A PRACTICAL CHECKLIST · GEORGE M. J. ZAK

Before You Deploy Your Vibe-Coded App

AI has made it possible for almost anyone to ship applications, websites, automations, and agents. That is genuinely amazing. But "I could build it" does not mean "it's safe to deploy." Most vibe-coded apps fail the same security checks. This is a practical walkthrough — not theory, not compliance talk — of what to verify before your AI-built project goes live, and the questions to put in front of your AI assistant to do the heavy lifting for you.

Cybersecurity | AI Builder Era | ~12 min read | v1 · 2026-06-06

George M. J. Zak · AI strategy, cybersecurity, and the human side of technology

WHAT'S INSIDE

SECTION · 01 · THE PREMISE

01. Why this matters now

I've spent years working in cybersecurity, risk, governance, and technology before any of this AI tooling existed. What I'm seeing in 2026 is unprecedented and slightly terrifying in equal measure: people who have never written a line of code in their life are now shipping applications. Real ones. Talking to real databases. Holding real users' data. Charging real money.

This is, on balance, wonderful. Software is finally returning to the hands of the people who want to solve their own problems with it. The pre-AI software industry treated coding as a priesthood; AI has democratized it. Good.

But the security part of software was never the easy part. It was hidden behind the priesthood. And now thousands of vibe-coded apps are going live every week, made by builders who never had to learn the security layer because someone else always did it for them.

> **THE SINGLE UNCOMFORTABLE TRUTH**
>
> The fact that you *could* build something does not mean it's safe to deploy. The fact that AI helped you build it does not mean AI checked whether it's safe. The two skills are different, and AI is currently better at the first one than the second.

This checklist is what I run with clients and what I run on my own projects. Go through it before you deploy. Then put it in front of your favorite AI assistant and ask it to actually do the work. The combination — your judgment, the AI's execution — is how you ship something that won't embarrass you in three months.

SECTION · 02 · IDENTITY

02. Authentication

Most security incidents start at the front door. If you got the front door wrong, the rest of the building doesn't matter.

VERIFY THESE:

- ☐ **Admin pages are protected end-to-end.**

  Not just the page that shows the admin UI — the API routes the admin UI calls. The UI gate is a convenience for honest users; the API gate is the real protection. If a URL exists that returns sensitive data without checking auth, that URL will be discovered.

- ☐ **User accounts are properly secured.**

  Password rules that don't fight password managers. Email verification before access to anything sensitive. Account lockout after repeated failures so brute-force costs the attacker more than it costs you.

- ☐ **Passwords are stored correctly.**

  Hashed with bcrypt, argon2, or scrypt — never plaintext, never reversible encryption. Salts are per-user. If your AI assistant produced plaintext password storage, fix it before deploy. This is a career-ending bug in production.

- ☐ **MFA is available for sensitive functions.**

  Even if you don't enforce it, offer it. TOTP via authenticator apps is the cheap default. SMS is better than nothing but vulnerable to SIM swap.

- ☐ **Session tokens are properly handled.**

  HttpOnly, Secure, SameSite=Lax (or Strict for high-security flows). Reasonable expiration. Revoked on logout. Bound to the user agent if you can.

THE PATTERN I SEE MOST OFTEN

AI-generated auth code looks correct because it follows established patterns. But the AI often skips the implementation details that actually make those patterns secure — token rotation, session invalidation on password change, replay protection. Ask explicitly: *"For my authentication implementation, walk me through what happens when (a) a user changes their password, (b) a session token leaks, (c) a user logs out, (d) someone tries the same password 50 times. Show me each code path."*

SECTION · 03 · DATA

03. Database security

If users can read each other's data, you don't have an app. You have a data leak with a UI on top.

VERIFY THESE:

- ☐ **Row Level Security is enabled on every table containing user data.**

  Especially if you're on Supabase, Firebase, or any service where the anon/client key is exposed to the browser. RLS is the wall between "what your code intends" and "what the browser can actually do." Without it, the wall isn't there.

