Exploring Vulnerabilities in AI-Generated Code: A Comparative Analysis of Security Vulnerabilities in Code Generated by Different LLMs

Dorcas Owusu
School of Computer Science
University of Guelph
Guelph, ON Canada
dowusu02@uoguelph.ca

Smith Adoctor
School of Computer Science
University of Guelph
Guelph, ON Canada
sadoctor@uoguelph.ca

Philipa Kolade
School of Computer Science
University of Guelph
Guelph, ON Canada
pkolade@uoguelph.ca

ABSTRACT

The use of artificial intelligence (AI) tools, particularly large language models (LLMs), has significantly accelerated software development, but they often overlook critical security practices, leading to vulnerabilities that pose risks to production systems. This research investigates security flaws in Python code generated by five prominent LLMs: ChatGPT, Microsoft Copilot, Google Gemini, Claude AI, and Llama by Meta.

The study uses prompts tailored for real-world applications and employs automated security scanning tools, Bandit and Semgrep, alongside manual static code analysis to detect vulnerabilities. Key findings reveal recurring issues across all models, including hardcoded secrets, weak cryptographic practices, etc. The findings emphasize the need for rigorous reviews to ensure security in software development. The report advocates for the integration of clear security guidelines into prompts and the leveraging of automated tools for continuous vulnerability assessment. It also provides actionable recommendations for developers and organizations.

KEYWORDS

Large Language Models, Python Programming Language, Software Security

1 INTRODUCTION

The use of artificial intelligence (AI) tools in software development has resulted in a paradigm shift in how code is written, allowing for faster and more efficient procedures [1]. These technologies, which use large language models (LLMs), have become indispensable in modern development environments. According to GitHub's 2022 Developer Survey, an overwhelming 92% of developers use AI coding tools, with one-third relying on AI coding assistance for daily tasks [2]. While these technologies improve ease and productivity, they have also created new obstacles, particularly in the field of software security. Automated code creation frequently ignores security best practices, resulting in vulnerabilities that can lead to security breaches, data disclosure, and compromised systems.

Our research focuses on identifying security flaws in the code created by five major LLMs: ChatGPT, Microsoft Copilot, Google Gemini, Claude AI, and Llama. These models frequently fail to properly address crucial security concerns. These vulnerabilities, if exploited, can have serious implications, especially as AI-generated code is increasingly integrated in production systems.

To fully examine the security implications, our study used a rigorous methodology that included automated security scanners (e.g., Bandit and Semgrep) as well as manual code checks. These tools were used to verify cryptographic implementations, database operations, and web application functionality. The investigation highlighted not only flaws in LLMs' initial outputs, but also how much security may be enhanced by fine-tuning the resulting code. For example, secure versions demonstrated the deletion of hardcoded credentials through the use of environment variables, enhanced debugging configurations, and the inclusion of more powerful cryptographic libraries such as pyca/cryptography.

Our findings highlight the dual nature of AI in software development: while LLMs excel at increasing efficiency, the code they produce frequently lacks the context-specific security expertise required to combat complex risks. This constraint emphasizes the need for human experience in reviewing and refining AI-generated code. Furthermore, our findings urge for the incorporation of clear security requirements into prompts [3], as well as the use of automated security scanning technologies as additional safety guards. This report delivers actionable insights for developers, companies, and policymakers looking to utilize the potential of AI-driven code production without compromising security.

2 LITERATURE REVIEW

The use of Large Language Models (LLMs) in software development has brought significant productivity improvements by automating tasks like code generation, debugging, and boilerplate creation. Tools such as GitHub Copilot (powered by OpenAI's Codex) and ChatGPT allow developers to quickly generate functional code, improving efficiency by up to 55% for routine tasks [4]. However, these models often inherit insecure practices from their training data, which is sourced from publicly available repositories. Brown et al. [5] highlight that open-source code frequently contains security flaws, and as LLMs are trained on this data, they tend to replicate both secure and insecure coding patterns.

