Web Application Penetration Test - Redacted

Black-box security assessment of an Argentine e-commerce platform (Next.js + Supabase + MercadoPago)

Client	[Redacted]
Target	Argentine e-commerce (storefront, admin panel, WhatsApp bot, database)
Engagement type	Black-box web application penetration test
Tester	Tomás Gorrini Rech
Effort	Approximately 24 work hours
Report date	July 7, 2026
Status	Reported issues remediated; published with the client's written consent
Version	1.0 (public, sanitized)

About this writeup

This is a sanitized, anonymized version of a private penetration-test report, published as a portfolio case study. The client's identity, domains, database identifiers, API keys, customer data, and step-by-step exploitation payloads have been removed. Findings are described at the level of the bug class, the impact, and the fix (the pattern rather than a reproducible attack map).

It was published only after the reported issues were fixed and with the client's written sign-off. Everything below describes the platform as it was during testing, not as it is now.

---

1. Executive summary

1.1. Context and objective

The client, an Argentine e-commerce operator, engaged me to attack their own platform before someone else did. The target was the full stack that a shopper and an operator actually touch: the online store, the admin panel, the WhatsApp bot that answers customers, and the database underneath all of it.

I worked black-box (no admin login, no staff account, no staging copy). I started with exactly what any shopper gets, a registered customer account, and worked outward against the live production site. That framing matters when reading the results, because everything below was reachable from the open internet, without any credential the business handed me.

1.2. Scope

Every asset here belongs to the client:

Ref.	Asset
A	Online storefront
B	Administration panel
C	WhatsApp service (bot)
D	API routes (`/api/*`) of A, B, and C
E	Database via its public REST API (Supabase PostgREST)
F	Owned-infrastructure subdomains of the client's domain

Out of scope were the providers themselves (the Vercel, Supabase, and Hetzner account panels), the third-party platforms the app integrates with (MercadoPago, MercadoLibre, and the shipping carriers, of which I tested only how the client's own code uses them, never their security as services), and real customer data (I pulled the minimum needed to prove each issue and never downloaded the database).

1.3. Severity scale

Severity	What it means
Critical	Bulk customer data, the ability to move or fake money, code execution, or an illegitimate jump to admin. Fix now.
High	Other users' data, sensitive endpoints with no authentication, or flaws that read or change private business data.
Medium	Business-logic abuse (coupons, prices), no attempt limits, or exposure of lower-sensitivity data.
Low	Missing configuration best practices and minor information leaks. Not exploitable alone, but they raise overall risk.
Informational	Hardening notes, with no direct risk.

Where it applies, I attach a CVSS 3.1 score (the industry's 0-to-10 severity standard) and its vector.

1.4. The bottom line

At the time of testing, the platform faced critical security gaps that needed immediate fixing. I logged 39 findings (13 Critical and 8 High). From the open internet, with no password and no insider help, I confirmed that an attacker could:

- **Read every customer's and prospect's personal data** (names, phones, emails, and the full text of their private WhatsApp conversations).
- **Set an order's price to any value, including a negative total.** I created a real order for minus two thousand pesos using an ordinary 5% coupon.
- **Mark orders as paid without paying,** and trigger MercadoPago payment reconciliation.
- **Rewrite the store's inventory** (zeroing products out to block sales, or inflating them into overselling).
- **Enter the admin panel with an ordinary customer account.**

None of this required a stolen credential. A curious shopper with browser developer tools could have performed most of it. I confirmed the issues live, with the smallest possible test, and stopped the moment each one was proven.

In business terms, that is three problems at once: a reportable personal-data breach under Argentina's Data Protection Act (Law 25.326), a direct path to payment fraud and lost revenue, and a free intelligence feed (margins, ad spend, ROI) for any competitor who found the same URLs.

1.5. Findings by severity

Severity	Count
Critical	13
High	8
Medium	6
Low	5
Informational	7
Total	39

1.6. Top risks

In plain terms, the highest-priority risks (the `F-xx` tag points to the technical detail later):

- **The customer and prospect database was openly readable (F-06, F-07, F-17, F-26).** Roughly 300 WhatsApp prospects with name and phone, the customer roster with email and phone, every user profile (including which user is the admin), and the full text of private WhatsApp conversations, all readable from the internet with no password.
- **Money and payments could be faked (F-19, F-32, F-33, F-36).** Requesting a MercadoPago reconciliation, marking orders paid or unpaid, and (the one I proved live)

creating orders at any price, including negative.

- **A shopper could become staff (F-09).** Reusing a customer's session cookies on the panel domain granted access to the admin interface: viewing sales, editing and deleting products, wiping the bot's knowledge base, reading prospects.
- **Inventory was writable anonymously (F-24).** No login was needed to zero out stock or inflate it.
- **The WhatsApp bot could be driven by outsiders (F-38).** The routes that make it reply in bulk or send promotions to the group required no authentication.
- **The company's finances leaked (F-18).** MercadoLibre billing, ad spend, revenue, profit, and ROI were all queryable without logging in.
- **The entire coupon table was readable and editable (F-23).** All active coupons and their rules, plus a confirmed case of one customer editing another customer's coupon.

1.7. Priority recommendations

1. **Lock down the panel and its API routes.** One central check on every panel route and API that verifies not just a session but a staff role. This alone removes a large share of the Critical findings.
2. **Fix the database access rules (RLS) in Supabase.** No table with customer data, coupons, or configuration should be readable or writable with the public key.
3. **Stop trusting the browser on money.** Recompute prices, discounts, and totals server-side, and take payment status only from MercadoPago's signed confirmation.
4. **Update Next.js** to the patched release (13 known CVEs are associated with the version in use).
5. **Rotate secrets and clean up the leftover test data,** per the private cleanup notes.

The rest of this document is written for the people who will fix these issues. It gets technical from here.

---

2. Methodology

I ran two lanes in parallel so they would not interfere with each other. The first lane was the quiet mechanical sweep: endpoint enumeration, version fingerprinting, an access-control sweep across every route, secret hunting in the JavaScript, and configuration hygiene. The second lane was hands-on manual testing: forms, injection, and business logic, all of it flowing through Caido (an intercepting proxy) so that every request landed in one place I could search and replay.

The goal was coverage rather than a single trophy. I walked a full checklist (access control, database rules, webhooks and payments, business logic, authentication and session, injection, file upload, information exposure, personal data, and the WhatsApp service) instead of tunneling on the first interesting bug.

Reconnaissance. Certificate-transparency logs and DNS gave me the asset map. One early correction came out of it: the panel was not on the subdomain the rules of engagement assumed (that hostname did not resolve at all), and the WhatsApp service was on its own subdomain. I then took the frontend JavaScript bundles apart, which is where the real map lived. About 45 panel endpoints fell out of the authenticated JavaScript, along with the framework and its exact version, which I cross-referenced against known CVEs. I also searched the bundles for leaked secrets; the only thing that surfaced was Supabase's public `anon` key, which is meant to be public. No privileged key leaked.

Tooling. Caido as the single source of traffic, plus `curl`, `dig`, a good deal of manual bundle reading, and direct queries against Supabase's public REST API with the `anon` key. I kept everything at a human pace (under five requests per second, no floods), because this was a live business rather than a lab.

What I deliberately did not do. No denial of service at scale (where I found an availability bug, I reproduced it twice and stopped). No real money moved (test orders went through as "cash" so no charge fired). No bulk data pulls (one redacted record per exposure, then stop). And no third-party credentials at any point; the only key I used was the public `anon` one.

Coverage notes. Access control, RLS, webhooks and payments, injection, file upload, information exposure, and personal data all received full coverage. Authentication is OTP by email only (there is no password login and no reset flow to attack, so those branches do not apply), with one residual item: I did not confirm session-token invalidation on logout. On business logic, coupons and price tampering are covered, the stock race condition tested clean (section 7), and the abandoned cart voucher flow is the one item I did not reach.

---

3. Root causes

Before the finding-by-finding detail, the honest summary is that this is not 39 unrelated bugs. It is a few mistakes repeated across the codebase. Fix these three patterns and most of the list collapses.

1. The panel checked whether you were logged in, never who you were. The panel's routing layer asked one question (is there a valid session?) and never asked for a role. A normal store customer's session is a valid session, so a normal customer got in. More seriously, most of the panel's API routes asked nothing at all and answered anyone.