- ☐ **Users cannot access data that belongs to other users.**

  Test this directly. Log in as User A. Try to read a record you know belongs to User B. If the API returns it, you have a horizontal privilege escalation bug — one of the most common AI-generated bugs because the AI added the read endpoint without scoping it to `auth.uid()`.

- ☐ **Permissions follow least-privilege.**

  The anon role should have the minimum grants possible. The service-role key should never touch the browser. Admin-only tables should have anon revoked entirely.

- ☐ **Sensitive fields are not over-exposed.**

  Email, phone, address, ID numbers — if the user doesn't need to see them, the API shouldn't return them. A list of users that returns every column is a list of users that exfiltrates everything when the wrong query runs.

- ☐ **Backups exist and have been tested.**

  "We have backups" is not the same as "we have tested that we can restore from backups." Restore drills are boring; they're also what separates the companies that recover from the ones that don't.

THE SUPABASE / FIREBASE / NOCODE TRAP

Hosted backends ship with their "anon" or "client" key visible in your frontend code. This is intentional and fine — *as long as RLS or equivalent rules are configured.* Many vibe-coded apps ship with RLS turned off on every table. That means the anon key, which any visitor can extract from your JavaScript, has full read/write access to your entire database. Check this before launch. Today. It's a free fix and a complete change in your security posture.

SECTION · 04 · SECRETS

04. Secrets & API keys

Every secret that ends up in your repository or in your frontend bundle is a secret on the open internet. Treat it accordingly.

VERIFY THESE:

- ☐ **No API keys are present in frontend code.**

  Anything in your `src/` or `public/` directory becomes part of the JavaScript bundle the browser downloads. Scan your repo for the prefixes of every service you use — Stripe (`sk_`), Resend (`re_`), OpenAI (`sk-`), etc. If any of them appear outside an environment-variable reference, rotate the key immediately.

- ☐ **Secrets are stored in environment variables.**

  Not in code. Not in config files committed to git. Always check `.gitignore` includes `.env*` and your secrets file is on the gitignored side.

- ☐ **Test credentials are not present in production.**

  If you used a test Stripe key during development, make sure the production deploy has the live key. If you used a test Resend audience, make sure the production deploy isn't still pointing at it. AI-generated boilerplate often leaves test credentials behind.

- ☐ **Service-account keys are scoped to least privilege.**

  A service-account key that can read everything in your Google Cloud project is a key you should not have. Scope to specific buckets, specific projects, specific roles.

- ☐ **If a key has ever been exposed, it has been rotated.**

  No exceptions. If you committed it to a public repo for ten seconds, the key is compromised. GitHub indexes commits within minutes; bots scrape new repos within hours.

SECTION · 05 · INPUTS

05. Input validation

Every form, every API endpoint, every URL parameter is an opportunity for someone to send something you didn't expect. Assume malicious input by default.

VERIFY THESE:

- ☐ **Server-side validation on every form.**

  Client-side validation is for UX. Server-side validation is for security. If your AI gave you only client-side validation, you have no validation at all. `curl` bypasses every JavaScript check.

- ☐ **SQL injection is impossible.**

  Use parameterized queries / prepared statements. Never concatenate user input into SQL. If you use an ORM or query builder (Supabase, Prisma, etc.), this is usually handled — but check any raw queries you wrote.

- ☐ **XSS (cross-site scripting) is blocked.**

  If your app renders user-supplied text, the text must be escaped or sanitized. React, Vue, and Svelte do this by default — until you use `dangerouslySetInnerHTML` or `v-html`. Audit every use of those.

- ☐ **File uploads are validated.**

  Check MIME type (server-side, not just the client-sent header). Limit file size. Strip metadata. If you allow images, consider re-encoding them through a library that drops anything that isn't pixel data. Never trust the filename.

