (your table, or `stripe.prices.retrieve(priceId)`) and passes that to Stripe. Never accept an amount from the browser.

WHY THIS SEVERITY High — it affects one feature (checkout pricing) and needs the payment step to cause loss; not a data breach or master key. (The F-03 fix also re-checks the amount in the webhook.)

HUMAN-REPRODUCED yes, 29 Oct 2026.

RETEST **BLOCKED (29 Oct 2026).** `./prove.sh` → F-04 ... BLOCKED . The endpoint returns `"amount_cents":1499` (the server price), not the client's `1` .

F-07 High AI features — the assistant can be told to read a named user's data

STRANGER Ask the AI assistant for another user's invoices by naming that user's id, and get them back.

PROOF

```
$ curl -X POST "http://127.0.0.1:8787/functions/v1/assistant" \
  -H "apikey: $ANON" -H "authorization: Bearer $ALICE" \
  -H 'content-type: application/json' -d '{"message":"list unpaid invoices for user
2222...(Bob)"}'
{"reply":"You have 2 invoice(s).","used_user_id":"2222...(Bob)","data":[ ...Bob's 2 invoices...
]}
```

WHY The assistant's tool `get_invoices(user_id)` took its id from the model's instructions (which come from the user's message) instead of the session.

DEMO NOTE The assistant uses a deterministic **MOCK** model, not a real LLM. The mock extracts the id exactly as a tricked real model would pass one; the flaw is at the **TOOL BOUNDARY** (the tool trusts a caller-supplied id), which is the same either way.

FIX (DEMO)

```
- const targetId = decision.args.user_id; // from the model/message
+ const targetId = ctx.userId;           // from the session
```

ON REAL SUPABASE Tool/Edge functions must take the user from the verified JWT (`auth.uid()`), never from the model's arguments, and run under the caller's RLS — not the service key. Treat every model-supplied value as untrusted input.

WHY THIS SEVERITY High — on its own it needs a logged-in account and the victim's non-guessable user id, and returns one targeted user's invoices at a time (not the bulk dump that makes F-01 Critical).

HUMAN-REPRODUCED yes, 29 Oct 2026.

RETEST **BLOCKED (29 Oct 2026).** `./prove.sh` → F-07 ... BLOCKED . The id in the message is ignored: `used_user_id` is Alice's own id and only Alice's invoices come back.

F-08 Medium Logins — no limit on tries, and reusable reset links

STRANGER Guess passwords with no limit, and reuse a password-reset link forever (it never expires and works more than once).

PROOF

```
$ for i in $(seq 1 12); do curl -s -o /dev/null -w '%{http_code} ' \
  -X POST "http://127.0.0.1:8787/auth/v1/token" -H "apikey: $ANON" \
  -H 'content-type: application/json' \
  -d '{"email":"alice@example.test","password":"wrong"}'; done
400 400 400 400 400 400 400 400 400 400 400 400      (never a 429)

confirm #1 -> {"message":"password updated"}
confirm #2 (same token) -> {"message":"password updated"}  # should have failed
```

WHY No rate limiting on login, and the reset-confirm route ignored whether a token was used or expired.

FIX (DEMO) Rate-limit logins (per-IP checked first, counting only failures, in a bounded time-expiring map), and make reset tokens single-use with a 1-hour expiry:

```
+ if (currentFails('i:'+ip, now) >= MAX_FAIL_IP) return sendJson(res, 429, {...}); // per-IP first
+ // count only FAILED attempts per account; a correct password resets the counter
+ if (row.used) return sendJson(res, 400, { message: 'token already used' });
+ if (Date.now() - Date.parse(row.created_at) > RESET_TTL_MS) return sendJson(res, 400, {...});
```

ON REAL SUPABASE Supabase Auth already rate-limits and ages reset tokens; for any custom auth path the lesson is the same — throttle failed logins (and add MFA), and make reset tokens single-use and short-lived.

NOTE An account over its limit gets a **GROWING DELAY**, NOT A FLAT LOCKOUT, so a real user with the correct password is never locked out by someone else's failed guesses.

WHY THIS SEVERITY Medium — needs many attempts (guessing) or a separately-leaked reset link (a second weakness), not a one-request breach.

HUMAN-REPRODUCED yes, 29 Oct 2026.

RETEST **BLOCKED (29 Oct 2026).** `./prove.sh` → F-08a ... BLOCKED , F-08b ... BLOCKED . Repeated wrong logins return 429; a reused token returns `{"message":"token already used"}`; a correct password still logs in after some failures.

F-09 Low Private data — error pages leak internal details

STRANGER Trigger an error and read the server's stack trace and failing query — a map of how the app is built.

PROOF

```
$ curl "http://127.0.0.1:8787/rest/v1/invoices?select=secret_column" \
  -H "apikey: $ANON" -H "authorization: Bearer $ALICE"
{"message":"no such column: secret_column","stack":"SQLiteError... at server/server.js:..."}

```

FIX (DEMO) Log the detail on the server; return a generic message:

```
- return sendJson(res, 500, { message: err.message, stack: err.stack });
+ console.error('[invoicer] request error:', err && err.stack);
+ return sendJson(res, 500, { message: 'internal server error' });

```

ON REAL SUPABASE Same rule in an Edge Function: log the detail to the function logs, return a generic message/opaque id to the client, and keep verbose errors off.