I will admit I chased this one down the wrong path first. My instinct was that the bug lived in the middleware, so I spent time trying to slip past it (`.png` tricks, double slashes, `%2e%2e`, `.json`), and it held every time. The middleware was fine. The real problem was that it was only ever wired to guard pages, so the `/api/*` routes sat behind it completely uncovered, and a direct call to one passed straight through. Once that became clear, half of the Critical findings followed naturally.

2. The database was on the internet, trusting row-level rules that were not there. Supabase exposes tables over a REST API, and the only thing between "public" and "private" is each table's row-level security (RLS) policy. On several key tables (profiles, discount codes, pricing configuration), those policies were missing or wrong, so the public key could read (and in one case write) data that should never have left the server.

3. The server believed the browser about money. Prices, discounts, payment flags, and quantities were sent by the client and used by the server with no recomputation. That is how I produced a negative order total with a legitimate coupon.

Every Critical and High below is really one of these three wearing a different hat.

---

4. Findings summary

ID	Title	Severity	Asset
F-06	Missing RLS on the user-profiles table: PII and roles readable with no session	Critical (7.5)	Database (Supabase)
F-07	Missing authentication on the prospect-listing endpoint: PII dump of 300 prospects	Critical (7.5)	Panel
F-09	Broken access control and privilege escalation: a customer session reaches the admin panel	Critical (8.7 to 9.0)	Panel
F-17	Missing authentication on the customer-search endpoint leaks customer PII	Critical (7.5)	Panel
F-19	Missing authentication on the payment-reconciliation endpoint: MP payments confused deputy	Critical (9.1)	Panel
F-23	Open RLS and write IDOR on the discount-codes table	Critical (8.1)	Database
F-24	Missing authentication on the stock-adjust and stock-transfer endpoints	Critical (8.2)	Panel
F-26	IDOR and missing authentication on the prospect-detail endpoint: WhatsApp conversation leak	Critical (7.5)	Panel
F-32	Missing authentication and confused deputy on the storefront payment-verification endpoint	Critical (9.1)	Storefront
F-33	Missing authentication on the order payment-status endpoint: payment-status tampering	Critical (8.2)	Panel

ID	Title	Severity	Asset
F-34	IDOR on the shipping-label endpoints: recipient PII exposure	Critical (7.5)	Panel
F-36	Price tampering on the order-creation endpoint: arbitrary discount yields a negative total	Critical (7.7)	Storefront
F-38	Missing authentication on the WhatsApp bot-action endpoints	Critical (7.7)	Panel
F-11	Mass assignment on the panel order-creation endpoint (price and payment status)	High (6.5)	Panel
F-12	Stored prompt injection via the bot knowledge-base endpoint	High (6.5)	Panel and bot
F-13	Mass assignment on the product-management endpoint (price, cost, markup client-controlled)	High (6.5)	Panel
F-14 & F-35	Path traversal and arbitrary file write (unauthenticated) on the image-upload endpoint	High (7.3)	Panel
F-16	Outdated Next.js with known CVEs (13 advisories)	High (7.5)	Storefront and panel
F-18	Missing authentication on multiple panel routes: business finances exposed	High (7.5)	Panel
F-30	Pre-auth denial of service via a self-referential `next-url`	High (7.5)	Storefront
F-08	Open RLS: business data readable by `anon`	Medium (5.3)	Database
F-15	No real lockout on OTP verification (CWE-307)	Medium (5.3)	Authentication
F-20	No origin validation on the MercadoLibre webhook endpoint	Medium (5.3)	Panel
F-22	Missing authentication and denial of wallet (economic DoS) on the AI-assistant endpoint	Medium (5.3)	Storefront
F-25	Missing role check on the MercadoLibre item-inspect endpoint	Medium (4.3)	Panel
F-27	Client-side field lock in the profile (CWE-602), stored XSS candidate	Medium (4.3)	Storefront
F-31	Session cookie without `HttpOnly` or `Secure` (CWE-1004, CWE-614)	Low (5.3)	Storefront
F-02	Missing security headers	Low (4.3)	Storefront and panel

ID	Title	Severity	Asset
F-10	Verbose error messages (PostgREST)	Low	Panel
F-28	Bot health endpoint leaks session state	Low (5.3)	Bot
F-29	Secret in the URL (`?key=` , CWE-598) on the bot	Low (3.7)	Bot
F-21	Attack surface: WhatsApp service exposed to the internet	Low	Bot
F-01	Scope correction: the panel is on a different subdomain than assumed	Informational	DNS
F-03	CORS: `Access-Control-Allow-Origin: *` on static assets	Informational	Panel
F-04	Stack disclosure via response headers	Informational	Storefront and panel
F-05	Supabase `anon` key in the bundle (by design)	Informational	Storefront and DB
F-37	Schema enumeration via the `PGRST205` hint	Informational	Database
F-39	Missing `security.txt` and an indexable panel	Informational	All

---

5. Detailed findings

Grouped by severity, highest to lowest. Each keeps its `F-xx` tag. I have stripped the literal routes, real identifiers, keys, and customer data for this public version, but kept the HTTP status codes and structural evidence where they carry the point.

5.1. Critical

F-06: Missing RLS on the user-profiles table

- Critical. CVSS 3.1: 7.5 (`AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N`).
- Asset: database, Supabase PostgREST, the user-profiles table.
- Status: confirmed (unauthenticated read). I did not test writes, per the stop rule.

The database runs on Supabase, which puts tables on the internet behind a REST API. The only lock on each table is its row-level security policy. The user-profiles table (first name, last name, phone, role, tax ID, tax condition, Instagram, date of birth, notes) had no policy limiting reads to your own row, so an unfiltered `SELECT *` returned every user, including the row with `role: "admin"` .

I first noticed this with an authenticated customer token, which was already bad. Then I removed the token entirely and sent the request with nothing but the public `anon` key, and it still returned the full table. Anyone on the internet could read every user record.

Structurally, the rows looked like this, with an admin sitting right next to the customers:

```
{ "id": "<uuid>", "first_name": "<redacted>", "role": "customer" }
{ "id": "<uuid>", "first_name": "admin",      "role": "admin" }
```

That is a personal-data leak on its own, and it is also a targeting list: once you know which row is the admin, the rest of this report becomes considerably more dangerous. And if the same table also allowed writes from the `authenticated` role with no policy (which I did not test, to stay inside the stop rule), a customer could set their own `role` to `admin`, which would be a real privilege escalation.

Remediation.

1. Enable RLS and scope reads to the caller's own row:

   ```
   ALTER TABLE public.profiles ENABLE ROW LEVEL SECURITY;
   REVOKE ALL ON public.profiles FROM anon, authenticated;
   GRANT SELECT ON public.profiles TO authenticated;
   CREATE POLICY profiles_select_own ON public.profiles
     FOR SELECT TO authenticated
     USING (id = auth.uid());
   ```

2. When a legitimate flow needs to list profiles (the panel, for example), perform that read from a server route with the `service_role` key, never handing the capability to `anon` or `authenticated` over direct PostgREST. If you need to tell staff apart, wrap the role check in a `SECURITY DEFINER` function so it does not recurse on the table.
3. Explicitly confirm there are no open `INSERT`, `UPDATE`, or `DELETE` policies here.
4. Audit the other tables the same way (see F-08 and F-23).

F-07: Missing authentication on the prospect-listing endpoint (300 prospects in one request)

- Critical. CVSS 3.1: 7.5 (`AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N`).
- Asset: panel, the prospect-listing API route (GET).
- Status: confirmed. I kept one redacted record as proof and did not paginate.

This is the clearest example of root cause number one. I sent a plain `GET` with no cookie, no API key, and no bearer token, and it returned HTTP 200 with a JSON document of 300 prospect records. That is the business's entire WhatsApp prospect base, returned to an anonymous request.

What makes it look like an oversight rather than a decision: in the same sweep, the sibling customer-listing route answered `401`. The gating existed; it simply was not applied here.

Route by route, whether a check was present was close to a coin flip.