- ☐ **Honeypots are in place on public forms.**

  A hidden input named `website` or `hp` that real users won't fill but bots will. If it's filled, silently drop the submission. Cheapest spam filter that exists.

SECTION · 06 · ENDPOINTS

06. Public APIs

Every API endpoint is a public surface unless you've explicitly gated it. Treat each one as a probable attack target.

VERIFY THESE:

- ☐ **Rate limiting is in place.**

  Per IP, per user, per endpoint. Most platforms (Cloudflare, Vercel, AWS API Gateway) offer this with a few clicks. Use it. An unlimited login endpoint is an unlimited brute force target.

- ☐ **Sensitive data is not exposed.**

  A "get user profile" endpoint shouldn't return password hashes, internal IDs, or admin flags. Define a public schema explicitly and stick to it.

- ☐ **Attackers cannot enumerate users or records.**

  A signup endpoint that returns "this email already exists" is enumerable — an attacker can iterate through emails and discover who is registered. A login endpoint that distinguishes "wrong password" from "user not found" is enumerable. Normalize your responses.

- ☐ **CORS is configured tightly.**

  If you don't need cross-origin access, don't allow it. If you do, allow the specific origins you trust — not ∗.

- ☐ **HTTP methods are scoped to what each endpoint needs.**

  A read-only endpoint shouldn't accept POST. A modify endpoint shouldn't accept GET. Tighten each route.

SECTION · 07 · VISIBILITY

07. Logging & monitoring

You cannot defend what you cannot see. The first thing an attacker takes from you is your ability to know they're there.

VERIFY THESE:

- ☐ **Security-relevant events are logged.**

  Login attempts (success and failure), password changes, permission elevations, admin actions, API errors, rate-limit triggers. Keep these on a separate logging surface from regular app logs if possible.

- ☐ **Logs are retained long enough to be useful.**

  90 days minimum for most applications. Longer for anything regulated. Logs you delete after 24 hours catch nothing meaningful.

- ☐ **You can detect abuse in progress.**

  Sudden spike in login failures? Unusual geography? New IP making admin API calls? Set up at least one or two alerts. Anomaly detection is hard; alerting on the obvious is easy.

- ☐ **You know when something fails silently.**

  An email-sending service that returns an error code your code ignores will fail invisibly until your customers complain. Wire up application error tracking (Sentry, Rollbar, or equivalent). Free tiers cover most small apps.

- ☐ **PII is not in your logs.**

  Passwords, full email addresses, payment data, identity documents — none of this should be written to logs. Logs leak too. If you have to log a user identifier, hash it.

SECTION · 08 · AI-SPECIFIC

08. AI-specific risks

If your app uses AI — any AI, LLMs, embeddings, agents, vector search — you have a new layer of risks that didn't exist five years ago. None of them are exotic. All of them are exploitable today.

RISK	SEVERITY	WHAT TO VERIFY
Prompt injection	CRITICAL	User-supplied text is being mixed into your system prompt. An attacker writes "ignore previous instructions and..." and your agent obeys. Treat all user input as untrusted; never concatenate it directly into prompts. Use clear delimiters; validate output before acting on it.
Tool / agent overpermission	CRITICAL	If your agent can execute code, send email, transfer funds, or modify data — what's the blast radius if a prompt injection succeeds? Restrict tool permissions to the minimum. Require human-in-the-loop confirmation for high-impact actions.
Data exfiltration via output	HIGH	An AI given access to a database can be tricked into revealing data the user shouldn't see. If the model can read more than the user is authorized to read, that gap will be exploited. Scope AI access to the data the requesting user is allowed to access.
Cost amplification	HIGH	A single unauthorized user with no rate limit can run up your token bill into the thousands of dollars in an afternoon. Rate-limit AI endpoints. Cap per-user monthly token spend. Monitor for unusual usage.
Insecure output handling	HIGH	If you render AI output as HTML, an attacker can prompt-inject HTML that runs in your users' browsers. If you execute AI-generated code, an attacker can prompt-inject malicious code. Treat AI output as untrusted input to the next system.
Training data leakage	MEDIUM	If you fine-tune on user data, that data may surface in responses to other users. If you must train on user data, sanitize and obtain explicit consent. Most apps don't need fine-tuning; consider retrieval-augmented generation instead.
Model swapping by provider	MEDIUM	Your provider quietly upgrades your model. The new version behaves differently. Behaviors you depended on stop working — or worse, start working in subtly wrong ways. Pin your model versions where the provider allows it; test before unpinning.
Conversation log exposure	MEDIUM	If you log AI conversations, those logs may contain PII, secrets, or proprietary content the user shared with the model. Apply the same protection to AI conversation logs as you do to user data — possibly stronger.