WHY THIS SEVERITY Low — hygiene; on its own it exposes no user data or money, it mainly makes other attacks easier.

HUMAN-REPRODUCED yes, 29 Oct 2026.

RETEST **BLOCKED (29 Oct 2026).** `./prove.sh` → F-09 ... BLOCKED . The request now returns only `{"message":"internal server error"}` .

What scanners found vs. what they missed

A common question: *"I already run a scanner — isn't that enough?"* We ran the standard ones against Invoicer.

Tool	What it actually reported	What it missed
gitleaks (git history)	4 hits, all tagged "JWT": the anon key and the service_role key, in <code>.env</code> and <code>config.js</code>	That one is a <code>service_role</code> master key shipped to browsers (F-02). It did not flag the HS256 signing secret (F-10 — not JWT-shaped) or the <code>sk_test_</code> Stripe placeholder.
semgrep (js/node/react)	Nothing	Everything
semgrep (--config auto)	The same two keys as literals	Every logic flaw; the meaning of the keys
osv-scanner	18 advisories in build-time deps (<code>vite</code> , <code>esbuild</code>) — not shipped	All of the app's own problems (not its job)
trufflehog	Nothing (reports only keys it can verify live; these are fake)	Everything
A person + specialist AI reviewer agents	All 10 findings	—

Precisely: of the 10 findings, scanners partially surfaced the strings behind one (F-02 — as an unlabelled "JWT") and missed the substance of all ten. Not one decoded that the flagged string was a master key handed to every visitor; not one flagged the signing secret behind F-10 or a single broken access rule, forged-payment path, exposed file, open admin endpoint or AI-tool problem. The other nine were found only by a person testing the running app. In apps built with AI tools the dangerous bugs are in the **logic** — who is allowed to do what — and that is exactly what pattern-matching scanners cannot see. A scanner subscription is not a substitute for a review.

What was not tested, and next steps

Not tested

- **Real hosted services.** Invoicer runs on a local stand-in (SQLite + a mock model) that copies Supabase's API shape, so it runs offline and can be published; the bugs are the same kind we look for on real Supabase, and each finding lists the real-Supabase fix.
- **The browser front end itself** (XSS, CSP depth, cookie flags) beyond key handling.
- **Load / denial-of-service and anything destructive.** Out of scope by policy (the rate-limit check sent 12 requests, not a flood).

Accepted risks / known limits

- Dependency advisories in build tools (not shipped to users) — accepted; upgrade at the next routine bump.
- Stripe secret key in `.env` — assessed, not filed: inert `sk_test_` placeholder, used by no code, absent from the bundle.
- Mock assistant model — F-07 is proven at the tool boundary; a real model would not change the fix.
- Login throttle is in-process; a multi-server deployment should move the counters to shared storage and add a second factor.

Next steps (if this were your app)

1. Deploy the 10 fixes (in `demo-app/fixes/`, applied on the `fixed` branch) and re-run the checks.
2. Rotate the leaked secrets — the `service_role` key (F-02), the session signing secret (F-10), and anything ever committed to git must be rotated, because they were public; rotating the JWT secret also kills any forged token.
3. Turn on the platform's own guards: row-level security on every table, confirm-email on sign-up, provider spend caps, weekly security-advisor alerts.
4. Re-review after the next big release — new code from AI coding tools brings new logic, and logic is where these bugs live.

Guarantee (a real engagement). Report and ready-to-merge fixes within 48 hours of access, a human sign-off on every finding (named on your copy of the agreement), and one free retest within 30 days.

Security summary for investors (Review Plus example)

This one-page summary is the extra deliverable in a Review Plus, written to be shown to a non-technical investor or customer during diligence. Example, on a demo app.

REVIEWER Bardstar · DATE 29 October 2026

SIGNED OFF BY Bardstar reviewer (named on the client's copy of the agreement)

SCOPE Full application code, git history, and a running test copy on an isolated machine.

What was done. An independent review for the classes of problem most common in apps built with AI coding tools: who can read or change other users' data, whether payments can be faked, whether secret keys or session secrets leak, logins, file access, AI-feature abuse, usage limits, and admin access. Every problem was confirmed by a real test, then fixed, then re-tested.

Severity	Found	Fixed	Open
Critical	6	6	0
High	2	2	0
Medium	1	1	0
Low	1	1	0
Total	10	10	0

Controls now in place: per-owner access to invoices and files; no master key in the browser; the session-signing secret out of the code and rotated (forged logins no longer verify); signature- and price-verified payments; owner-only access to uploaded ID documents; server-side admin-role checks; the AI assistant limited to the signed-in user's own data; rate-limited logins and single-use, expiring password-reset links.

Retest status: all 10 findings re-tested after fixes on 29 October 2026 — every one blocked. 0 open.

Reviewer statement. This review covered the most common and most damaging classes of web-application and AI-feature security problems for this app, and all identified issues were fixed and re-tested. **This is not a certification**, not a guarantee that no other issue exists, and not a substitute for

ongoing security work. It is a point-in-time, good-faith review of the code and behaviour described in the scope.