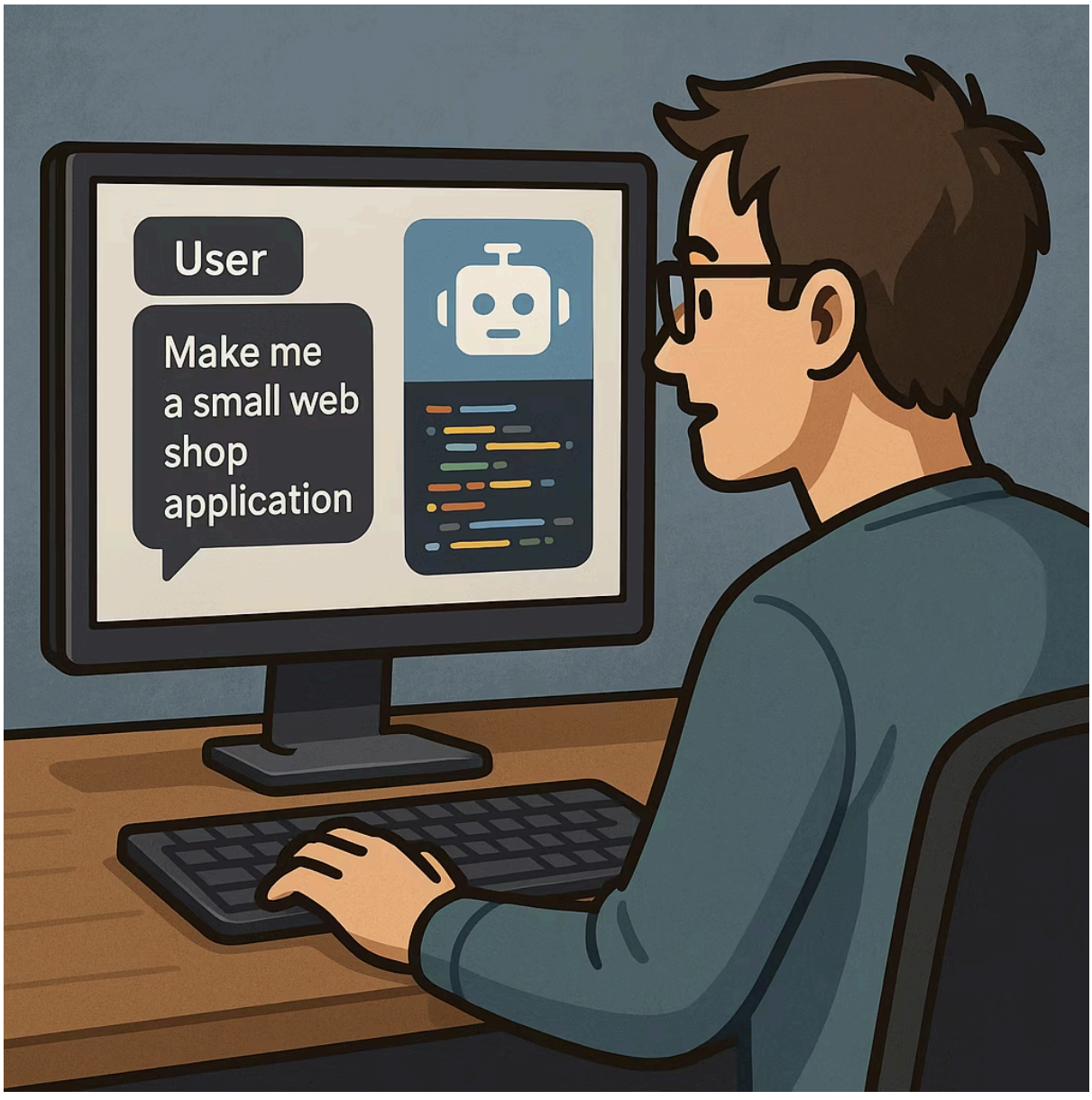
Finding more vulnerabilities in vibe coded apps

BY INTIGRITI · APRIL 16, 2025

Vibe coding is the latest trend sweeping through developer communities. It's the art of describing a concept, feeding it to an AI, and letting the LLM (Large Language Model) manifest the code based purely on vibes. The quote states, "You fully give in to the vibes, embrace exponentials, and forget that the code even exists."