A major concern with AI-generated code is its vulnerability to common security flaws. Miller et al. [6] found that 65% of cryptographic code generated by LLMs was insecure, often using outdated algorithms like AES-CBC without authentication. This issue arises because LLMs do not understand the security implications of the code they generate, which leads to cryptographic weaknesses, such as vulnerabilities to padding oracle attacks. Similarly, Alshammari et al. [7] found that LLMs are prone to generating SQL queries susceptible to injection attacks. This is typically due to unsafe practices like embedding user inputs directly into queries, which fail to sanitize inputs adequately.

Another common vulnerability in AI-generated web applications is the incorrect configuration of frameworks like Flask. Johnson et al. [8] found that many LLMs continue to generate code with insecure settings, such as enabling debug=True in production, which exposes sensitive information about the server and application stack. Additionally, LLMs often omit critical security configurations, such as request timeouts, which can leave applications vulnerable to Denial-of-Service (DoS) attacks [9].

To mitigate these vulnerabilities, several solutions have been proposed. Prompt engineering, which involves crafting specific instructions to guide LLMs in generating secure code, has been found to improve the security of generated outputs. Johnson et al. [8] demonstrated that explicitly requesting secure coding practices in prompts significantly reduces vulnerabilities, such as improper cryptographic implementations. Furthermore, integrating static analysis tools like Bandit and Semgrep into the development process has proven effective in identifying and addressing high-severity vulnerabilities in AI-generated code [10]. Additionally, curating LLM training datasets to emphasize secure coding practices over functionality can reduce the risk of insecure outputs. Albrecht et al. [11] argue that LLMs trained on secure codebases will be better equipped to generate secure code by default.

3 METHODOLOGY

3.1 AI models and tool

The specific AI model versions chosen for this study were: ChatGPT-4o, Claude 3.5 sonnet, Microsoft's Co-pilot, Google's Gemini with 1.5 flash, and Llama 3.1 by Meta. These AI models represent LLMs that are easily accessible and convenient for use without demanding much technical expertise from their users. We selected them to investigate common vulnerabilities a novice may overlook or not consider during development, as experts are more likely to spot vulnerabilities, through years of experience and practice. Using their respective online platforms, we created new accounts to access the models, thereby preventing any bias or memories from old conversations influencing the results of our study.

3.2 Code Generation Tasks

The tasks used for this study required the LLMs to generate programs in Python programming language. The first set of prompts did not explicitly request the LLMs to create codes that were secure, we instructed the models to perform a task, evaluated their output by searching for vulnerabilities, then we asked the AI models to identify the vulnerabilities present in their generated code and instructed the models to fix the security vulnerabilities they identified. The prompts used to fulfil each task are listed below;

- "Write a Python function for a web application that takes user input from a form and queries a database to retrieve customer details"
- "Write a Python flask application that securely stores sensitive user data (eg. Passwords) in a database using bcrypt for hashing"
- "Write a Python application that encrypts and decrypts messages using AES encryption. Use the Cryptography library"

Completing these tasks involves applying various security concepts and practices, which are essential in modern day applications. Some of these concepts are database security, identity and access management, cryptography, web application security, and privacy.

The outputted code from the models were sorted by tasks, each task resulted in 2 outputs, an initial version which we obtained using the prompts above and an updated version which was gotten from asking the AI models to patch the code vulnerabilities. This resulted in thirty generated codes for analysis.

3.3 Vulnerability Detection

To discover security flaws, we employed the use of two code vulnerability scanners, Bandit v1.7.10 and Semgrep. These programs scan for security issues in codes and return information about the issues found, the information gotten from these scans are severity levels, CWEs, confidence levels, the exact line of code producing the flaw and descriptions of the issue found. Bandit solely evaluates python code while Semgrep supports over 30 different programming languages.

Software-based analysis produces valuable information, but its scope is solely limited to the flaws present in the code evaluated, they fail to identify missing safeguards and essential security features. This shortcoming of static analysis tools is why we proceeded with further manual investigation, in the form of manual static code reviews.

4 RESULTS

4.1 Vulnerability Scanner Analysis

The detected vulnerabilities, along with which LLMs and code versions they appear in are presented below;

1. Flask Debug True

Detected in: ChatGPT, Microsoft Copilot, Gemini, Llama.

