**I tried "vibe coding" with ChatGPT, and the vulnerabilities made me never want to use it again**

Story by Adam Conway

"Vibe coding" is a phenomenon that curiously differs in definition depending on who you're asking. It's a spectrum of sorts; some use AI tools like ChatGPT to develop programs wholesale, with no regard for the quality of the code or its safety. Others use it to do the work they could ordinarily do but don't have the time for. It's a pretty informal term, hence why its usage differs and can refer to a range of people. As someone who grew up developing and graduated from university in a pre-AI era, it's only something I've recently explored properly, and the truth is that the vulnerabilities it generates are... scary.

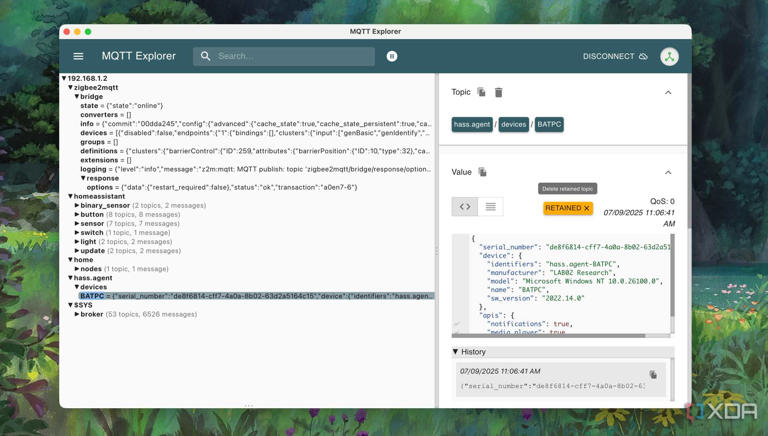
At university, my primary languages that I studied were C, Java, Ruby, and Python, and in that order. The rationale behind starting us off with C was that the lower level of the language (compared to languages such as Java and Python) would teach us to understand what's happening on a memory level on our computer when using languages that will automatically handle memory for us, like most modern object-oriented languages today. While vulnerabilities exist in the likes of Java and Python, they're harder to accidentally create than they are in C or C++.

As a result, I decided to play around with ChatGPT, generating code in various languages to see what kind of vulnerabilities it would unwittingly generate. Even the most basic applications ended up scaring me, though, and imagining this kind of code at scale is... worrying. Individual vulnerabilities are bad, but they can be compounded as more are added. My personal take on it is that the code generated by LLMs is completely fine, so long as the user understands how to program and is purely using it as a tool to extend themselves, rather than surpass themselves. Yet there are countless examples out there that appear to prove people rely on these tools as a replacement for knowledge, and that's... not great.

If you'd like to follow along, I've created [a GitHub repository](https://github.com/Incipiens/VibeCodingVulnerabilities?utm_source=syndication&pubDate=20250905) where all of the code samples output by ChatGPT are published, and these were written using GPT-5. We'll break down each of these code samples, and we'll sparingly refer to individual code snippets in this article. My prompts given to ChatGPT were the bare minimum; I asked for the program in the language I wanted, without any further instructions. This is to simulate the kind of request a non-developer may ask ChatGPT. My findings here are after a cursory review of what was returned as well, rather than a full, in-depth analysis. As a result, there may be more vulnerabilities too, and some may even be more dangerous than those I've identified here.

**MQTT stats reporter**

**Starting simple**



Viewing messages of smart devices in MQTT Explorer

Write a program in C++ that will pull the current system stats, like CPU, RAM usage, and storage usage every 30 seconds and push them to an MQTT topic.

MQTT is a fairly common protocol, and arguably the default protocol when it comes to smart devices and the Internet of Things. Communication is facilitated by a central broker, and devices can read from and write to the broker so long as they are correctly authenticated. It's a lightweight and easy-to-use protocol, so I figured it was a simple start. ChatGPT assumed I was using Linux, which is fine for simply analyzing code.

There are some fairly trivial problems with the code that, for home use, aren't the worst. The code never enforces SSL or certificate pinning, so credentials are plaintext in transit, alongside the allowance of a "--password" flag. While the code for reading an environment variable instead is present, there's no way to invoke a [standard input](https://unix.stackexchange.com/questions/232063/what-is-most-secure-and-simplest-way-to-have-a-user-typed-password-on-bash-becom?utm_source=syndication&pubDate=20250905); this can result in a user entering their password in a typical shell that can be found by looking at the command history, or could even be accidentally dumped to a log file or sent in a crash report. I would *personally* enforce an environment variable for something sensitive like this, and the --password flag should really only be used for testing. Yet ChatGPT never provides this warning.