Each record carried a chat ID, name, phone, status, topic, message count, and contact timestamps:

```
{
  "id": "<id>",
  "name": "<redacted>",
  "phone": "<redacted AR WhatsApp>",
  "status": "<status>",
  "topic": "<topic>",
  "message_count": 5
}
```

I captured one request and one redacted row. The full response (roughly 87 KB) I left alone.

The damage is a bulk personal-data leak (name and WhatsApp number for 300 people, plus what they were shopping for and when they last spoke with the business). That is a ready-made list for spam, smishing, and social engineering, or a valuable one for a competitor. It also chains directly into F-26, which returns the full conversation behind each of those records.

Remediation.

1. Check the session and staff role in the handler before answering. A reusable wrapper (`requireStaff(handler)`) that resolves the session from cookies, reads `profiles.role`, and returns `401` / `403` before any logic runs is the clean pattern.
2. Fix the root cause it shares with F-09, F-17, and F-18: gate the panel's `/api/*` centrally and deny by default (see F-09).

F-09: A customer session opens the entire admin panel

- Critical. CVSS 3.1 (approximate): 8.7 to 9.0 (`AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H`).
- Asset: the whole administration panel.
- Status: confirmed (interface access plus test writes, since reverted).

The panel only ever checked that `getUser()` was not null. It never looked at the role. And the Supabase Auth session cookie is valid across both the storefront and the panel, because they share a parent domain.

Here is what I actually did: I logged in as a plain customer on the storefront, opened developer tools, copied the two session cookies, pasted them onto the panel origin, and reloaded. The dashboard loaded with full access, never showing the login form, which means not even the client-side role check the form performs ever ran. Anyone who signs up as a customer (and signup is automatic, with no human approval) can reach admin-level access by copying their own cookies to another subdomain.

Once inside, I mapped what actually worked versus what was merely visible. Navigation opened everywhere; the table below is where I confirmed real data and real writes:

Section	Confirmed access
Dashboard	Data read
New sale	Read and create (write)
WhatsApp prospects	Full personal-data read and status edit
Bot knowledge base	Create and delete entries (destructive write)
Products	Read, edit, create, delete (full CRUD)
Stock	Data read
Wholesale sale	Submit (write)

Every test write (one sale, one knowledge entry, one product edit, one stock edit) I reported and deleted before closing out, so nothing was left behind.

The serious part is not the reads but the destructive writes: a customer could delete real catalog products or wipe the bot's knowledge base. Chained with F-06 (which identifies the real admin), a shopper account turns into partial but real operational control of the business.

Remediation.

1. Perform the role check centrally, not just on login submit. In the Next.js middleware or a server-side panel layout, resolve the session, read `profiles.role`, and require a staff role on every protected page. A live JWT is not enough.
2. Repeat that check in every panel API handler, not only at the page layer, because middleware does not cover `/api/*` called directly. A `requireStaff()` at the top of each handler makes denial the default.
3. Consider isolating the panel session from the store session. Today one cookie works on both because they share the parent domain; scope the panel cookie's domain, or require a panel-only authentication flow that will not accept store sessions.

F-17: Missing authentication on the customer-search endpoint

- Critical. CVSS 3.1: 7.5 (`AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N`).
- Asset: panel, the customer-search endpoint (GET).
- Status: confirmed. One redacted record kept.

Same pattern as F-07, different route. The customer-search endpoint returned real customer data (email, first name, last name, phone, and a `guest` flag) to a request with no cookie, no bearer token, and no `anon` key. Again the sibling gives it away: the plain customer-listing route returns `401`, but search does not. One valid search term returned 10 records:

```
{
  "id": "<id>",
  "email": "<redacted>",
  "first_name": "<redacted>",
  "last_name": "<redacted>",
  "phone": null,
  "guest": false
}
```


Because it takes a search term, you can walk it (brute-forcing common names and paging through the customer base). That enables re-identification, targeted phishing and smishing, and easy correlation with the F-07 prospects, with the same Law 25.326 exposure attached.

Remediation.

1. Give customer-search the same session and role check the customer-listing route already has, before it queries anything.
2. Fix the root cause with central `/api/*` gating (see F-09).
3. Confirm the underlying customer table's RLS does not allow `anon` reads. The handler is probably using `service_role` server-side, so route gating is the only real defense here.

F-19: Missing authentication on the payment-reconciliation endpoint (confused deputy)

- Critical. CVSS 3.1: 9.1 (`AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N`).
- Asset: panel, the payment-reconciliation endpoint (POST, under `/api/admin/*`).
- Status: confirmed reachable with no authentication, running real logic against MercadoPago. Tested only with a nonexistent payment identifier.

This route lives under `/api/admin/*` and its job is to reconcile a MercadoPago payment against an order. It asked for no credential. I sent an anonymous POST with a fabricated payment identifier, and instead of stopping me with a `401`, the server actually searched MercadoPago's accounts for that payment and returned `404` ("payment not found in any MP account").

That `404` is the whole point. A `401` would mean the door was locked. A `404` means my anonymous request passed the access control and reached the real MercadoPago API. With a genuine approved payment identifier, the expected next step is that it ties the payment to the order and marks it paid.

I could not push it further, because the rules of engagement forbid using a real payment identifier to complete the mutation (that would move money). But I did not need to. An anonymous caller reaching a live payments function and getting it to run is complete proof that access control is broken here, which is why I rated integrity as fully impacted rather than hedging. The real risk is the confused-deputy pattern: if the server does not check that the

payment actually matches this order (exact amount, `external_reference`, collector account), an attacker with one inexpensive real payment could get a much larger order marked as paid without ever paying for it. I confirmed that exact "server trusts the payment blindly" pattern live on the storefront's twin route in F-36.

Remediation.

1. Require a session and admin role on everything under `/api/admin/*`. Nothing there should be anonymous.
2. Take payment status only from MercadoPago's signed webhook or an authenticated server-to-MP query, never from a reconciliation a stranger can trigger.
3. Before marking an order paid, validate the match: exact amount, the `external_reference` embedded in the payment, and the correct collector account.
4. Add idempotency and an audit trail (who and when), and reject retries against an already-paid order.

F-23: Open RLS and write IDOR on the discount-codes table

- Critical. CVSS 3.1: 8.1 (`AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:N`).
- Asset: database, Supabase PostgREST, the discount-codes table.
- Status: confirmed (unauthenticated read, plus a write into another customer's coupon found in the traffic history). Reported during the engagement.

The discount-codes table was fully readable with the public `anon` key (the same missing-RLS problem as F-06). An unauthenticated attacker can read the entire coupon and voucher inventory: code, type, value, minimum spend, max uses, validity, status. My read returned 155 active rows, and 23 of them carried a `customer_email` (personal vouchers tied to a specific customer and their original order).

The reads were bad; the writes were worse. Reviewing the Caido history, I found a real, successful `PATCH` where one authenticated customer account had modified a coupon issued to a different customer (deactivating it, and receiving a `204 No Content` back). The `UPDATE` policy never checked that the coupon belonged to the person editing it, so any customer could write over any row. The specific coupon in that request happened to be expired already, so the change itself had no effect, but the flaw applies to all 155 codes.

Put together, that is three problems:

- **Coupon harvesting with no effort** (all 155 active codes and their rules, ready to abuse at checkout).
- **PII and voucher theft** (23 vouchers link a customer to an order and an email, so an attacker can claim someone else's voucher).
- **Discount sabotage** (reactivating expired coupons, disabling a live campaign, or rewriting a coupon's value, max uses, or expiry).

Remediation.

1. Close the RLS. `anon` should not read this table. Validate codes server-side, by exact code, ideally through a `SECURITY DEFINER` RPC that takes a code and returns only whether it is valid for that user (no `SELECT *`):

```
ALTER TABLE public.discount_codes ENABLE ROW LEVEL SECURITY;
REVOKE ALL ON public.discount_codes FROM anon, authenticated;
-- Validation goes through a controlled function only:
-- CREATE FUNCTION validate_discount(code text) RETURNS ... SECURITY
DEFINER ...
```

2. Block `UPDATE` and `DELETE` entirely for `anon` and `authenticated`. Every change (activate, deactivate, increment a use counter) goes through a server route with `service_role` that checks role and ownership.
3. If some legitimate `UPDATE` is truly needed, require `customer_email = auth.email()`.
4. Keep `customer_email` and the source-order reference out of any client-side read.
5. Rotate or invalidate the active generic codes, assuming they are already harvested.