THE MENTAL MODEL

An LLM agent in your application is approximately as trustworthy as a contractor you met at a job site five minutes ago. Smart, capable, eager to help — and absolutely should not be given the master keys, the bank login, or unsupervised access to the parts of the building you wouldn't show a stranger. The "principle of least authority" matters more for AI components than for any other part of your stack, because AI components are easier to manipulate.

SECTION · 09 · DEPLOYMENT

09. Deployment hardening

The last mile. If you get the deployment wrong, you can do everything else right and still have a public security incident.

VERIFY THESE:

- ☐ **HTTPS is enforced everywhere.**
  No HTTP. No mixed content. HSTS header set with a reasonable max-age. Plain HTTP requests should redirect, not just be available.
- ☐ **Security headers are configured.**
  Content-Security-Policy (even a permissive starter is better than none). X-Content-Type-Options: nosniff. X-Frame-Options or frame-ancestors. Referrer-Policy. These are five lines of config that block half of common attacks.
- ☐ **Cloudflare / WAF / equivalent is active.**
  If you're not behind a CDN with WAF, you're directly exposed. Cloudflare's free tier handles most of what a small app needs. Use it.
- ☐ **DDoS mitigation is in place.**
  Cloudflare gives this for free at small scale. Vercel and similar platforms include rate limiting by default. Make sure you've turned the relevant features on.
- ☐ **Dependency vulnerabilities are checked.**
  Run `npm audit` / `pip-audit` / equivalent on each deploy. Fix critical and high before launch. Subscribe to Dependabot or Renovate so you find out about new CVEs without checking.
- ☐ **Domain hardening is done.**
  DNSSEC enabled. Registrar lock active. WHOIS privacy on. Strong password + MFA on your DNS provider account. Your domain is your identity online — losing control of it ruins your week.

10. The prompt to use on your AI

Here's the practical move. Copy-paste this into your AI assistant. It will do most of the work above for you. Your job is to read what it surfaces and decide what's worth fixing.

COPY THIS PROMPT:

```
Act as a senior cybersecurity architect performing a pre-deployment review of this application.

Your job is to identify security risks at four severity levels: CRITICAL (deploy-blocking), HIGH (fix this week), MEDIUM (fix this month), LOW (track and revisit).

Walk through these categories systematically:

1. Authentication — admin pages, user accounts, password handling, MFA, session security
2. Database security — RLS or equivalent, horizontal access controls, sensitive field exposure
3. Secrets — API keys in frontend, env var hygiene, test credentials in production
4. Input validation — server-side checks, SQL injection, XSS, file uploads, honeypots
5. Public APIs — rate limiting, sensitive data exposure, enumeration, CORS, method scoping
6. Logging — what's logged, retention, abuse detection, silent failure detection, PII in logs
7. AI-specific — prompt injection, tool permissions, output handling, cost amplification
8. Deployment — HTTPS, security headers, WAF, dependency vulnerabilities, domain hardening

For each finding, give me:
— The category and severity
— A one-sentence description of the risk
— The specific file or location in my code
— A concrete fix I can apply

End with a prioritized punch list of what to do before this ships, sorted by severity.

Be thorough. Be specific. Don't reassure me — surface real issues. Be honest about uncertainty when you can't see something from the code I shared.
```

