Security Checklist

Shipping an app has never been easier. You can go from idea to deployed product in a weekend with an agent doing 99% of the typing. The problem is that AI writes code that *works* long before it writes code that is *safe*, and the difference usually stays invisible until a stranger on the internet finds it for you.

The good news: vibe-coded apps tend to get burned by the same handful of mistakes, and all of them are checkable in an afternoon. This guide covers the seven I see most, in the order I would check them. For each one you get three things:

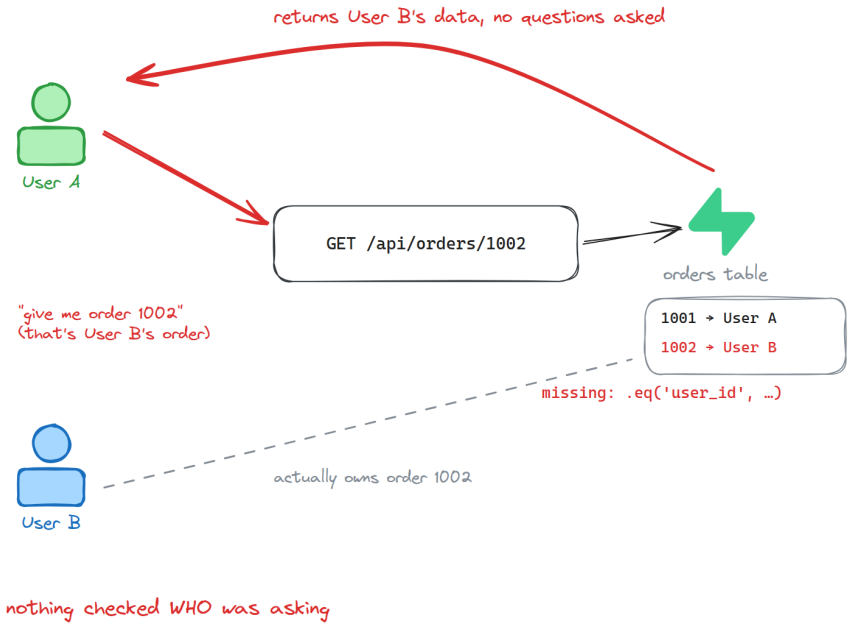
- A diagram of what the attack actually looks like
- The fix
- A prompt you can paste into your agent to fix it for you

Coding models are pretty good now. You do not need to become a security engineer, you just need to know what to ask for. That is what the prompts are for.

This is the companion resource to my video, so if you want to see these attacks demoed live, start there. Otherwise, open your codebase and work down the list.

1. Authorization

This is the first thing I check because it's quick to fix and failure is catastrophic. How to test for it: log in as User A, take any request with an ID in it (an order, a project, a document), and swap in someone else's ID. If the API returns data, congratulations, you have found the single most common vulnerability in vibe-coded apps. The fancy name is **IDOR** (insecure direct object reference). The root cause is almost always the same: your app checked that *someone* was logged in, but never checked *who* was asking for *what*.



User A asks for User B's order. The backend hands it over.

The fix

Defend in two layers. Turn on Row Level Security so the database itself refuses to hand out rows the user does not own, and scope every query in your API routes anyway.