F-24: Missing authentication on the stock endpoints (anonymous inventory writes)

- Critical. CVSS 3.1: 8.2 (`AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:H/A:L`).
- Asset: panel, the stock-adjust and stock-transfer endpoints (POST).
- Status: confirmed reachable with no authentication, down to the write layer. Tested with an empty body only, so no stock row was written.

Both stock-mutation routes skipped authentication, so an anonymous POST reached the write logic. I sent stock-transfer an empty body and it did not reject me at the door; it attempted the write and only failed on a database constraint (a not-null column complaining). That failure is the tell: my attacker-controlled input reached the database layer with no access check in between. With a valid body, an anonymous caller adjusts or transfers stock at will.

The empty-body POSTs returned a `500` (handler reached, input invalid) rather than a `401`. I never sent a valid body, so no stock moved.

The impact is straightforward inventory sabotage by anyone: zeroing products out to stop sales, inflating them into overselling that cannot be fulfilled, or shuffling stock between locations, all of which mean real money and real operational disruption.

Remediation.

1. Require session and staff role on both routes via the central `/api/*` guard (same root fix as F-09 and F-17).
2. Validate input before touching the database, and wrap errors so the raw Postgres message does not leak (see F-10).
3. Log and audit every stock adjustment and transfer.

F-26: IDOR and missing authentication on the prospect-detail endpoint (full WhatsApp conversations)

- Critical. CVSS 3.1: 7.5 (`AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N`).
- Asset: panel, the prospect-detail endpoint (GET).
- Status: confirmed (one prospect read, anonymous). I did not enumerate the rest, per the stop rule.

If F-07 leaks the phone book, this one leaks the conversations. An anonymous `GET` to the prospect-detail endpoint (no cookie, no authentication header) returned a roughly 22 KB document that, on top of the metadata F-07 already exposed, included a `history` field: the entire exchange between the prospect and the WhatsApp bot, every message with its role, text, and timestamp.

```
{
  "id": "<id>",
  "name": "<redacted>",
  "topic": "<topic>",
  "status": "<status>",
  "history": [
    { "role": "user", "content": "<message>" },
    { "role": "assistant", "content": "<bot reply>" }
  ]
}
```

That is private communication (purchase intent, prices discussed, delivery details, and whatever the customer typed into a chat they believed was private). The chain with F-07 is trivial: F-07 already returns all roughly 300 prospect identifiers unauthenticated, so an attacker loops them through this endpoint and dumps every conversation the business has had. I did not run that loop (bulk PII, stop rule), but there is nothing clever left to work out; the iteration is trivial to script.

One thing that was done correctly, and worth noting: the reply endpoint (sending a message as the business) is properly locked (`401` with no credentials, `403` as a customer). Reading was open while writing was guarded, an odd split, but the write side deserves credit.

Remediation.

1. Require session and an authorized role on prospect-detail (same root fix as F-07 and F-09).
2. Treat `history` as private communication and restrict it to roles with a genuine need to see it.
3. Review the access logs for this endpoint to see whether anyone enumerated it before this assessment.

F-32: Missing authentication and confused deputy on the storefront payment-verification endpoint

- Critical. CVSS 3.1: 9.1 (`AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N`).
- Asset: storefront, the payment-verification endpoint (GET).
- Status: confirmed reachable with no authentication, querying MercadoPago with the identifier I supplied. Tested with a fake identifier only.

The storefront's payment-verification endpoint needs no login and runs its logic anyway: it takes a payment identifier you provide, queries the MercadoPago API, and returns a verdict. Same confused-deputy shape as F-19 (the app trusts a caller-supplied payment identifier, here an anonymous one, without checking that the payment belongs to one of your own orders).

I sent a nonexistent payment identifier with no authentication and received `404` ("payment not found in MP"), not `401`. That confirms it queried MercadoPago about the identifier I chose. I only ever used a fake one.

If this endpoint marks the matching order paid when MercadoPago confirms (the usual post-checkout wiring), then someone who knows or guesses an approved payment identifier could trigger confirmation on an order that is not theirs. It chains with F-11, F-19, and F-33 into several anonymous routes to the same destination: order payment status.

Remediation.

1. Require a session and check that the payment or order belongs to the logged-in user before verifying.
2. Take payment status only from MercadoPago's signed webhook (which the storefront does validate) or a server query bound to the user's own order.

F-33: Missing authentication on the order payment-status endpoint

- Critical. CVSS 3.1: 8.2 (`AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:H/A:N`).
- Asset: panel, the order payment-status endpoint (POST).
- Status: confirmed reachable with no authentication, down to business validation. Tested with a fake identifier and empty body, so no real order was touched.

No session check, no role check. An anonymous POST passed the gate and reached the body validation, which asked for a boolean `paid` field. So the order of operations is backwards: it validates the shape of the input before it ever validates who is asking. With a real order identifier and `{"paid": true}`, an anonymous attacker marks an order paid, or unpays a paid one.

I received a `400` asking for the `paid` field from a completely unauthenticated request; a `401` would have stopped me earlier, so the `400` proves I reached the logic. I only used a nonexistent order identifier and an empty body, so nothing real changed. Once again the

siblings give it away: restock and send-receipt on the same order both return `401`. Payment-status was the one left open.

An anonymous attacker flipping orders to paid means goods ship without payment; flipping them to unpaid means legitimate orders get denied. Either way it hits money and the integrity of the sales record, and if order identifiers are sequential it is trivially enumerable.

Remediation.

1. Verify session and role at the top of the handler, before validating the body.
2. Derive payment status only from a verified MercadoPago confirmation, never a client-sent boolean.

F-34: IDOR on the shipping-label endpoints

- Critical. CVSS 3.1: 7.5 (`AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N`).
- Asset: panel, two shipping-label endpoints (GET).
- Status: confirmed reachable with no authentication, running the lookup and proxy. Tested with nonexistent identifiers only.

Two label-generation endpoints, neither checking session or role, both reachable with an anonymous `GET`:

- The carrier-A label endpoint looks up an order by identifier and, with a valid one, builds the MercadoLibre label. Unauthenticated, a fake identifier returned `404` ("order not found"), so it reached the lookup rather than a `401`.
- The carrier-B label endpoint is a direct proxy to the carrier's API with the code you pass. Unauthenticated, a fake code returned `502` with the carrier's own "shipment not found" error passing through, so my anonymous request reached the carrier.

A shipping label carries the recipient's name, address, and phone. With a valid identifier or code, an anonymous caller retrieves the label and the PII on it for other people's shipments. I only ever used nonexistent identifiers, so I never retrieved a real label.

If the identifiers or codes are sequential or short, this becomes mass extraction by enumeration (a base for shipping fraud, social engineering, and competitor snooping).

Remediation.

1. Check session and staff role at the top of both handlers.
2. For a customer viewing their own label, validate ownership (the order or shipment has to belong to them).
3. Wrap the third-party carrier errors before returning them (see F-10).

F-36: Price tampering on the order-creation endpoint (arbitrary discount yields a negative total)

- Critical. CVSS 3.1: 7.7 (`AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:H/A:N`).
- Asset: storefront, the order-creation endpoint (and the coupon-validation endpoint).
- Status: confirmed live (two real orders created, both cash, no charge fired).

This is root cause number three confirmed in the clearest possible way. The order-creation endpoint takes a `discount_amount` in the body along with a coupon reference. The server does not recompute that discount from the actual coupon and the cart subtotal; it simply subtracts the number you sent. So with any valid coupon in hand, even a 5% one, I could set the discount to any value, including more than the cart was worth.

I took one real item and one legitimate 5% coupon and changed nothing but the discount field. A small discount gave the correct total. Then I set the discount higher than the subtotal, and the server created the order with a negative total:

Discount sent	Total returned	Result
A small amount	Correct discounted total	Order created
Larger than the subtotal	Negative total	Order created

Both were real orders, placed as cash so no charge went out, and I flagged both for cancellation.

While I was in there I noticed the coupon-validation endpoint also takes a client-supplied `subtotal` and computes the discount from that (another place the browser gets to do the math). But the one that actually bites is order-creation, which accepts `discount_amount` directly and skips validation entirely.