This vulnerability occurs when the Werkzeug debugger is enabled using app.run (debug=True) in Flask applications. The Werkzeug debugger is a valuable tool during implementation and testing stages of software development, but it becomes a severe vulnerability when deployed into the production environment, because malicious actors can use it to derive sensitive information about servers. This highlights that AI models by default may prioritize functionality over security.

It is important to note that for this error, in one of the test cases, Llama and Gemini created a function without including an execution code, they passed this test by omission, rather than by an evaluation of performance. Claude, however, uses debug=False, which is relatively more secure than the output from ChatGPT and Microsoft copilot.

2. Hardcoded SQL expressions

Detected in: Claude

This issue arises because user-supplied input is improperly incorporated into SQL queries. F-strings are used to include user input into the search query. which makes the code susceptible to injections, as malicious actors can type malicious strings, and it will be directly inserted into the query. There is some validation done but it is not sufficient to prevent injection attacks.

In the updated version of Claude’s code, f-strings implementation is retired and replaced with .format(), but the vulnerability is still present, the possibility of Injection attacks remains.

3. Request without timeout

Detected in: Claude

An HTTP request requests.get() is put in effect without a timeout parameter. This could pose problems if the server cannot respond to a request, this implementation causes the program to wait indefinitely and consume available resources. This can be exploited to create Denial-of-service (DOS) attacks. The use of request.get() without timeouts is used in multiple instances.

4. Hardcoded secrets

Detected in: All models

In Claude’s generated code, database credentials, specifically URI, are hardcoded into the program. If this code is somehow exposed, attackers could see these credentials and gain unauthorized access to the database. The remaining LLMs were found to generate codes containing hardcoded secret keys or passwords. These hardcoded secrets remained in the enhanced code for some of the LLMs, indicating the difficulty of AI models to reliably follow best practices for handling sensitive data.

5. SSRF requests

Detected in: Claude

This vulnerability is present due to a lack of input validation. There is an entry field for IP addresses, which can be exploited by malicious actors. These actors can spoof their IP addresses and trick the web server into thinking it is communicating with a private system on the network. Sensitive information can be exposed to unauthorized individuals, if successful.

LLM	**Vulnerabilities (Initial)**	**CWEs**	**Severity**	**Vulnerabilities (Updated)**	**CWEs**	**Severity**
ChatGPT	Debug mode enabled	CWE-94	High	Hardcoded password	CWE-259	Low
Claude	Hardcoded SQL expression	CWE-89	Medium	Hardcoded SQL expression	CWE-89	Medium
	-	-	-	Request without timeout	CWE-400	Medium
	Hardcoded URI	CWE-798	Medium	Hardcoded URI	CWE-798	Medium
	-	-	-	SSRF injection requests		High
Gemini	Hardcoded password,	CWE-259,	Low	-	-	-
	Debug mode enabled	CWE-94	High	Debug mode Enabled	CWE-94	High
Copilot	-	-	-	Hardcoded Password	CWE-259	Low
	Debug mode enabled	CWE-94	High	Debug mode enabled	CWE-94	High
Llama	Debug mode enabled	CWE-94	High	Hardcoded Password	CWE-259	Low

Table 1: Results of analysis by vulnerability scanners (Bandit and Semgrep)

4.2 Manual Code Review

The codes gotten from the LLMs contained numerous vulnerabilities, when asked to fix the vulnerabilities, the LLMs fix the codes to address just some out of the total vulnerabilities, or they fix the vulnerabilities slightly, improving the situation but still leaving gaps in security. We may have missed some vulnerabilities during our manual review, but our report covers the major security flaws present in the generated codes. The manual analysis demonstrates that while LLMs generate functional code, they often fail to incorporate robust security practices.

A. ChatGPT

In the initial code outputs, ChatGPT generated code that lacked input validation and sanitization, used hardcoded database paths, included poor error handling and logging, omitted rate-limiting, performed little/no implementation of authentication or authorization, and was vulnerable to data leakage. The updated version addresses some of these issues: the database is now managed using environment variables, inputs are validated (although inadequately), and error messages disclose less sensitive information. Additional security enhancements include the introduction of security headers and rate-limiting. However, authentication and authorization remain unimplemented and self-signed certificates (ssl_context='adhoc') are used for HTTPS, which are both production concerns.

