VibeScan Security Audit

Repo: [SAMPLE — fictional repo, real finding patterns] · Order #0 · 2026-04-20 09:34 UTC

Summary

Critical	2	must-fix before launch
High	3	fix in next sprint
Medium	3	address when convenient
Files scanned	38	highest-risk files prioritised (API routes, auth, config, SQL)

Findings

[CRITICAL] Supabase service_role key committed to the repo
→ `src/lib/supabase.ts:3`

The SUPABASE_SERVICE_ROLE_KEY is hardcoded in the client bundle and ships with every page load. This key bypasses all Row-Level Security policies. Anyone who loads your deployed site can extract it from the bundled JS in DevTools and gain full read/write access to every table in the database.

> ***Fix:*** *Rotate the key immediately in the Supabase dashboard (Settings → API → Reset service role key), move it to a server-side-only environment variable (never prefixed with NEXT_PUBLIC_ / VITE_), and scrub it from git history with git filter-repo before pushing the rotated-key commit.*

[CRITICAL] Edge function writes orders without verifying the caller
→ `supabase/functions/process-payment/index.ts:12`

The process-payment function reads user_id from the request body with no JWT verification. Combined with the service-role client below, an attacker can POST to the public function URL with any user_id and any amount and insert fraudulent orders on any user's behalf, bypassing your React app entirely.

> ***Fix:*** *Verify the Authorization header via supabase.auth.getUser() using the anon key with the user's JWT attached, reject if the call returns an error, and use the returned user.id as the authoritative identity — never trust anything in the request body.*

[HIGH] No rate limit on /api/contact
→ `src/app/api/contact/route.ts:15`

The contact-form endpoint accepts unlimited POSTs. A single script can submit 10,000 forms per minute, exhausting your transactional-email credits (Resend / Postmark / SendGrid) and flooding your support inbox. No CAPTCHA, no IP-based rate limit, no request-signature check.

> ***Fix:*** *Add Upstash Ratelimit (Ratelimit.slidingWindow(5, "60 s") per IP) plus a hidden honeypot field. Reject requests where the honeypot is non-empty — bots fill every visible and hidden field, humans don't.*

[HIGH] CORS wide open on an endpoint that accepts credentials
→ `src/middleware.ts:8`

You return Access-Control-Allow-Origin: * AND accept credentials. Any malicious site can read authenticated responses from your API via a logged-in user's browser. The

* wildcard is silently ignored by browsers when credentials are involved — but the presence tells attackers the intent was to allow everything.

> ***Fix:*** *Replace the wildcard with an allowlist of origins you actually trust (your own production domain + localhost for dev). If you truly need open CORS, don't send credentials with the request.*

[HIGH] RLS policy `USING (true)` on cases table
→ `supabase/migrations/20240315_rls.sql:42`

The SELECT policy on public.cases uses USING (true), which means any authenticated user can read every row, not just their own. On an open-signup app, a throwaway Gmail account can read all user data — including records the UI never surfaces to the current user.

> ***Fix:*** *Replace with USING (user_id = auth.uid()). For UPDATE policies, also add WITH CHECK (user_id = auth.uid()) so users can't change the user_id column during an update and steal other people's rows.*

[MEDIUM] User-controlled redirect_uri on OAuth callback
→ `src/app/auth/callback/route.ts:31`

The next query parameter is used as the redirect target after successful authentication, with no allowlist check. An attacker can craft links like /auth/callback?next=https://phish.example.com that send users to a phishing page immediately after they complete login on your domain — a common account-takeover vector.

> ***Fix:*** *Either accept only relative paths (reject anything starting with http:// or https:// or //), or check the redirect URL's host against an allowlist of your own domains.*

[MEDIUM] Password reset token in URL exposes it to logs and referrers
→ `src/app/reset-password/page.tsx:24`

The reset token appears in the URL as a query string. Browser history, HTTP Referer headers, server access logs, and analytics scripts (Google Analytics, Plausible, anyone's tag manager) all capture it, extending the token's effective lifetime far beyond the reset flow. A stolen browser history file grants account takeover as long as the token hasn't been consumed.

> ***Fix:*** *Move the token to the request body via a POST, OR use a short-lived (5 min) one-time-use token that the server burns on first read. Set the Referrer-Policy: no-referrer header on the reset route either way.*

[MEDIUM] API keys stored unencrypted in localStorage
→ `src/context/ApiKeyContext.tsx:18`

User-provided API keys are persisted to localStorage in plain text. Any XSS vector — including a compromised third-party script (analytics, customer-support widget, font loader) — can read them via localStorage.getItem() and exfiltrate them via a single fetch call. XSS is common in AI-scaffolded apps because escaping is often missed.

> ***Fix:*** *Move the key to an httpOnly cookie (server-side session) so JS can't read it. If you must keep it client-side, encrypt with a key derived from the user's session and stored only in memory — never on disk.*