A negative total, depending on how billing and reconciliation treat it downstream, can turn into money the store owes the customer (a credit or a refund). Replace the cash method with one that fires MercadoPago and the same manipulation sets the amount that is charged. This is the live confirmation of the pattern that F-11 and F-19 only describe.

Remediation.

1. Recompute `discount_amount` entirely server-side, from the real coupon (looked up by identifier) and the real subtotal (each variant's real price times quantity). Ignore any `discount_amount` in the body, and preferably do not accept it as a field at all.
2. Clamp it: the discount cannot exceed the subtotal, and the total cannot go negative. Compute `total = max(0, subtotal − discount)`.
3. Apply the same at the coupon-validation endpoint (recompute over the real server-side cart, not a client-sent subtotal).
4. Cancel the two test orders.

F-38: Missing authentication on the WhatsApp bot-action endpoints

- Critical. CVSS 3.1: 7.7 (`AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:H/A:N`).
- Asset: panel, three WhatsApp bot-action endpoints (POST).
- Status: confirmed live. Tested with empty bodies, so no real combo identifier was sent.

Three panel routes that drive the business's WhatsApp bot, none of them checking authentication. Every one, on an anonymous POST, reached real execution instead of returning `401` / `403` :

- The **bulk-reply** action fires the bot's mass reply to unopened chats. An empty body returned a `502` carrying a real WhatsApp-service error ("WhatsApp not connected yet"), which means the proxy actually tried to run it against the live bot, and it failed only because the bot happened to be unlinked at the time (see F-28).
- The **combo/promotion** action sends a promotion to the business's WhatsApp group. An empty body returned `400` ("missing combo identifier"), so the business validation ran before any authentication check.
- The **label-sync** action behaved like bulk-reply (`502` , proxying to the real service with no authentication).

None returned `401` or `403` , so all three prove the anonymous request cleared access control. I never sent a real combo identifier, because that would have targeted the actual WhatsApp group.

With the bot linked and running, an outsider could trigger mass replies to real prospects (consuming AI credits and damaging the number's reputation), send unauthorized promotions to the group, or force runaway syncs.

Remediation.

1. Require session and staff role at the top of all three handlers, before any body validation or call to the WhatsApp service.
2. Rate-limit bulk-reply and combo on top of that, since they are expensive and reputation-sensitive even once authentication is fixed.

5.2. High

F-11: Mass assignment on the panel order-creation endpoint

- High. CVSS 3.1 (approximate): 6.5 (`AV:N/AC:L/PR:L/UI:N/S:U/C:L/I:H/A:N`). Critical if it reaches payment status.
- Asset: panel, the panel order-creation endpoint (POST).
- Status: confirmed at the API-contract level (from the bundle), and supported live by F-36 on the storefront's twin route.

Reading the panel bundle, I found the "New sale" form building the entire order body client-side, including fields that should be the server's decision and no one else's: each line's unit price, the discount, a manual discount, shipping cost, order status, and, most seriously, a

`payment_collected` boolean that declares whether the money was taken. If the server trusts any of that, whoever can reach this route can set arbitrary prices (any per-line price, 100% off, free shipping) and fake payment (`payment_collected: true` , status "delivered", with nothing actually collected).

I stopped short of firing a live panel order, because that is the stop rule (it moves money). But I did not have to guess whether the backend revalidates, because F-36 answered it on the storefront's twin route: same backend, same class of flaw, and there it accepted a client discount without recomputing and produced a negative total. That is strong evidence this route trusts the client too.

This is the kind of bug that outlives the access-control fix. Even after you lock down who can call the route, the question of what the server accepts is still open, and arbitrary prices, unlimited discounts, and orders booked as paid and delivered with no payment are outright fraud and a broken ledger.

Remediation.

1. Recompute the total server-side from the database by variant identifier, and ignore any client price, discount, or shipping.
2. `payment_collected` and payment status do not come from the client. Derive them from the processor's confirmation or a cash action with its own role check, and validate the state transitions server-side.
3. Enforce the role check in the handler.

F-12: Stored prompt injection via the bot knowledge-base endpoint

- High. CVSS 3.1 (approximate): 6.5 (`AV:N/AC:L/PR:L/UI:N/S:U/C:L/I:H/A:N`).
- Asset: panel, the bot knowledge-base endpoint (the `bot_knowledge` table), read by the WhatsApp bot and the AI-assistant endpoint.
- Status: confirmed write path (reachable as a `customer` via F-09). I did not trigger the effect on the bot's output live.

The "Bot knowledge base" section saves entries through the knowledge-base endpoint, with a free-text `content` field and no sanitization. That table feeds the AI assistant's context (both the WhatsApp bot and the storefront AI endpoint build answers from it). Since F-09 already proved a `customer` account can create and delete these entries, a low-privilege user can plant whatever "knowledge" they like, and the assistant will repeat it to other customers as fact, with no human in the loop.

I did not actually poison it, but the abuse is easy to describe: an invented coupon ("use code X for 90% off"), a fake price or product rule, or a redirect to a payment channel the attacker controls. Because the model treats this content as a trusted source, it is a persistent prompt injection (it stays in the knowledge base and colors every future conversation, not just one chat).

Remediation.

1. Restrict writes to the knowledge-base endpoint (`POST` / `PATCH` / `DELETE`) to admin or operator, server-side.
2. Treat `bot_knowledge` as untrusted in the prompt: separate it from the system prompt, give it no authority to rewrite the bot's rules, and validate or moderate `content` on save (length, injection patterns, human approval for new entries).
3. Record author and timestamp per entry for audit.

F-13: Mass assignment on the product-management endpoint

- High. CVSS 3.1 (approximate): 6.5 (`AV:N/AC:L/PR:L/UI:N/S:U/C:L/I:H/A:N`).
- Asset: panel, the product-management endpoint (`POST` / `DELETE`).
- Status: confirmed live (a persisted write with a `customer`-role cookie).

The product form builds the whole object client-side and sends it to the product-management endpoint: base price, and per variant the sale price, cost price (an internal field), and compare-at price, plus the MercadoLibre markup and a supplier reference. The server does not look the product up to derive its price; it takes whatever is sent.

I confirmed it. Through Caido, authenticated with nothing but a `customer`-role cookie, I sent a product write and the server persisted the fields I built on the client with no recompute and no role check (I raised the base price by a single unit and it held in the panel). I kept the change symbolic (one peso, deliberately) and restored it immediately after the screenshot. I did not touch the variant sale price on a real product, because that is the one the public storefront actually charges against, and moving it would move money (stop rule).

Beyond catalog price manipulation, this distorts internal costs and margins and moves the markup that drives automatic repricing to MercadoLibre. Chain it with F-11 and F-36 and the attacker controls the price in two places at once: the catalog and the checkout.

Remediation.

1. Treat cost price, supplier, markup, and price as fields controlled by the server and restricted by role, with range validation.
2. Enforce a real role check in the product-management handler (`POST` / `DELETE`).
3. Log price and cost changes to an audit trail for manipulation detection.

F-14 and F-35: Path traversal and unauthenticated arbitrary file write on the image-upload endpoint

- High. CVSS 3.1: 7.3 (`AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:H/A:L`), rising toward 8.6 with the write into other buckets.
- Asset: panel, the image-upload endpoint (POST), and Supabase Storage.
- Status: confirmed live (traversal, overwrite, and anonymous access). No real object was

overwritten.

I am folding F-14 (the traversal) and F-35 (that it works with no authentication at all) together, because they are the same bug seen twice.

The panel's image-upload helper sends the file plus a client-chosen `path` for where it should land. The server joins that `path` onto the Supabase Storage path with no sanitization, so `../` in the `path` climbs out of the intended bucket and into any other bucket in the project. And because the upload has `upsert` enabled, it overwrites what is already there.

I worked it in steps, dropping a harmless canary each time:

1. **Baseline.** A benign filename returned `200 OK` and the object appeared in the expected bucket, confirming that `path` is appended directly.
2. **Traversal across buckets.** A `path` with `../` aimed at a nonexistent bucket normalized out of the intended bucket and returned "bucket not found." The error changing from "path not found" to "bucket not found" is what told me I had escaped the bucket with an attacker-chosen destination, without writing anywhere real.
3. **Overwrite.** Re-uploading to the same path returned `200`, not `409`. `upsert` is on.
4. **No authentication (F-35).** The same multipart upload with no cookie, token, or key returned `200` and created the object. Fully anonymous.