And as more developers rely on AI to "vibe" their way through coding, we're entering a new golden age of bug bounty hunting. Why? Because AI-generated code often looks functional, it even runs but hides subtle (and sometimes catastrophic) security flaws.



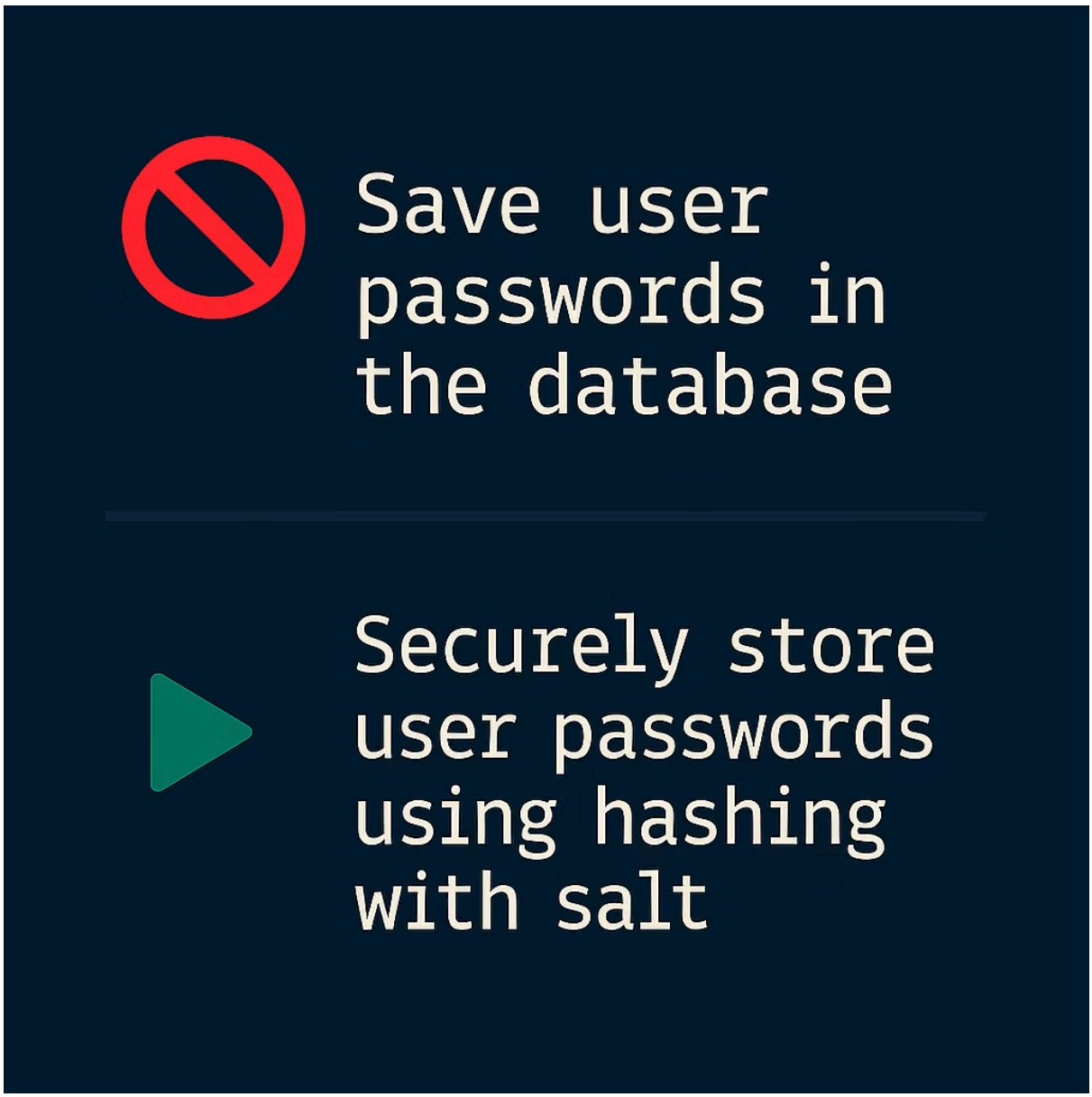
A developer prompting an Large Language Model (LLM) to generate code

Let's break down why "vibe coding" is a hacker's dream and how you can exploit it.

