Vibe-Coded App Security: Field Guide for Indie Founders

Ship AI-generated apps without handing attackers the keys to your SaaS.

NEURAGROWTH.CO

2026-07-20

CONTENTS

01

Why Vibe-Coded Apps Break Differently Than Hand-Written Code

The New Attack Surface Nobody Warned You About

When you describe a feature in plain English and watch Cursor, Lovable, Bolt, or Replit generate it in seconds, you are not just accelerating development. You are inheriting a specific, repeatable set of security mistakes that AI code generators make systematically - the same mistakes, across every codebase, for every founder who skipped the security chapter.

This is not about AI being bad at code. It is about AI being trained to produce working demonstrations, not hardened production systems. There is a difference.

How Traditional Security Advice Breaks Down Here

Most security guides assume: - You read every line of code before it ships - You have a staging environment and a CI pipeline - You know what a JWT is and why it matters - You have at least one engineer on the team who took a security course

If you are a solo founder vibe-coding a micro-SaaS between your day job and your kids' bedtime, none of those assumptions hold. That is fine. This guide does not assume them either.

What AI Generators Optimize For

LLMs generating code are rewarded (during training and in user feedback) for: - Code that runs on the first try - Code that demonstrates the requested feature clearly - Code that is readable and well-structured

They are not rewarded for: - Rejecting hardcoded credentials in favor of environment variables - Adding Row Level Security policies to every Supabase table - Verifying that every npm package they recommend actually exists - Treating every user input as potentially hostile

The result is a predictable vulnerability fingerprint. Every vibe-coded app tends to have the same holes.

The 5 Vulnerability Classes This Guide Covers

Through pattern analysis from the CSA AI Safety report and GitGuardian's 2026 State of Secrets Sprawl data, five classes account for the overwhelming majority of breaches in AI-generated micro-SaaS products:

#	Vulnerability Class	Typical Origin	Severity
1	Hardcoded API secrets	AI prefers concrete examples	Critical
2	Missing Supabase RLS	AI skips policy boilerplate	Critical
3	Hallucinated packages (slopsquatting)	AI invents plausible names	High
4	Unauthenticated admin endpoints	AI creates admin routes without guards	High
5	Prompt injection via user input	AI passes user text directly to LLM calls	High

Each of these gets its own chapter with a breach pattern, a root cause explanation, and a remediation checklist you can execute using the same tool that introduced the problem.

How to Read This Guide

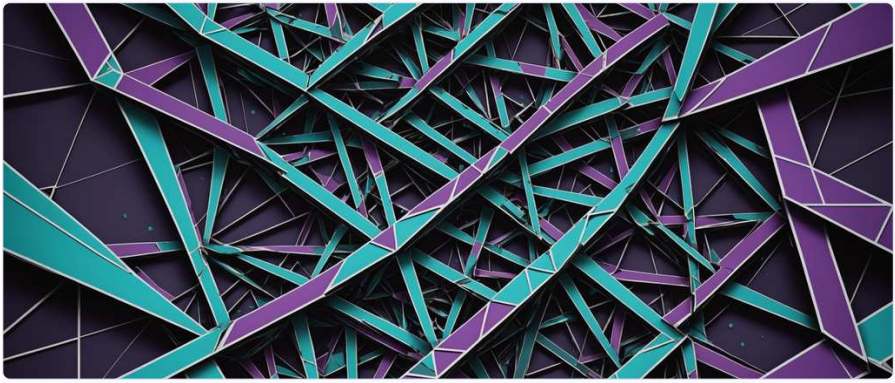
You do not need to read it front to back. If you are two days from launch and panicking, jump to the Pre-Launch Audit Checklist. If you just found a hardcoded key in your repo, jump to Chapter 2. If you are setting up a new project, start at Chapter 7 and add the CLAUDE.md rules template before you write your first prompt.

Every remediation in this guide is framed as a prompt you send to your AI coding tool or a config change in a dashboard. No terminal wizardry required unless you want it.

One Uncomfortable Truth Before We Start

The founders who get breached are not careless people. They are busy people who trusted that the AI knew what it was doing. The AI does not know what it is doing with security. It is pattern-matching on training data that includes thousands of tutorials where someone hardcoded a Stripe key to keep the example simple.

You are now the security layer. This guide is your toolkit.