A brief detour worth mentioning: I tried to turn this into stored XSS by uploading a `.svg` and an `.html` with script in them. It went nowhere, and it is Supabase's doing rather than the app's. Storage serves files with `Content-Disposition: attachment`, `Content-Security-Policy: sandbox`, and `X-Content-Type-Options: nosniff`, rewrites `text/html` to `text/plain`, and serves from a different origin than the app. That set of defaults closed the vector cleanly, which is a control working as intended.

Even without XSS, this is serious: an anonymous attacker can write and overwrite arbitrary objects in any bucket in the project, with the server key's authority (likely `service_role`). Realistic damage is catalog defacement (overwriting real product images) and overwriting files in other buckets.

Remediation.

1. The server generates the object name and path (`crypto.randomUUID()` plus a validated extension, or derive it from a validated identifier) instead of taking it from the client. Normalize the path and reject `..`, absolute paths, and separators.
2. Validate type, extension, and size server-side, by magic bytes rather than the declared extension or `Content-Type`, and set `upsert: false` unless it is truly needed.
3. Check session and role in the handler.

F-16: Outdated Next.js (13 known CVEs)

- High. CVSS 3.1: 7.5 (`AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N`).
- Asset: storefront and panel (two different Next.js builds, both in the affected range).
- Status: confirmed by version. I did not exploit the individual CVEs.

Both apps run Next.js builds from before the coordinated security release, and both sit inside the affected range. That release rolls up 13 advisories, two of which are App Router authorization-bypass issues via segment-prefetch (directly relevant, since the panel guards its pages with exactly the mechanism those CVEs undermine). The set also includes SSRF, several DoS bugs, cache poisoning, and XSS. The exact version is in the public bundle, so an attacker can map the CVE list without effort; I pulled it straight out of the JavaScript on both hosts.

The honest part: I spent a fair amount of time trying to actually land the segment-prefetch authorization-bypass against the panel, and it never worked. Without a session cookie the server simply redirected me to `/login` without leaking anything. So that specific CVE did not pan out here, but poking at the same RSC prefetch machinery is what turned up F-30, a real availability bug. And the rest of the advisories still apply by version regardless.

Remediation.

1. Update Next.js to the patched release (or the latest patched minor) in both projects, panel first, and confirm the reported version meets or exceeds it on both hosts.
2. Pin React to a stable production version instead of the canary; pinning the patched stable Next.js handles this.
3. Add `npm audit` (or equivalent) to CI so you do not fall behind the next batch.

F-18: Missing authentication on multiple panel routes (business finances exposed)

- High. CVSS 3.1: 7.5 (`AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N`).
- Asset: panel, several `/api/*` routes (see matrix).
- Status: confirmed (unauthenticated reads across several routes). Read-only.

I ran an access-control sweep (anonymous `GET` s across the panel routes) and a whole cluster returned internal business and finance data with no credential at all. Same root cause as F-07, F-09, and F-17, but the standout here is that the business's finances and ROI were simply readable.

Route category	HTTP	Gated	What leaked
WhatsApp business metrics	200	No	Revenue, profit, cost, ROI multiple
Advertising spend (MeLi ads)	200	No	Campaigns: name, status, spend, overhead, period

Route category	HTTP	Gated	What leaked
Billing (MeLi billing)	200	No	Monthly billing: commission, shipping, ads, installments, total
Integration debug	200	No	Integration state: user id, nickname, token expiry
Suppliers listing	200	No	Supplier list
Bot knowledge base	200	No	Full bot knowledge base (internal discount and operating rules)
Bot AI mode/model	200	No	Bot AI state and model
Customer listing	401	Yes	Correctly gated
Marketing ads report	401	Yes	Correctly gated

A competitor, or anyone, obtains the cost structure, margins, billing, and ROI without authenticating. The integration-debug route adds the integration state (user identifier, token expiry) as reconnaissance on top. And the two `401`s in that table are the point: the gating exists, it was just applied inconsistently.

Remediation.

1. Apply the central session and role guard to all of the panel's `/api/*`.
2. Treat finance, metrics, and integration-debug as privileged. The debug route should not exist in production, or it belongs behind admin authentication.

F-30: Pre-auth denial of service via a self-referential `next-url`

- High. CVSS 3.1: 7.5 (`AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H`).
- Asset: storefront, account pages (confirmed on the addresses page).
- Status: confirmed (reproduced twice). I did not escalate it, per the rules of engagement.

This one came out of the F-16 detour. The account pages are guarded by the middleware, which redirects to `/login` when there is no session. But the router has a second way in (the request Next.js makes to fetch a client-side transition's payload, using the `rsc`, `next-router-state-tree`, and `next-url` headers), and on that path the check hangs instead of redirecting, when two things line up: no valid session cookie, and a `next-url` header identical to the path being requested (self-reference).

The differential is what confirmed it: send that request with a valid cookie and it answers `200` normally; send it without one and the server never responds at all (I had to cancel the connection manually). So it is specifically the "redirect-to-login for a data refetch, but there is no session" path where a self-referential `next-url` spins into a loop with no exit.

Because it is pre-authentication, anyone without an account triggers it with a single inexpensive request, and each one pins a serverless function until the platform timeout (a

classic concurrency-exhaustion pattern). I did not push it toward actually taking the site down (DoS at scale is off-limits here), so I document it as reasoned but unmeasured. I reproduced the hang exactly twice, in isolation, and left it there.

Remediation.

1. Fix the loop with an explicit base case for "target path equals `next-url`."
2. Exit early on the unauthenticated path: no session, redirect to `/login` before processing the route tree.
3. Add an explicit internal timeout in the handler so a hung path fails fast instead of consuming the full function timeout.
4. Update Next.js (F-16); the same patch set fixes DoS bugs in this family.

5.3. Medium

F-08: Open RLS (business data readable by `anon`)

- Medium. CVSS 3.1: 5.3 (`AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:N`).
- Asset: database, the app-settings, stock, and products tables.
- Status: confirmed (read with the public `anon` key).

Profiles (F-06) was not the only table left open. A few more answered the bare `anon` key with internal business intelligence. App-settings returned the MercadoLibre repricing configuration (minimum margin, max price-change percent, wholesale markup, the bot model, and more), so a competitor learns your price floor and your change ceiling. The stock table exposed real inventory levels. The products table leaked its internal columns alongside the public ones: base price, markup, supplier reference, and MercadoLibre item mapping.

There is no PII and no money here, but it is a clean read of margins, automatic-repricing rules, per-supplier costs, and stock, which is real competitive intelligence and a lever to nudge the pricing autopilot indirectly.

Remediation.

1. Apply restrictive RLS on app-settings and stock, and on the internal columns of products.
2. For the storefront, expose only what it needs (name, sale price, image) via a view or column-level policy, and keep base price, supplier, and markup out of `anon`'s reach. Operational config should be `service_role`-only, server-side.

F-15: No real lockout on OTP verification (CWE-307)

- Medium. CVSS 3.1 (approximate): 5.3 (`AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N`).
- Asset: authentication, Supabase Auth OTP verification (email OTP login).
- Status: confirmed (two real tests). I did not brute-force anyone else's code.

Login is by email OTP, and the verify step has no failed-attempt counter tied to the account or the code. There is a generic infrastructure burst throttle (it triggers around the 33rd attempt at five requests per second with a `429`), but it resets in six seconds and neither locks the account nor invalidates the OTP. I confirmed that: right after the `429`, the real code was accepted as if nothing had happened. (The endpoint that sends codes is properly limited; it is the one that checks them that is not.)

Two things soften it: the token is 8 digits (10^8, not the usual 10^6) and it expires in minutes, which makes single-origin brute force impractical. But the underlying issue (nothing actually stops attempts against a specific OTP) is still present, and it scales poorly the moment you parallelize across IP addresses.

Remediation.