How vibe coding works (and why it's dangerous)

1) You provide a prompt

A developer describes what they want in natural language, for example, "Make a Python script that takes user input and saves it to a database." The art here to prevent vulnerabilities is to describe the functionalities of the application as extensively as possible and state how certain functions would need to be implemented to prevent security issues.



Security complications of vibe coding

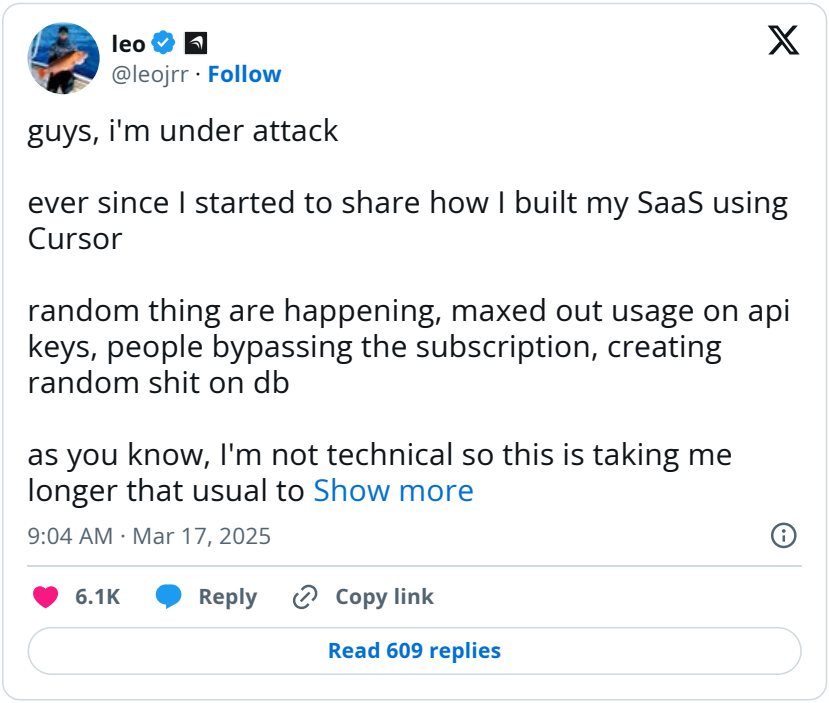
2) The AI "vibes" out code

Using all the knowledge it has, the AI model then generates the code, most of the time lacking the context security-wise. There are different problems with this because LLMs are trained on tons of code

repositories, but in those repositories are also vulnerabilities. So of course, when certain prompts are done to an AI, and certainly with vibe coding because people tend to build whole applications with one or more prompts, you're going to get vulnerabilities lodged in your application.

3) The developer deploys it

Most vibe coders assume the AI "knows" best and trust the code. Resulting in vulnerable code being pushed to production