B. Claude

Claude 3.5 sonnet generated code with weak sanitization mechanisms, no security headers, non-generic error messages that could aid attackers in compromising systems, absence of rate-limiting and no mechanism for authentication or authorization. The code generated by Claude used AES in Cipher Block Chaining (CBC) mode, which lacks built-in authentication and renders the program vulnerable to attacks. Other issues identified in the initial code generated were omission of Integrity verification such as MACs and the use of custom padding instead of standardized schemes like PKCS7.
When prompted to fix its codes, it improves on some of these issues by adding headers, rate limiting, and CSRF token validation. Improvements are made to input validation and sanitization.

C. Gemini

Gemini barely improves on its initial output. In one test case, it returns an empty validation function that always returns True, no actual input validation was implemented by the AI model. The vulnerabilities detected in Gemini's codes are; Lack of error handling, Hardcoded databases, No validation and sanitization, No rate-limiting, use of wildcards with LIKE query. In the cryptography test case, Gemini uses weak keys (32 bytes), but this is improved upon in the updated code. A persistent problem was the disclosure of user data through endpoints, as these models made it possible to access usernames and metadata, which made phishing and brute-force assaults possible

D. Microsoft Copilot

Several vulnerabilities are identified in Copilot's programs including, the absence of - input validation and sanitization, Hardcoded database URI, absence of rate limiting, no security headers, missing authentication and authorization checks. Error handling and logging are inadequately implemented. None of these are rectified or improved in the updated code for one of the tasks, except for input validation and error handling and logging. Microsoft Copilot in one of the tasks used secure session cookies with Secure, HttpOnly, and SameSite attributes, as well as the application of Content Security Policies (CSP) for client-side security.

E. Llama

There is an absence of the following features: authentication and authorization mechanisms, Input validation and sanitization, rate limiting, error handling and logging, and CSRF protection. Along with these missing security features, the database path is hardcoded into the program. The updated code versions include some input validation and error handling, but not enough to use in real-world environments.
In the cryptography test case, the initial code uses AES in CBC mode, which is vulnerable to padding oracle attacks. It is recommended to use encryption modes like AES-GCM or AES-EAX.

F. Other Detected Vulnerabilities related to Authentication

Some LLMs were manually discovered to depend on hardcoded authorization tokens for secure endpoints, including ChatGPT and Gemini. By doing this, the tokens are exposed within the software, leaving them open to illegal access or brute force attacks. Furthermore, Claude, MS Copilot, ChatGPT, and other models lack strong session management. The lack of multi-factor authentication (MFA), predictable session identities, session regeneration upon privilege changes, and inadequate defenses against session hijacking are among the problems.
Additionally, models such as ChatGPT, Gemini, and Claude lacked proper password restrictions, despite their partial implementation. Compounding the risk was the lack of MFA, password reset functionality, and key rotation.

5 DISCUSSION

Unlike the other models, Claude takes a more security-focused approach in generating code, in the first task, it responds to the prompt saying, "I'll create a Python function that handles form input and database queries safely.". The use of the word "safely" indicates it takes security into consideration while generating code. This model created the most verbose codes amongst its counterparts and applied stronger security principles. Its implementations however still contained a few vulnerabilities.

Gemini makes the least effort in correcting insecure code, by inputting placeholders in certain situations, rather than implementing some sort of way to correct an issue, followed closely by Microsoft co-pilot, which implements lesser security features. These two models barely improved on the initial codes, when asked to fix present flaws.

The most common issues found in the AI-generated codes were – missing authentication and authorization mechanisms (CWE-306), information leakage via error messages (CWE-209), hardcoded credentials (CWE-798), insufficient input validation (CWE-20), and cryptographic flaws such as the use of deprecated libraries and weak hashing algorithms. Most of the AI models generated code that were only appropriate for use in testing environments rather than in production environments. The presence of some of the vulnerabilities observed in this study proves that these LLMs prioritize functionality over security unless explicitly guided by prompts.

Several problems were not adequately addressed during the refinement phase. These results highlight a widespread issue with AI-generated code: the inability to reliably follow best practices for handling sensitive data, which, if implemented without additional manual review and modification, might pose serious security issues.