HOW TO USE IT WELL

Give the AI access to your repository if you can — through a code-aware assistant, an IDE integration, or by pasting representative files. The prompt is most useful when the AI has actual code to look at, not just your description of the app. Iterate: after the first pass, ask follow-ups on specific findings ("show me the exact code change for finding #3"). Then verify each fix actually applied. AIs sometimes claim to fix things they didn't fix.

SECTION · 11 · COMPLIANCE

11. EU AI Act & governance touchpoints

If your app touches European users, the EU AI Act applies regardless of where your company is registered. The law is being enforced in waves through 2026; most general-purpose AI obligations land in August 2026. A few of the act's requirements map directly onto the security checklist above.

WHAT TO VERIFY IF YOU'RE SHIPPING IN OR TO THE EU:

- ☐ **Article 12 — Record-keeping.**
  High-risk AI systems must maintain logs. Your AI monitoring and audit logs aren't optional if you're in scope — they're statutory.
- ☐ **Article 14 — Human oversight.**
  High-risk systems require human oversight of model output. Tool / agent overpermission isn't just a security risk — it's a compliance issue if your agent is acting without supervision in a high-risk domain.
- ☐ **Article 10 — Data governance.**
  Training data quality, bias mitigation, and data lineage. If you fine-tune, you need a story for how the training data was collected, validated, and bias-checked.
- ☐ **Article 15 — Accuracy, robustness, cybersecurity.**
  This article maps directly to the entire checklist above. The act explicitly requires AI providers to address cybersecurity. Your security posture is now a regulatory artifact.
- ☐ **GDPR overlay.**
  Personal data passing through AI components is still personal data under GDPR. Right to erasure, data minimization, purpose limitation — all still apply. AI does not exempt anyone from data protection law.

IF YOU'RE NOT SURE WHETHER YOUR APP IS "HIGH-RISK" UNDER THE EU AI ACT

The Annex III list of high-risk use cases includes credit scoring, employment / HR screening, education access decisions, law enforcement applications, critical infrastructure, and several other domains. Most consumer-grade apps aren't high-risk. But the general-purpose AI model provider obligations apply to *everyone* shipping an AI product into the EU, and many of the security best practices map directly onto compliance expectations. Operating securely is now the floor, not the ceiling.

12. The honest closing

Building is easier than it has ever been. Securing what you build still takes attention. The combination — AI builds it, you and AI together verify it — is the workflow that produces software you can be proud of shipping.

None of the items on this checklist are difficult individually. The challenge is that they're easy to miss. Each one is one of those "I'll do it later" tasks that builders skip because shipping feels more urgent than securing. And then, three months after launch, something goes wrong. Usually quietly. Usually visible only in retrospect.

Run the checklist. Use the prompt. Read what your AI surfaces. Fix the criticals before deploy. Track the rest in a tasklist with deadlines. Ship.

And then, three months later, do it again. The security posture of a live application isn't a state — it's a practice.

WHY I'M PUBLISHING THIS FOR FREE

If you're shipping AI-built software into the world, the entire ecosystem benefits when you do it securely. Bad incidents — leaked databases, prompt-injection scandals, AI agents that took actions they shouldn't have — slow down adoption for everyone. The way to keep the AI builder era going is for the people building in it to ship reasonably-secure software. This checklist is one small contribution toward that.

If this helped, you can pay it forward.

Share it with one other person who's shipping AI-built software. Send it to the engineering team at the company you work for. Run it on your own app and tell me what was surprising.

If your team needs more direct help — architectural review, EU AI Act readiness, AI governance design, or hands-on security work — that's the field work I do. Get in touch.

— George M. J. Zak · jorgemjzak.com · 2026-06-06