Common vulnerabilities in vibe-coded apps: examples

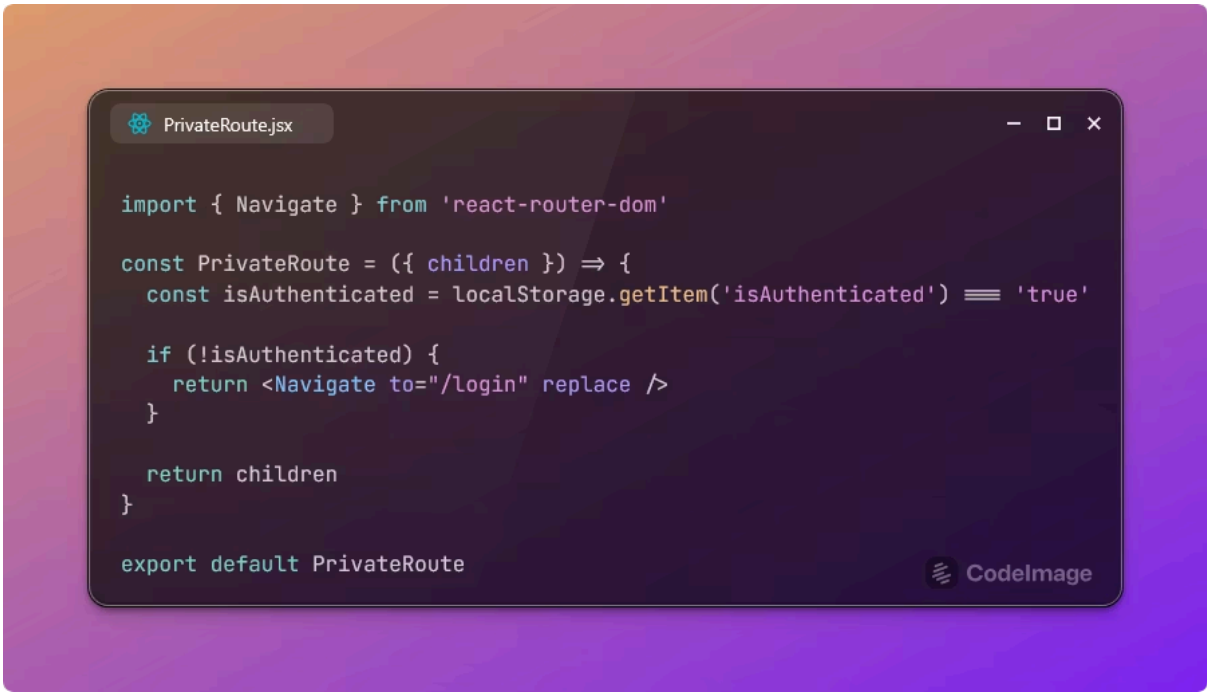
No input sanitization

Most LLMs do not take input sanitization, one of the most crucial foundations of having a secure app, into consideration. This makes injection attacks like SQLi and XSS possible.

Improper access controls

Let's take a look at another example of an LLM's output. This time we're prompting with Cursor, a widely used tool that integrates third-party LLMs from OpenAI, Anthropic, etc. into VSCode. We will be asking it to generate an admin dashboard, here's our prompt:

```
Make me an app that has a login and an admin panel in React. Make it pretty.
```



Improper access controls on AI generated code

Initially the code seems fine, but when we look deeper, we see that the only check that is done to get to the admin dashboard is to check if a property in localStorage is set to **true** (which can easily be manipulated by the end user). This code is unrealistic since the authentication is done entirely on the client side. The preferred approach to this would be to implement server-side authentication and authorization mechanisms and preferably use a token, like a JSON Web Token (JWT).