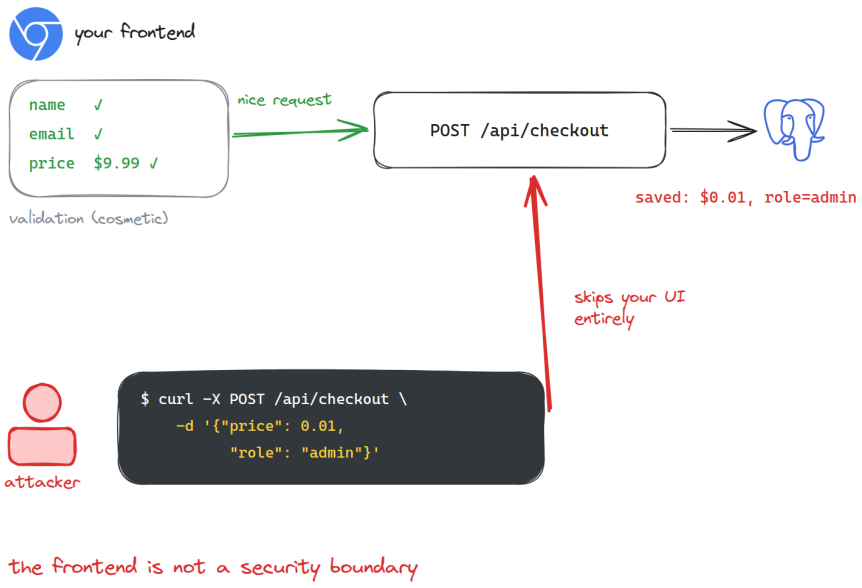
```
alter table orders enable row level security;

create policy "users read only their own orders"
  on orders for select
  using (auth.uid() = user_id);

-- and in your API routes
const { data } = await supabase
  .from("orders")
  .select("*")
  .eq("id", orderId)
  .eq("user_id", user.id);  // the missing check
```

Prompt your agent:

> *"Audit every API route for broken authorization. Verify the logged-in user owns every row they read or write, and enable RLS on all tables with policies scoped to auth.uid()."*

2. Never Trust the Client

Every check in your frontend is a SUGGESTION. Users can open devtools, edit requests, or skip your UI entirely and hit the API with curl. If your server trusts the request body, an attacker can set their own price at checkout or quietly hand themselves an admin role. Your beautiful form validation never even gets a say.



The attacker does not use your form.

The fix

Validate everything again on the server. Define a schema for each route, whitelist the fields you accept, and never let the client decide things like prices, roles, or permissions.

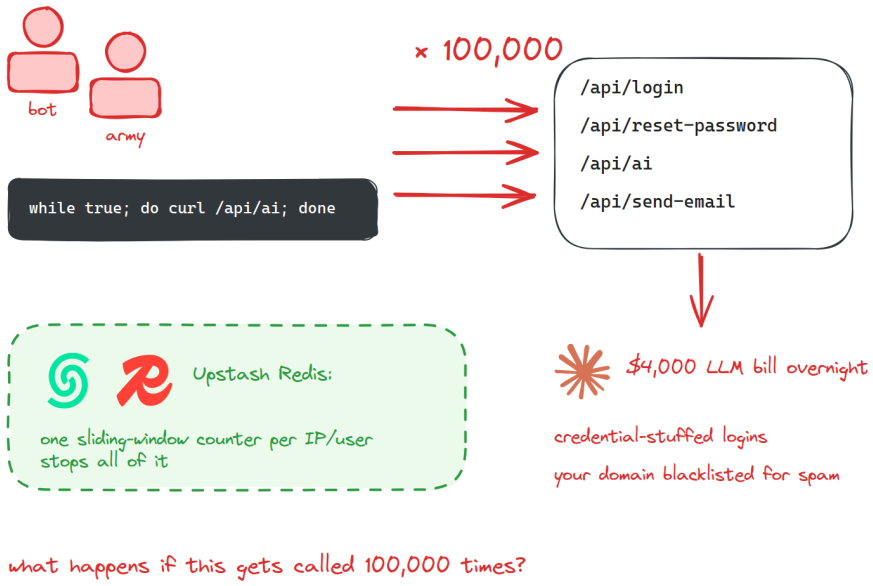
```
const Body = z.object({
  productId: z.string().uuid(),
  quantity:  z.number().int().min(1).max(100),
});  // no "price" field: the server decides prices

const body = Body.safeParse(await req.json());
if (!body.success)
  return Response.json({ error: "bad input" }, { status: 400 });
```

Prompt your agent:

> *"Add server-side Zod validation to every API route. Whitelist fields, never spread request bodies into DB writes, and never take prices, roles or permissions from the client."*

3. Rate Limiting

Ask this about every endpoint: what happens if someone calls it 100,000 times tonight? For most vibe-coded apps the honest answer is a four-figure AI bill, a spammed user base, or a database on fire. The usual victims are login, password reset, anything that calls an LLM, and anything that sends email or SMS.