WHY VIBE-CODED APPS BREAK DIFFERENTLY THAN HAND-WRITTEN CODE

02

Hardcoded Secrets: The Most Common Critical Mistake

The Breach Pattern

In 2025 and 2026, GitGuardian's scanning of public GitHub repositories found that over 12 million secrets were exposed in public commits. A significant and growing fraction came from repositories created with AI coding assistants. The pattern is consistent: a founder prompts an AI to build an integration with Stripe, OpenAI, Twilio, or SendGrid. The AI writes working code with a real-looking placeholder - or worse, the founder pastes their actual key into the chat for context, and the AI incorporates it literally.

The key ends up in: - A `.js` or `.py` file committed to a public GitHub repo - A `config.json` file that gets pushed because `.gitignore` was never set up - A frontend bundle where it is visible to any browser user who opens DevTools

Within minutes of a public push, automated scanners run by both security researchers and malicious actors detect the key. Average time to first abuse of a leaked key: under 4 minutes, per GitGuardian's data.

Why AI Generators Do This

Prompt: "Add Stripe payment processing to my checkout page."

The AI has seen thousands of Stripe tutorial examples. Most of them look like this:

```
// Example from AI output
const stripe = require('stripe')('sk_live_YOURKEYHERE');
```

But sometimes the founder's key is in the chat context, or the AI fills in a realistic-looking fake, and the founder does not notice the difference. The AI is not checking whether that string should be in an environment variable. It is just completing the pattern.

Finding Secrets in Your Existing Codebase

Before you fix anything, audit what is already there.

Option A: Prompt your AI tool directly

Open Cursor, Bolt, or whatever tool you use and send this prompt:

```
Search every file in this project for strings that look like API keys,
secret tokens, passwords, or credentials. Look for patterns like:
- Strings starting with sk_, pk_, rk_, AKIA, ghp_, xoxb-
- Variables named secret, api_key, password, token, private_key
- Any string longer than 20 characters assigned to those variable names
List every file and line number where you find one.
```

Option B: Use Gitleaks (free, open source)

If you are comfortable with a terminal:

```
brew install gitleaks
gitleaks detect --source . --report-format json --report-path leaks.json
```

This scans your entire git history, not just current files - important because deleting a file does not remove it from git history.

Option C: GitHub's built-in secret scanning

If your repo is on GitHub, go to Settings > Security > Secret scanning. GitHub scans for over 200 secret patterns automatically on both public and private repos (private requires GitHub Advanced Security on paid plans).

Remediating Hardcoded Secrets

Step 1: Rotate the key immediately

Before you fix the code, assume the key is compromised. Log in to Stripe/OpenAI/Twilio/whoever and generate a new key. Revoke the old one.

Step 2: Move secrets to environment variables

Send this prompt to your AI tool:

```
Refactor this project so that every API key, secret,
password, and token
is read from environment variables instead of being
hardcoded. Use
process.env.VARIABLE_NAME in Node.js /
os.environ.get('VARIABLE_NAME')
in Python. Create a .env.example file listing every required
variable
with placeholder values. Make sure .env is in .gitignore.
```

Step 3: Scrub git history

Deleting the file is not enough. If the secret was ever committed, it lives in git history. Use the BFG Repo-Cleaner or GitHub's built-in secret removal tool:

```
# Install BFG
brew install bfg

# Remove the secret from history
bfg --replace-text secrets.txt my-repo.git
git reflog expire --expire=now --all
git gc --prune=now --aggressive
git push --force
```

Step 4: Set up environment variables in your hosting platform

On Vercel: Project Settings > Environment Variables On Render: Service > Environment On Fly.io: `fly secrets set KEY=value` On Replit: Secrets tab in the left sidebar

Step 5: Prevent recurrence with a pre-commit hook

Send this prompt:

```
Add a pre-commit git hook using the 'pre-commit' framework
that runs
gitleaks before every commit and blocks the commit if any
secrets are
detected. Show me the .pre-commit-config.yaml file and the
installation
steps.
```

The One-Minute Rule

If a secret ever touches a public channel - a commit, a Slack message, a screenshot - treat it as burned. Rotate first, fix second, investigate third. Never convince yourself it was up for "only a second" and nobody saw it.

03

Supabase Row Level Security: The Silent Data Exposure

The Breach Pattern

The scenario plays out constantly in the indie hacker community. A founder builds a multi-tenant SaaS on Supabase. The AI generates clean CRUD operations. The app works perfectly in testing - because in testing, there is usually only one user. Launch day arrives. Users sign up. User A, curious or malicious, modifies a fetch request in the browser's network tab, changing their user ID to a different number. They get back User B's data. Every row in the database is readable and writable by any authenticated user.