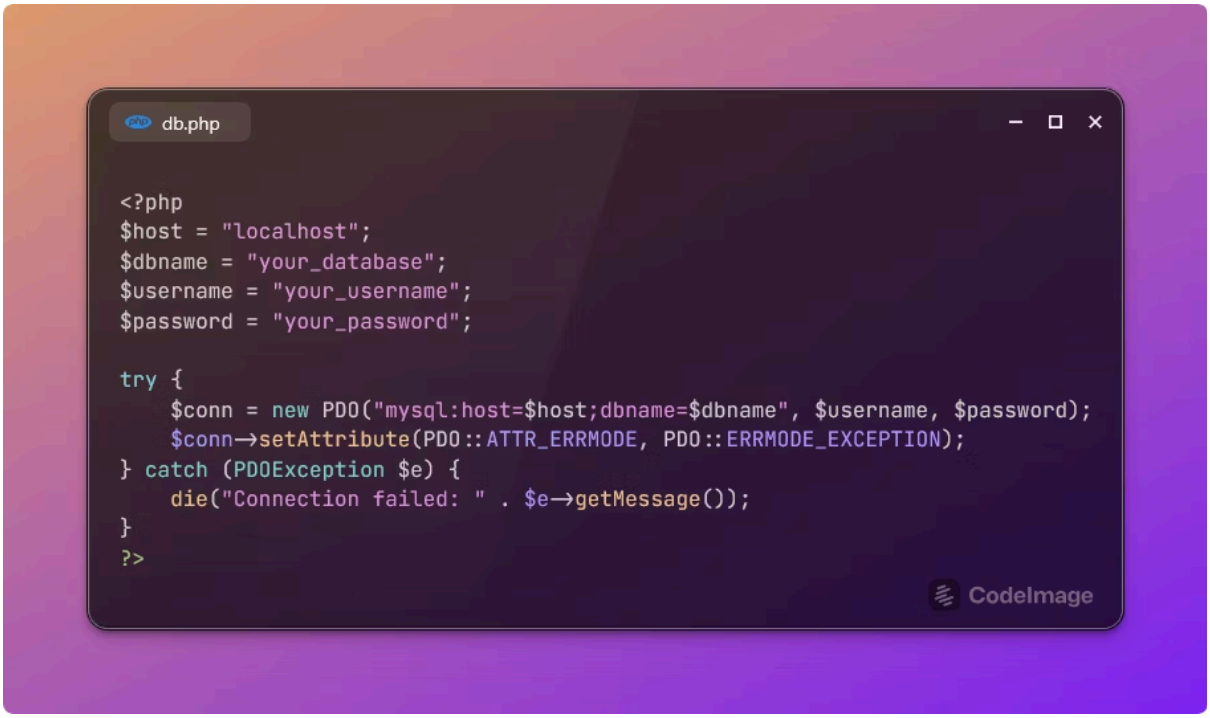
When the user logs in, the token would be generated for that user, and when trying to access a restricted page, the token would be given in the request, validated on the server-side, and if valid, the page would be made accessible to the user.

Sensitive secret exposure (hard-coded credentials)

Let's take a look at another example that mainly affects beginners and people who are coming from non-technical backgrounds. Most LLMs main objective is to provide an answer to your prompt, and that often does not include adhering to security best practices.

In this instance, we've asked ChatGPT to help us with developing an application in PHP that stores objects in our database:

```
I need to implement saving a flower object into a database in php.
```



Sensitive secret exposure (hard-coded credentials) in AI-generated code

As we can see, the AI says that we need to hardcode the database credentials in the same file where our main application logic is located. Newcomers will not be familiar with security best practices, such as storing sensitive secrets in secure environments, and will follow the example they're presented with instead. If this same file is ever made accessible, for instance, pushed to a public GitHub repository, placed in the web root folder, or through any other means, it would essentially allow malicious users to initiate a connection to the database and gain full control.

How to exploit vibe-coded apps

When it comes to exploiting these apps, it's very simple. Most of these apps don't comply with the best practices like we have seen so far. Here are some examples of how to go about actually attacking these applications.

Look for common AI-generated code patterns

If we look at the way AI forms code, we can see some significant similarities and patterns. Getting yourself familiar with these patterns can help you recognize applications like this instantly. Here are a few characteristics to look out for:

- Overly generic variable names (temp1, data2).
- Comments explaining how the code works: AI tends to over-explain basic logic, while critical security checks are missing.
- Lack of error handling (try/catch missing): A lot of times AI will not do proper error handling; therefore, you get generic error messages being thrown, which is interesting for an attacker to get a deeper insight into how the system works.

- Over-reliance on outdated libraries: AI may suggest deprecated libraries with known vulnerabilities (e.g., an old version of Mongoose).