One while-loop away from a very bad morning.

The fix

One sliding-window counter per IP and user, checked at the top of each sensitive route. Upstash Redis makes this about ten lines.

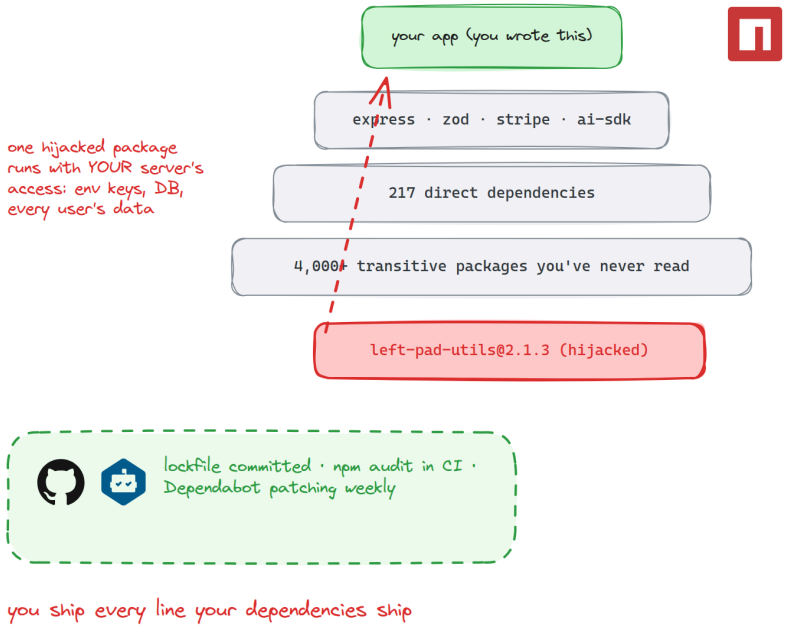
```
const limiter = new Ratelimit({
  redis: Redis.fromEnv(),
  limiter: Ratelimit.slidingWindow(10, "10 s"),
});

const { success } = await limiter.limit(`login:${ip}`);
if (!success)
  return new Response("slow down", { status: 429 });
```

Prompt your agent:

> *"Add rate limiting with @upstash/ratelimit, keyed by IP and user ID, to every endpoint that costs money or can be abused: login, signup, password reset, AI calls, emails."*

4. Supply-Chain Security

You wrote maybe 2% of the code you are shipping. The rest came from npm, and every line of it runs with full access to your server, your environment variables, and your database. One hijacked package deep in the dependency tree becomes your problem the moment you install it.



Your app is the small box on top.

The fix

Know what you install, pin what you know, and let robots watch it for you. Commit the lockfile, install with npm ci so builds are reproducible, fail CI on known vulnerabilities, and turn on Dependabot.

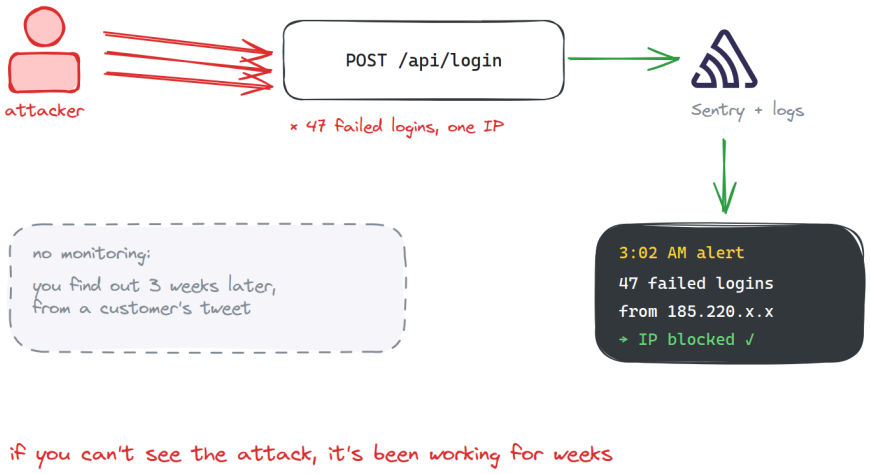
```
npm ci                          # installs exactly the lockfile
npm audit --audit-level=high    # fail CI on known vulns

# .github/dependabot.yml
version: 2
updates:
  - package-ecosystem: "npm"
    directory: "/"
    schedule: { interval: "weekly" }
```