1. Add rate-limiting on verify, tied to the account and the OTP; block that OTP after 5 to 10 failures, independent of the infrastructure throttle. A persistent counter (Redis or a table) keyed by account and OTP does it.
2. Invalidate the OTP after N failures, not just on use or time expiry.
3. Consider progressive backoff or a short account lockout after several failed OTPs in a row.

F-20: No origin validation on the MercadoLibre webhook endpoint

- Medium. CVSS 3.1: 5.3 (`AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:L/A:N`).
- Asset: panel, the MercadoLibre webhook endpoint (POST).
- Status: confirmed (tested against a nonexistent resource, no effect on real data).

The MercadoLibre webhook takes notifications with no authentication and no origin check, and answers `200`. So an anonymous attacker can forge notifications pointing at arbitrary resources. MercadoLibre does not sign its webhooks, which means the correct design is: on notification, re-fetch the resource from the MeLi API with your own token before acting on it. If the handler skips that re-fetch and trusts the body, the impact is worse, but I could not confirm that black-box without real data. (For contrast, the storefront's MercadoPago webhook does validate a signature and rejects with `401` when the header is missing. The team knows how to do this; it just was not done here.)

Even at minimum, outsiders can inject notifications and flood the endpoint, driving reprocessing and MeLi API calls that consume the account's quota (an economic denial of service). If the handler trusts the body, it becomes outright event forgery.

Remediation.

1. Validate that the payload's user and application identifiers are yours, and, above all, always re-fetch the resource from the MeLi API before acting; do not trust the body's fields.
2. Restrict by MeLi's source IP addresses if they publish them, and rate-limit.

F-22: Missing authentication and denial of wallet on the AI-assistant endpoint

- Medium. CVSS 3.1: 5.3 (`AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:L`). High if credit exhaustion takes down the WhatsApp bot.
- Asset: storefront, the AI-assistant endpoint (POST).
- Status: confirmed (one minimal completion; I did not repeat it, because repeating is the attack).

The storefront AI endpoint calls the model (Claude) and answers with no credentials required, and every call spends the business's Anthropic credits. I sent one minimal message, received a real completion with no authentication, and stopped there, because repeating the call would be the denial of wallet itself. An outsider scripting this drains the credits (direct cost), and if the web assistant and the WhatsApp bot share an account or quota, it can take the bot down with it (there is a documented past incident of the bot failing on exhausted credits). It is also an unauthenticated prompt-injection surface. (A malformed body, incidentally, returned a `500` with the runtime's raw error, which is more material for F-10.)

Remediation.

1. Require a session, or at least a storefront token, for the AI endpoint.
2. Rate-limit and quota per IP address or session, with a maximum token budget per window.
3. Separate the web assistant's Anthropic quota from the WhatsApp bot's so abusing the web endpoint cannot starve the bot.
4. Wrap handler errors and bound the input (message length and count).

F-25: Missing role check on the MercadoLibre item-inspect endpoint

- Medium. CVSS 3.1: 4.3 (`AV:N/AC:L/PR:L/UI:N/S:U/C:L/I:N/A:N`).
- Asset: panel, the MercadoLibre item-inspect endpoint (GET).
- Status: confirmed (with a `customer`-role test token).

This endpoint checks that you are logged in but not that you are staff. So an ordinary `customer` pulls internal MercadoLibre integration data for any variant (item mapping, listing fields). It is an inconsistency more than a crisis: its sibling routes all return `403` to a customer, and this one alone omitted the role check. With a plain customer session cookie it returned `200` and the internal MeLi mapping.

The direct impact is low (no PII, no writes), but it is one more data point showing that role enforcement here is decided route by route instead of centrally.

Remediation.

1. Add the staff-role check its MeLi sibling routes already have.

2. Apply the central role guard across all of the panel's `/api/*`.

F-27: Client-side field lock in the profile (CWE-602), a stored XSS candidate

- Medium. CVSS 3.1: 4.3 (`AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:L/A:N`). Rises if a stored XSS is confirmed.
- Asset: storefront, the profile page, persisting to the Supabase user-profiles table.
- Status: confirmed live. The test write was reverted.

The "My profile" form disables the first-name, last-name, and Instagram fields when the account is linked to Google. That lock is purely cosmetic (client-side only). I removed the `disabled` attribute from the inputs in developer tools, typed a test payload, and saved, and the Server Action accepted and persisted it without re-checking server-side that those fields were supposed to be locked. I confirmed it landed by reading the profile back through the `anon` key (F-06):

```
{
  "first_name": "<injected test payload>",
  "last_name": "<injected test payload>",
  "instagram": "<injected test payload>"
}
```

Worth noting the email field is not exposed to this. It is not part of what this Server Action serializes, and changing it goes through a separate confirmation flow. That part is designed correctly.

So integrity is broken: any customer writes arbitrary values into fields the interface presents as locked, and the payload sits in a table `anon` can read. I am calling it a stored XSS candidate rather than a confirmed XSS. If these fields get rendered unescaped somewhere in the panel (a customer view, an order detail) or placed into a string-built email, it becomes XSS that could fire in a real admin's session. Finding that unsafe render point is the piece I did not reach.

Remediation.

1. Enforce the same rule server-side that the interface pretends to: if the account has a linked OAuth identity, reject changes to those fields or always take them from the provider.
2. Sanitize and validate Instagram (a handle, not HTML).
3. Confirm that every render of these fields escapes HTML by default, and specifically audit any email or PDF that builds them by string concatenation.

---

6. Hygiene and best-practice findings (low and informational)

None of these is a direct threat on its own, but they widen the blast radius of the real bugs, ease an attacker's reconnaissance, or reflect careless configuration. Condensed.

F-31: Session cookie without `HttpOnly` or `Secure` (CWE-1004, CWE-614)

Low (CVSS 5.3). Confirmed on the storefront. The Supabase Auth session cookies ship with `Path=/; SameSite=lax` but no `HttpOnly` and no `Secure`. Without `HttpOnly`, any JavaScript on the origin can read the session token via `document.cookie`, which is exactly the payoff an XSS is after. That directly amplifies F-27: confirm that unsafe render point and the missing `HttpOnly` turns it into full session theft. Confirmed on the storefront; likely on the panel too by shared code, though I did not confirm it independently. Fix: set `httpOnly: true` and `secure: true` in the `@supabase/ssr` cookie options in both projects.

F-02: Missing security headers

Low (CVSS 4.3). Confirmed. Neither app returns the standard set: no `Content-Security-Policy`, no `X-Frame-Options` (the panel login is framable, a clickjacking risk), no `X-Content-Type-Options: nosniff`, no `Referrer-Policy`, no `Permissions-Policy`. `Strict-Transport-Security` is present but without `includeSubDomains` or `preload`. Low on its own, but the missing CSP raises the ceiling on any XSS, and the missing frame protection makes login clickjackable. Fix: add the set in the framework config or middleware (`X-Frame-Options: DENY`, `X-Content-Type-Options: nosniff`, `Referrer-Policy: strict-origin-when-cross-origin`, a tight `Permissions-Policy`, and a CSP in `report-only` to start), and finish HSTS with `includeSubDomains; preload`.

F-10: Verbose error messages (PostgREST and Supabase)

Low or Informational. Confirmed. Several panel and storage errors return the raw PostgreSQL, PostgREST, or Fastify message ("violates row-level security policy for table ...", or the stock-table constraint error from F-24). Not exploitable alone, but it confirms the stack and hands over table and column names (free reconnaissance). Fix: catch backend errors per route, return your own generic message, and log the real detail server-side only.

F-37: Schema enumeration via the PostgREST `PGRST205` hint

Informational or Low. Confirmed. Ask for a table that does not exist and the Supabase API suggests a real one by similarity ("Perhaps you meant the table ..."). That turns the API into a schema-enumeration oracle, and it is how I turned up several real table names during this test, including discount-codes (F-23). Fix: normalize PostgREST 404s before returning them, and trim the exposed schema to what the frontend actually needs.

F-29: Secret in the URL (`?key=`, CWE-598) on the WhatsApp bot

Low (CVSS 3.7). Confirmed. The WhatsApp service guards its routes with a secret passed as a query-string parameter (`?key=`), which an error message spelled out. A secret in the URL ends up in CDN logs, proxy logs, server logs, and browser history (the classic secret-in-URL anti-pattern). The guard itself is solid (it does not distinguish an empty key from a wrong one, so it is no oracle, and an exhaustive search for the secret across everything I had collected came up empty). It rises to Critical if that secret ever leaks, because the QR-pairing endpoint would then hand over the pairing QR for the business's WhatsApp (full channel takeover). Fix: move authentication to a header (`Authorization` or `X-API-Key`), fix the error message so it stops advertising the mechanism, and rotate the current secret.