Test for "obvious" flaws

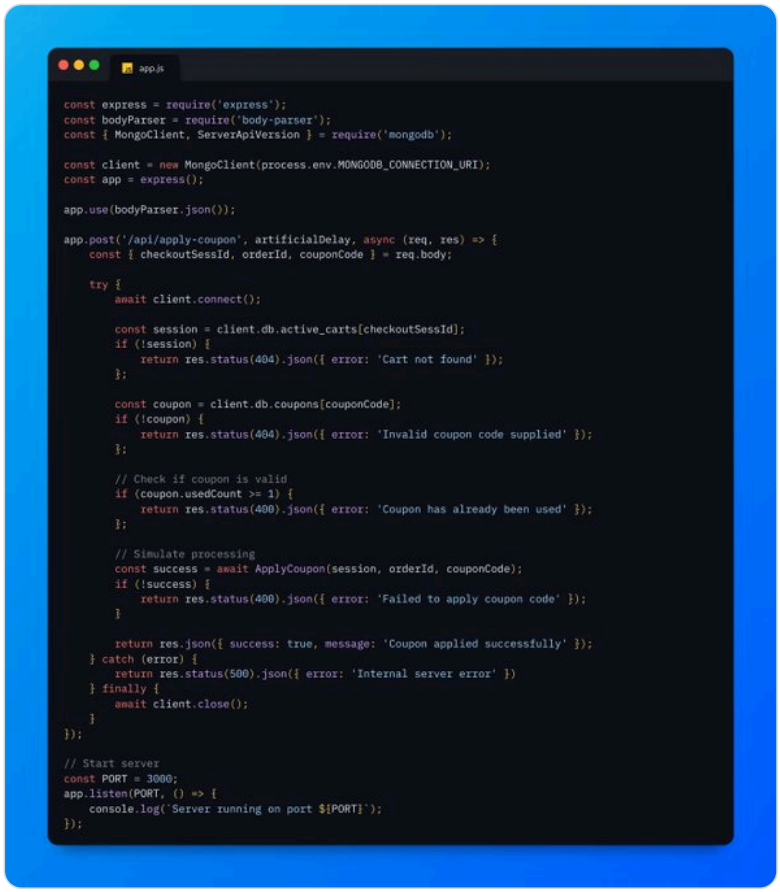
- **SQLi?** Throw a `' OR 1=1--` into every input.
- **XSS?** Drop a `<script>alert(1)</script>` or a polyglot string and see if it gives a popup or if you manage to find a symbol that breaks the application.
- **IDOR?** Change `/user?id=1` to `/user?id=2`.
- Check for missing input validation
- Look for errors leaking information
- API key leaks
- Logic errors

Here in this post that we did recently, which shows exactly how such a logic error could look. The AI lacks the context and doesn't include that the birth date can be changed as much as the user wants, resulting in a valid coupon code every day.

Intigriti
@intigriti · **Follow**

What issue can you spot in the following coupon code implementation?

How would you exploit it?



9:08 AM · Apr 13, 2025

95 Reply Copy link

Read 15 replies

Like we saw, the vulnerabilities that are in these generated applications are most of the time very easy ones, and the bigger the application, the more potential it has to be exploited. Most of these things are covered in our new Hackademy.

Scan for hard-coded secrets

Like we saw, AI likes to generate code that has hard-coded secrets and doesn't suggest using any environment variables or secret managers. This gives us the opportunity to maybe do some secret scanning on, for example, GitHub, or maybe it's simply leaking in the front end (JavaScript files with AWS S3 credentials as an example).

The future of vibe coding exploits: conclusion

As AI-generated code becomes mainstream, **bug hunters should:**

- **Automate detection** of AI code patterns.
- **Prioritize low-hanging fruit** (SQLi, XSS, secrets).
- **Monitor GitHub** for freshly pushed AI projects.

Developers should always review AI-generated code, especially security-critical parts. Make sure that every line of code is reviewed and that the aforementioned low-hanging fruit vulnerabilities are taken care of. This can be done by following industry standard guides like OWASP: https://cheatsheetseries.owasp.org/

You've just learned how to hunt for security vulnerabilities in vibe-coded applications... Right now, it's time to put your skills to the test! Browse through our 70+ public bug bounty programs on Intigriti, and who knows, maybe your next bounty will be earned with us!

Happy hunting. The era of vibe-driven vulnerabilities is here.