Prompt your agent:

> *"Audit my dependencies and remove anything unneeded or sketchy. Commit the lockfile, use npm ci in CI, fix npm audit highs, and add weekly Dependabot updates."*

5. Logging & Monitoring

If someone was hammering your login endpoint right now, would you know? Most apps find out about attacks weeks later, usually from a customer. With a bit of logging and one alert rule, that becomes a 3 AM notification and a blocked IP instead.



Same attack, two very different mornings.

The fix

Add Sentry for errors, then log security *events* with enough context to act on: failed logins, rate-limit hits, permission denials. Top it off with one alert rule that pages you when something spikes.

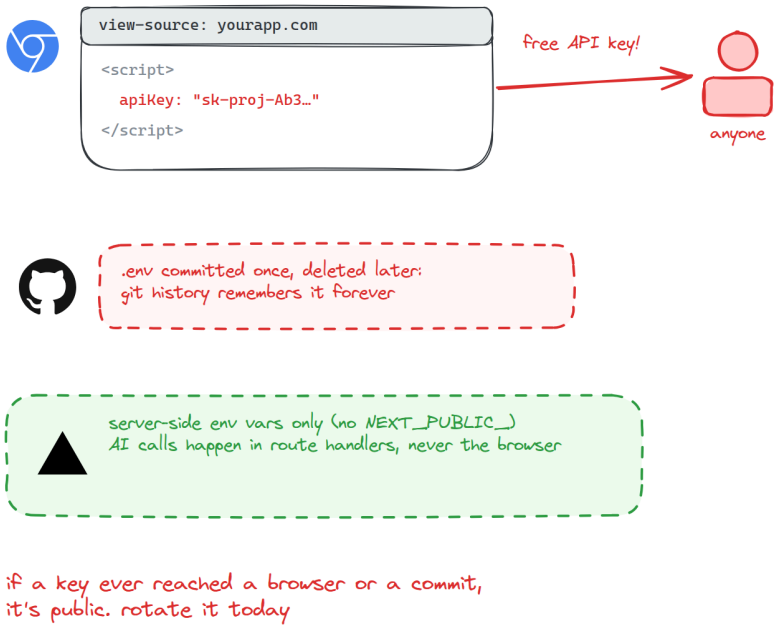
```
Sentry.init({ dsn: process.env.SENTRY_DSN });

logger.warn("auth.failed_login", { ip, email });
logger.warn("auth.rate_limited", { ip, route });
logger.warn("authz.denied",      { ip, userId, resource });

// alert rule: > 20 failed logins in 5 min from one IP
```

Prompt your agent:

> *"Add Sentry and structured logs for security events (failed logins, 403s, rate-limit hits), plus an alert for spikes like 20+ failed logins from one IP in 5 minutes."*

6. Secrets

This one is quick because you already know it. Two rules: secrets never reach the browser, and secrets never reach a commit. In Next.js, anything prefixed **NEXT_PUBLIC_** ships to every visitor. Anything committed to git stays in history even after you delete the file. If a key breaks either rule, assume it is stolen. Rotating it is the only real fix.



View-source is a free API key vending machine.