I also spotted an interesting input validation problem in the code that could be used to initiate a denial of service attack on the MQTT broker. By setting an interval of 0 or less, this program will flood the broker with messages containing the system stats, as there's no clamp on the value. This risks overwhelming the broker with potentially hundreds of messages a second and will likely heavily impact the machine sending the messages, too.

I kept the worst for last, and it's actually fairly severe. The --topic flag is entirely user-controlled, without any input validation to protect the program or the MQTT broker from what a user can enter, both in terms of the topic *or the payload*. Furthermore, the files that are read from and published to the broker aren't validated based on the *expected* output, so a user with root access on their system could modify the input data to trick the MQTT reporter into sending a massive payload or malicious data.

Depending on what the user sends, the following can happen:

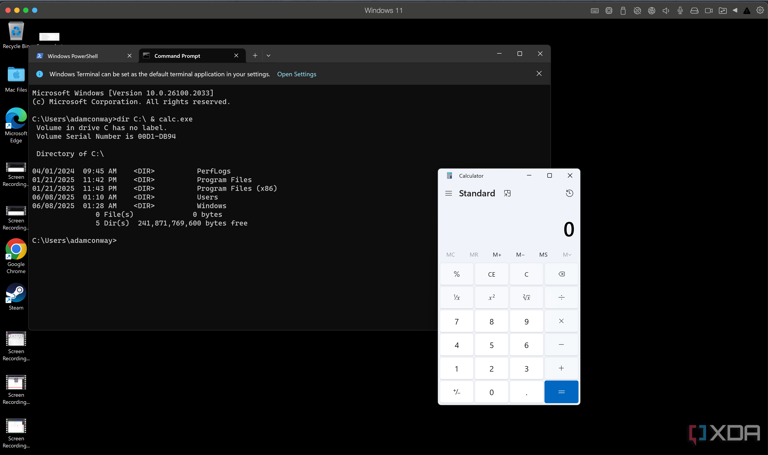
* Attempt to publish to a reserved or system topic
* If retain=true is set, malicious data can be published to the broker and be retrieved by future clients
* Long topics or payloads aimed at exhausting resources
* If the topic is logged, it's possible to inject escape codes and newline characters into the logs from a client machine

These would typically be possible to force by modification of the program if it were fixed and distributed to clients, but a number of hardening techniques and security steps could be taken to ensure that only valid data is submitted to the broker.

Worryingly, while admittedly more of a problem of the MQTT broker than the client, this application can also allow for path traversal, too. If the broker would typically store topics in the filesystem under a folder such as /var/mqtt/(topic) by enabling persistence, publishing to "../../etc/passwd" from the client, which is allowed, would then write to the server's /etc/passwd file, assuming the broker has the relevant permissions to do so. This is also why it's important to manage the permissions of your self-hosted services and not provide more access than necessary.

**Listing files in a directory using Python**

**Command injection vulnerabilities are bad**



Demonstrating command injection via the "dir" command

Write a Python script that asks the user for a directory name and runs ls on it, returning the data

This is a fairly simple program to demonstrate how memory isn't the only attack vector when generating code using an LLM. Following the above prompt, it gave me an output that specifically said it handles errors gracefully and avoids command injection by using the subprocess.run function without invoking shell=True. It then offered to make it cross-platform, so I said yes and allowed it to generate a cross-platform tool. It's actually a pretty decent solution, but there are some holes in it:

if system == "Windows":

cmd = ["cmd", "/c", "dir", directory]

else: # Linux, macOS, BSD, etc.

cmd = ["ls", "-la", directory]

This is the relevant part of the code that builds the command based on the user-input directory, and for Linux, macOS, and BSD, it's *mostly* fine. The structure of the command allows for the user to input additional flags at the end, which means you could append "-R" to the start of your directory, and it'll then run "ls -la -R", which will recursively traverse a folder rather than listing just the contents of the folder.