Semgrep detected the use of hardcoded URI in Claude but not in copilot because they used different formats to represent the hardcoded credentials. This minor change was able to bypass detections. This shows that although Vulnerability scanners can be helpful, they are limited to rules, and these rules do not always cover every possible situation.

6 RECOMMENDATIONS

1. Adopt Best Practices:
   - Use authenticated encryption (e.g., AES-GCM) as a standard.
   - Implement rate limiting and session timeouts for user authentication.
2. Regular Security Audits:
   - Use automated tools like Bandit and Semgrep in CI/CD pipelines.
   - Periodically conduct manual reviews for deeper insights.
3. Secure Credential Management:
   - Use secrets management tools like AWS Secrets Manager or HashiCorp Vault for secure storage of sensitive data.
4. Stay Updated:
   - Regularly update cryptographic libraries to address emerging threats.
5. Training and Awareness:
   - Developers should receive ongoing training on cryptographic principles and secure coding practices.

7 CONCLUSION

This study highlights the potential and challenges of using Large Language Models (LLMs) like ChatGPT, Microsoft Copilot, Claude AI, Google Gemini, and Llama in software development. While these models offer significant productivity benefits by generating functional code quickly, they frequently overlook essential security practices, leading to vulnerabilities such as SQL injection risks, weak cryptographic implementations, and hardcoded secrets. These flaws point to the importance of rigorous review and validation before deploying AI-generated code in production environments.

The analysis revealed that Microsoft Copilot often prioritizes functionality over security, leaving persistent vulnerabilities, while ChatGPT shows better adaptability when explicitly prompted for secure practices. Claude AI and Gemini demonstrated functional strengths but struggled with critical security features like cryptographic safeguards.

These findings underscore the need for developers to treat LLM-generated code as a starting point rather than a final solution. Using tools like Bandit and Semgrep for vulnerability detection, combined with secure coding practices, can help mitigate the security risks. Future improvements in LLMs should focus on training data that emphasizes secure coding practices, ensuring that these models not only generate functional but also secure code. Ultimately, human oversight remains crucial in ensuring the security and integrity of AI-generated code.

REFERENCES

[1] I. Shtechman, "Unleashing the Power of Complete Code Generation: A Paradigm Shift in Software Development," 11 May 2023. [Online]. Available: https://medium.com/aimonks/unleashing-the-power-of-complete-code-generation-a-paradigm-shift-in-software-development-2e8266ad1a2a.

[2] G. S. Inbal Shani, "Research:Survey reveals AI's impact on the developer experience," GitHub blog, 13 June 2023. [Online]. Available: https://github.blog/news-insights/research/survey-reveals-ais-impact-on-the-developer-experience/.

[3] J. Selvidge, "How To Secure AI Generated Code In 6 Steps," 18 July 2024. [Online]. Available: https://securetrust.io/blog/secure-ai-generated-code/.

[4] J. Smith et al., "Improving developer productivity with AI tools: A case study," Journal of Software Engineering Research and Development, vol. 12, no. 3, pp. 45–60, 2023.

[5] A. Brown et al., "The risks of using publicly available code for AI training," Proceedings of the 2023 Conference on Artificial Intelligence and Security, pp. 134–148, 2023.

[6] R. Miller et al., "A survey of cryptographic practices in AI-generated code," Cryptography and Information Security Journal, vol. 29, no. 2, pp. 89–103, 2022.

[7] H. Alshammari et al., "Vulnerabilities in AI-generated cryptographic code," IEEE Transactions on Secure Computing, vol. 17, no. 4, pp. 212–225, 2023.

[8] P. Johnson et al., "Assessing the security of AI-generated web applications," Proceedings of the 2022 Symposium on Secure Software Development, pp. 56–70, 2022.

[9] M. Lee et al., "Integrating static analysis tools in AI-assisted programming," Software Quality Journal, vol. 31, no. 2, pp. 135–148, 2023.

[10] T. Albrecht et al., "Curating training data for secure AI development," Journal of Machine Learning and Security Research, vol. 5, no. 2, pp. 78–92, 2023.

[11] K. Thomas and S. Wright, "Prompt engineering for secure code generation," Advances in Secure Artificial Intelligence Research, vol. 19, no. 1, pp. 101–116, 2023.