The fix

```
# .gitignore, BEFORE your first commit
.env*

# bad: shipped to every visitor's browser
NEXT_PUBLIC_OPENAI_API_KEY=sk-...

# good: server-only, set in your host's env vars
ANTHROPIC_API_KEY=sk-ant-...

// only read it in route handlers / server code
const key = process.env.ANTHROPIC_API_KEY;
```

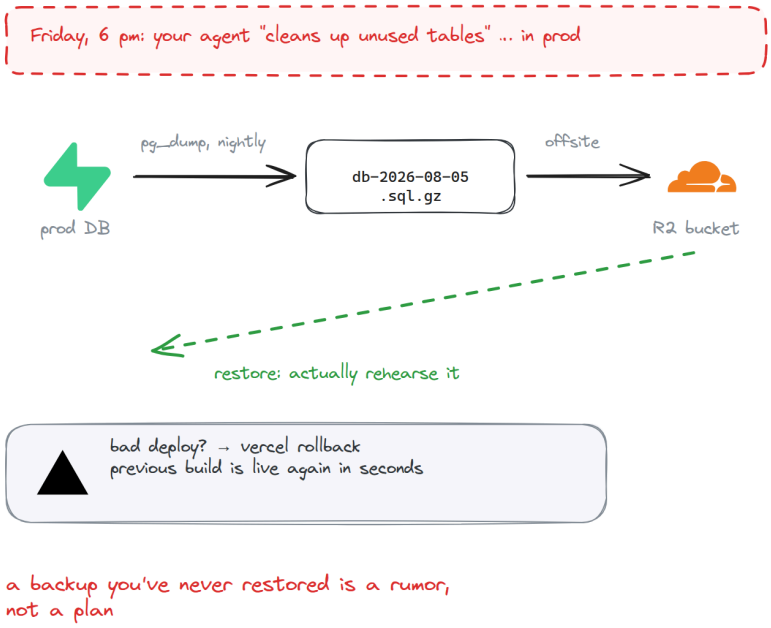
Prompt your agent:

> *"Scan this repo and its full git history for leaked secrets, and flag any key that reaches the browser. Move everything to server-only env vars and list what I need to rotate."*

7. Backups & Recovery

Everything above tries to prevent bad days. This check assumes the bad day happens anyway. Databases get dropped, deploys go sideways, and agents occasionally "clean up" a production table on a Friday evening. The only question that matters is whether you can come back.

One warning from experience: a backup you have never restored is a rumor. Rehearse the restore at least once.



Nightly dumps going offsite, and a restore path you have actually tested.

The fix

```
# nightly dump, stored OFF-platform
pg_dump "$DATABASE_URL" | gzip \
  | aws s3 cp - "s3://backups/db-$(date +%F).sql.gz" \
      --endpoint-url "$R2_ENDPOINT"

# rehearse the restore (THIS is the real backup)
createdb rehearsal
gunzip -c db-2026-08-05.sql.gz | psql rehearsal

# bad deploy? previous build back in seconds
vercel rollback
```

Prompt your agent:

> *"Set up nightly pg_dump backups to offsite storage, a restore script, and a short recovery runbook. Then run one restore rehearsal to prove it works."*

TLDR:

Run this before every launch. If you can honestly tick all seven, your app is in better shape than most of the internet.

☐ Log in as one user and request another user's data by ID. It fails.

☐ Hit your API with curl and a hostile payload. The server rejects it.

☐ Every endpoint that costs money or can be abused has a rate limit.

☐ Lockfile committed, npm audit clean, Dependabot on.

☐ Security events are logged, and at least one alert can wake you up.

☐ No secrets in the browser bundle or in git history. Anything that leaked is rotated.

☐ You restored a backup recently, and you know how to roll back a deploy.

P.S. If you want to actually learn software engineering beyond just coding (like this), you'll love my app Devmaxx. You get daily questions on everything beyond coding (system design, cybersecurity, scaling, infra, etc) and a private discord community where you can ask me questions directly, get referrals at top companies, and chat with your peers learning the same stuff as you. If you ask me, it's the highest leverage decision you can make right now if you want to level up as a developer.