F-28: Bot health endpoint leaks session state

Low (CVSS 5.3). Confirmed. The WhatsApp service's health endpoint answers with no authentication and states the bot's internal session state (waiting-for-QR, unpaired, ready/connected flags). That lets anyone watch, in real time, whether the bot is linked or down (a timing signal for scheduling actions, and it pairs with F-29). Fix: reduce the body to `{"ok":true}`, or restrict `/health` to a monitor's IP addresses.

F-21: Attack surface (the WhatsApp service is on the internet)

Low. Confirmed (surface mapped via an active sweep). The WhatsApp service sits on its own subdomain (Express and Node behind Cloudflare). I swept it actively, and it held up well: uniform, correct gating (a generic `401` for everything except F-28 and F-29), no unauthenticated functional endpoints, no CORS, no reachable stack traces, and no recoverable origin IP (likely a Cloudflare Tunnel). Documented as mapped surface; the actual issues are F-28 and F-29. Fix: see those two, and make sure the origin only accepts Cloudflare traffic.

F-05: Supabase `anon` key in the bundle (by design)

Informational. Confirmed, and expected. The `anon` key and project URL are in the frontend bundle, which is how Supabase is meant to work, with real security resting on RLS (F-06, F-08, F-23). It is not a vulnerability but a starting point. On the positive side, I searched for privileged keys (`service_role`, MercadoPago, Anthropic, Resend, Google tokens) across the bundles and traffic and found none. Fix: nothing for `anon` itself; keep RLS strict and make sure `service_role` never reaches the client (verified today).

F-04: Stack disclosure via response headers

Informational. Confirmed. The hosts advertise their stack: `x-powered-by: Next.js`, `server: Vercel`, and (the useful one) `X-Matched-Path`, which returns the literal internal Next.js route pattern, brackets and all, even in error responses to anonymous calls. That hands an attacker the project's route-naming convention, which is exactly how you guess sibling routes. Fix: drop `x-powered-by` (`poweredByHeader: false`) and check whether the platform allows suppressing `X-Matched-Path` in production.

F-03: CORS (`Access-Control-Allow-Origin: *` on the panel's static responses)

Informational. Confirmed, not currently exploitable. The panel login page and static assets return `Access-Control-Allow-Origin: *`. I checked the dangerous combination (it does not reflect the attacker's `Origin` and does not pair with `Allow-Credentials: true`), so it is harmless here, just Vercel's default for static content. Flagging it as a reminder: if any authenticated `/api/*` endpoint ever copied this pattern with credentials, it would be a High. Fix: for sensitive endpoints, use an explicit origin allowlist and never pair `Allow-Credentials` with a wildcard.

F-39: Missing `security.txt` and an indexable panel

Informational. Confirmed. Three small items: there is no `/.well-known/security.txt` on any host (no formal way to report a bug responsibly, RFC 9116); the storefront `robots.txt` has a `Disallow: /admin` pointing at a route that does not exist on that host (configuration noise); and the panel login has no `noindex`, so the admin panel could appear in search results. Fix: publish `security.txt`, clean up the `robots.txt` line, and add `noindex` to the panel login.

F-01: Panel scope correction

Informational. Confirmed (a scope note, not a vulnerability). The rules of engagement assumed the panel might live on one subdomain, a hostname that did not resolve (NXDOMAIN). The real panel is on a different one. Documented and the scope amended at the time.

---

7. Coverage: tests that passed

A black-box test is never a straight line, and the things that did not break are part of the value, because they mark what is already solid. For the record:

- **Stock race condition and overselling: solid.** I fired two purchase requests at once (quantity 6 each) against a product with 2 real units. One was rejected ("insufficient stock: available 2, requested 6") and the other went through, with no oversell. The stock check is atomic and holds under concurrency. (This created one test order, flagged for cancellation.)
- **SQL injection in the store search: ruled out.** Boolean-, error-, and time-based tests on the product search produced no differential and no syntax error; the input arrives parameterized. It does behave oddly around punctuation, but that is the full-text-search parser, not attacker-controllable, so it is hygiene rather than injection.
- **Stored XSS via file upload: ruled out.** Supabase Storage's hardening (attachment disposition, sandbox CSP, nosniff, `text/html` to `text/plain` rewriting, separate origin)

closes the vector (see F-14 and F-35).

- **Open redirect in Supabase Auth verification: ruled out.** `redirect_to` respects a server allowlist, and the flow uses PKCE (an authorization code, not tokens), which bounds the impact even in a hypothetical bypass.
- **Segment-prefetch authorization bypass (the Next.js CVE): did not work here.** Tested head-on against an account page with no session cookie; the server simply redirected to `/login` without leaking. The same machinery did produce the availability bug F-30.
- **Cron endpoints: solid.** All five (broadcast, stock reconcile, catalog pricing, abandoned cart email, pending-order cancel) require a cron secret and return `401` without it.
- **Middleware matcher bypass: ruled out.** I tried the usual tricks (`.png` , `.svg` , double slash, `%2e%2e` , `.json`), and it held every time. (The irony is that the middleware was never the problem; the ungated `/api/*` behind it was.)
- **Coupon enumeration at checkout: ruled out.** The validation endpoint returns the same "invalid or expired" for a code that does not exist, one that belongs to someone else, and one that is expired, with no differential to enumerate valid codes.
- **Password login and recovery: not applicable by design.** No password login, no reset flow; everything is OTP (F-15).
- **MercadoPago webhook (storefront): solid.** Validates the signature and rejects with `401` when the header is missing (the exact thing the MeLi webhook, F-20, does not do).
- **WhatsApp message send (the reply endpoint): solid.** `401` with no credentials, `403` as a customer (see F-26).
- **Privileged secret leaks: none.** Searched bundles and traffic for `service_role` , MercadoPago, Anthropic, Resend, and Google, and found nothing (F-05).

Residual items I did not reach, in the interest of honesty: the abandoned cart voucher flow (claim and review), whether the session token (JWT) is actually invalidated on logout, and independent confirmation of the cookie flags (F-31) on the panel (I confirmed them on the storefront).

---

8. Prioritized recommendations

Now.

1. One central session and role guard on all of the panel's `/api/*` and on panel page loads (F-07, F-09, F-17, F-18, F-24, F-25, F-26, F-32, F-33, F-34, F-38). Highest return by a wide margin, since it removes the largest share of the Critical findings at once.
2. Close the RLS on user-profiles, discount-codes, app-settings, and stock, and on the internal columns of products; block all writes from `anon` and `authenticated` (F-06, F-08, F-23).

3. Recompute prices, discounts, and payment status server-side and ignore the client (F-11, F-13, F-36); take payment status only from MercadoPago's signed webhook (F-19, F-32, F-33).
4. Cancel the test orders and delete the test objects (per the private cleanup notes).

Next 2 to 4 weeks.

5. Update Next.js to the patched release in both projects (F-16).
6. Move the bot's authentication to a header and rotate the secret (F-29); trim the health endpoint (F-28).
7. Set `HttpOnly` and `Secure` on the session cookies (F-31).
8. Add rate-limiting tied to the account on OTP verify (F-15) and on the AI endpoint (F-22); validate the MeLi webhook's origin (F-20).
9. Have the server name uploaded files and validate type and role (F-14 and F-35).

Then (hardening).

10. Add the security headers and a CSP (F-02); normalize error messages (F-10, F-37); drop `x-powered-by` and `X-Matched-Path` where possible (F-04).
11. Treat the bot knowledge base as untrusted content (F-12); sanitize profile fields server-side (F-27).
12. Publish `security.txt` and add `noindex` to the panel (F-39).
13. Put dependency scanning in CI, and review RLS every time you add a table.

Anonymized case study by Tomás Gorrini Rech. Client identity, domains, credentials, and reproduction detail removed for public release. Published with the client's written consent, after the reported issues were fixed.