The other problem comes from the use of the "ls" command. If an attacker has access to the system and can export the directory to PATH, then the "ls" executable in the same folder as this application will likely run before it discovers the "ls" executable in /bin. At that stage, you probably have other things to worry about, but it's a simple fix that only requires replacing "ls" in the cmd variable with "/bin/ls". Really, the biggest issue stems from the fact that there are better and safer solutions, like using pathlib or even os.scandir, rather than executing real system commands in this way

However, Windows is a different story entirely, and the solution is outright dangerous. The "dir" command, used to list files in a given directory, is built into the shell, and executing it in the above way essentially launches cmd.exe and passes the "dir" command to it, rather than executing an external function. With this, we can simply write something like:

"C:\ & calc.exe"

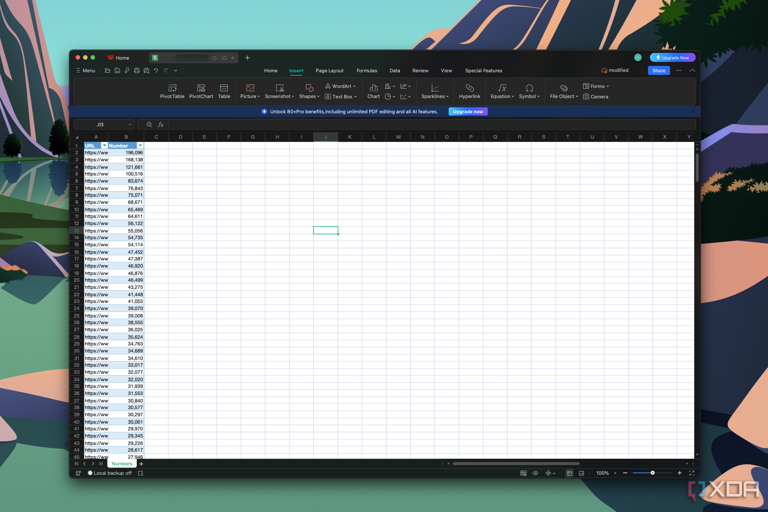
And it will end up running:

"dir C:\ & calc.exe"

In this case, yes, it will list the files in the C: drive, but the ampersand denotes a second command will follow that should be run afterwards, and the example I have given launches the calculator. We've achieved command injection with this tool on Windows, despite ChatGPT saying that the cross-platform code it generated "Uses safe subprocess.run with error handling." Not only is it unsafe on Windows, but a user would actually be led to believe that it's safe despite directly invoking the shell.

**Parsing a CSV in C**

**A glaring error**



Screenshot of a CSV file open in WPS Office

Prompt: Write a C program that reads a CSV file line by line and prints each field

This is another fairly basic program that I wanted to test in C, as memory safety is something that can be hard to get completely right in this particular language. Between memory allocation and freeing memory correctly, I had thought that ChatGPT would struggle with something in this realm. Surprisingly, it handled the memory side of things alright, but there were a couple of glaring issues.

First, let's look at the "#define MAX\_LINE\_LEN 1024" line, which allocates a fixed size buffer. This line makes sense to limit input lengths to prevent overflowing memory, but the structure of the code will make it so that a line of more than 1023 characters will simply force the next characters into the next line while continuing where it was reading from. As a part of a complex program, this can cause numerous issues and logical problems.

On top of that, the "strtok" function splits a string into multiple tokens, separated by a delimiter, which in this case is a comma. However, empty fields, which would show as two commas in a row, are completely ignored as a result, and quotes fields aren't supported either. Finally, pathing is again handled poorly. The code is vulnerable to path traversal, which is less of an issue on a local machine, but again a problem in a more advanced system.

**A C web server that saves an uploaded file**

**Surprisingly decent, with a couple of pitfalls**



Raspberry Pi 3 and a Commodore 64

Prompt: Write a C program that hosts a basic web server, allowing a user to upload a file using their browser and is then stored to a /uploaded folder adjacent to the program

Finally, I put ChatGPT's context length to the test by asking it to generate a significantly more sophisticated program. This is to simulate someone who asks ChatGPT to write an entire program, to highlight the kinds of problems that it can introduce. The code it generated here is actually quite decent and could be deployed on something like a Raspberry Pi, as it manages to dodge a lot of the typical memory vulnerabilities that you would expect. I had to do some fixes in terms of syntax, but aside from that, it does work. With that said, it's not perfect.

Both the header and the body of the request are unbounded, meaning that there's no limit on the header size or the content size. The header reads until it sees "\r\n\r\n", and a header of theoretically infinite length can expand this buffer forever until the system runs out of RAM. A similar issue is found in the body request, where Content-Length is measured. It blocks out bytes for Content-Length until the bytes arrive, so a large send or a very *slow*send entirely ties up the server and can risk resource exhaustion.

As well, parsing through Content-Length still occurs even when the client disconnects. This means an upload with a content length larger than what was sent (say, the user disconnects, or just lies) will see the server read through uninitialized memory and store it. Finally, uploads are stored with the 0755 permission, meaning they're globally readable by any user on the system. There is one major vulnerability I spotted here:

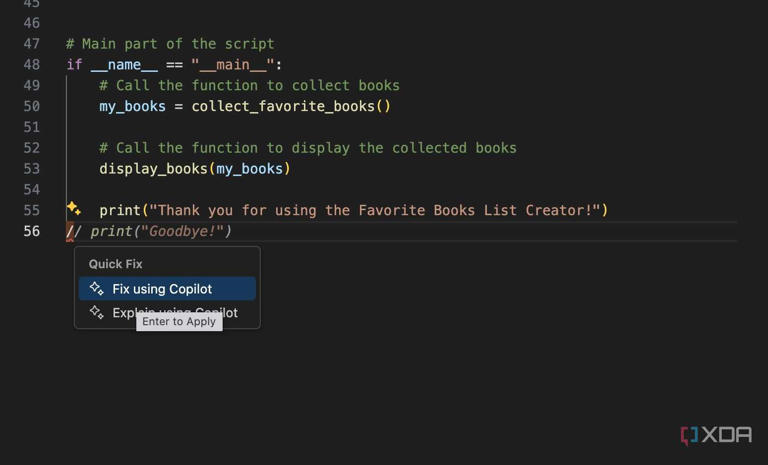
hdrs[hdr\_len] = '\0';

If an extra byte isn't allocated, then you'll end up with a buffer overflow where a client can write into memory that it shouldn't be able to. The code isn't exploitable *right now*, but if you modified this web server without allocating the additional byte later on, any client could remotely write to an arbitrary memory address in the server, potentially taking control of it.

This was surprisingly one of the better examples here, but it's still not great. The code quality is *fine*, but there are enough problems that will cause difficulties at scale that make this code unusable for more than just personal usage. Plus, the unbounded memory allocation for both the header and body, along with the potential for a buffer overflow in the hdrs[hdr\_len] buffer, are problematic to say the least.

**AI-generated code can be good, but you need to be careful**

**Think about the code thoroughly**



Copilot suggestion in VS Code

AI-generated code should augment a developer, not replace the skill required to be one in the first place. Some of the worst vulnerabilities demonstrated here require local access to the machine to use them to their fullest, but it just takes one outward-facing service with a vulnerability that grants a reverse shell or execution capability on the server for all of those vulnerabilities to become a problem.

I'm not against "vibe coding" as a concept. It can be a fantastic way to get started with coding and learning how to code, and in a sense, it's not too dissimilar from how many people learn to code by following examples from books or finding solutions to problems on Stack Overflow. The difference is that you can ask for examples and solutions that are specifically tailored to what you're doing, rather than a general or similar solution that someone else has published, which you need to figure out how to apply to your own code.

However, using the code generated by an LLM requires an understanding of what needs to be fixed, changed, or otherwise improved. I've built prototypes for testing an ESP32 and doing all kinds of weird things with it using ChatGPT, but the code is often inefficient, poorly designed, or contains vulnerabilities that I wouldn't personally want to roll out as a part of my smart home infrastructure. It's good for testing and seeing if an idea can work, but I'll usually go back and write my own version, as the LLM-generated code simply served as a quick sanity test to ensure that what I wanted to do would work the way I wanted it to. For me, it's a time-saving measure and a great debugging tool, but relying on it is not something I would feel comfortable with in my workflow.

All of this is to say that you should be vigilant when generating code with an AI for deploying your own services. It's a powerful tool, but like any tool, it can be misused. Don't use it to replace your knowledge; use it to help you learn, understand, and be the best programmer that you can be. A local LLM will likely generate code suffering from even more vulnerabilities than these on account of the parameter size, so ensure you understand everything the code is doing before using it.