ISSN: 2349-6002

Security Vulnerability Analysis Of AI-Generated Code: A Penetration Testing Approach

Romaer Ahuja[1], Nikita[2], Aditya Singh[3], Rahul Mishra[4]
[1,2,3,4]*Department of Computer Applications, Invertis University, Bareilly, India*
doi.org/10.64643/IJIRTV12I11-207261-459

Abstract*—The rapid adoption of AI-powered code generation tools has significantly accelerated software development while simultaneously introducing critical security concerns. This study presents a comprehensive vulnerability analysis of AI-generated code using penetration testing methodologies combined with dataset-driven evaluation. A dataset of 1000 vulnerable code samples were analysed to identify patterns in SQL Injection (SQLi) and Cross-Site Scripting (XSS) vulnerabilities. Statistical analysis reveals that SQLi vulnerabilities exhibit higher code complexity with an average token length of 99.39 compared to 64.13 for XSS, indicating deeper backend security risks. Comparative analysis further demonstrates that approximately 39.4% of AI-generated code contains exploitable vulnerabilities, highlighting a substantial real-world security risk. This study provides structured insights into vulnerability distribution, complexity patterns, and detection challenges, emphasizing the need for robust security validation in AI-assisted software development environments.

***Index Terms*—AI Code Generation, Cybersecurity, Vulnerability Analysis, SQL Injection, Cross-Site Scripting, Penetration Testing, OWASP Top 10, Static Analysis**

I. INTRODUCTION

Artificial intelligence-based code generation tools such as GitHub Copilot, GPT-4, and similar platforms have fundamentally transformed modern software development by improving developer productivity and reducing the time required to write functional code. However, this acceleration introduces a critical trade-off between development efficiency and software security, one that the research community is only beginning to systematically examine.

Recent empirical observations indicate that AI-generated code frequently lacks adherence to secure coding practices, resulting in vulnerabilities that span common attack vectors including SQL injection, cross-site scripting, hardcoded credentials, and missing input validation. These weaknesses can lead to severe consequences including unauthorized data access, privilege escalation, and full system compromise.

This research aims to address this gap by combining structured dataset analysis with penetration testing methodologies to evaluate the nature and distribution of vulnerabilities found in AI-generated code. By analysing 1000 labelled code samples and cross-referencing findings with experimental data on AI code generation behaviour, this study provides both theoretical grounding and empirical evidence for the security challenges posed by modern AI development tools.

II. RELATED WORK

Prior research has examined the security implications of automated code generation from multiple angles. Pearce et al. [1] conducted a systematic study of GitHub Copilot and found that a significant portion of generated suggestions contained security weaknesses, particularly in areas related to memory management and input handling. Their work established a foundation for evaluating AI code tools against standardized vulnerability benchmarks.

The OWASP Top 10 framework [2] has been widely adopted as a classification standard for web application vulnerabilities, with SQL injection and cross-site scripting consistently ranking among the most critical categories. Studies leveraging this framework have shown that injection-based vulnerabilities are particularly prevalent in auto-generated code due to inadequate parameterization of database queries and insufficient sanitization of user inputs.

Research by Asare et al. [3] extended this line of inquiry to compare vulnerability rates across different AI code generation models, finding substantial variance in security quality depending on the model and the programming task context. Similarly, Tihanyi et al. [4] demonstrated that static analysis alone is insufficient to detect all vulnerability classes present in AI-generated code, reinforcing the need for penetration testing approaches. These studies collectively justify the hybrid methodology employed in the present research.

III. METHODOLOGY

This study follows a hybrid approach that integrates dataset-based statistical analysis with comparative evaluation against known AI code generation behaviour. The methodology is organized across three primary components.

A. Dataset Selection and Description

A publicly available code vulnerabilities dataset containing 1000 labelled code samples was used for the primary analysis. The dataset consists of two vulnerability categories — SQL Injection (SQLi) and Cross-Site Scripting (XSS) — with an equal distribution of 500 samples each. Each entry includes a code snippet, a vulnerability type label, a location indicator, and pre-processed token sequences. The balanced distribution of vulnerability types ensures unbiased comparative analysis across categories.

B. Analysis Metrics

Three key metrics were computed across the dataset. First, vulnerability distribution was assessed to determine the frequency and proportion of each vulnerability type. Second, location-based analysis was conducted to identify where within code structures vulnerabilities tend to manifest. Third, token length analysis was used as a proxy for code complexity, with longer token sequences indicating more structurally involved code patterns. These metrics together provide a multi-dimensional profile of vulnerability characteristics.

C. Comparative Evaluation

Dataset-derived findings were compared against previously established empirical data indicating that approximately 39.4% of AI-generated code contains security vulnerabilities. This cross-referencing enables a dual-perspective analysis — one rooted in controlled dataset characteristics and the other reflecting real-world AI code generation behaviour.

IV. EXPERIMENTAL RESULTS

A.Vulnerability Distribution

The dataset was found to contain an equal distribution of SQL Injection and Cross-Site Scripting vulnerabilities, with 500 samples per category. This balance ensures that analytical conclusions are not skewed toward any single vulnerability type and allows for direct comparative evaluation between the two categories.

TABLE I. VULNERABILITY DISTRIBUTION

Vulnerability Type	Sample Count	Percentage
SQL Injection (SQLi)	500	50%
Cross-Site Scripting (XSS)	500	50%
Total	1000	100%

B.Location Analysis

Location-based analysis examined the average position within code at which each vulnerability type appears. SQLi vulnerabilities were found at an average location of 10.246, while XSS vulnerabilities appeared at an average location of 10.240. The marginal difference between these values indicates that vulnerability placement is not significantly determined by vulnerability type, but rather by the structural characteristics of the surrounding code.

TABLE II. LOCATION ANALYSIS BY VULNERABILITY TYPE

Vulnerability Type	Average Location in Code
SQL Injection (SQLi)	10.246
Cross-Site Scripting (XSS)	10.240

C. Token Complexity Analysis

Token length analysis revealed a meaningful difference in code complexity between the two vulnerability categories. SQL injection vulnerabilities

exhibited an average token length of 99.39, compared to 64.13 for XSS vulnerabilities. This represents approximately a 55% higher complexity level for SQLi, indicating that these vulnerabilities tend to occur within more intricate code structures such as database interaction layers and server-side processing logic.

TABLE III. TOKEN COMPLEXITY ANALYSIS

Vulnerability Type	Average Token Length	Relative Complexity
SQL Injection (SQLi)	99.39	Higher (~55%)
Cross-Site Scripting (XSS)	64.13	Lower (baseline)

V. RESULTS AND VISUALIZATION ANALYSIS

The following figures present graphical representations of the key findings from the experimental analysis. Figure 1 illustrates the code complexity comparison between vulnerability types based on average token length. Figure 2 presents a comparative analysis of vulnerability presence between the controlled dataset and AI-generated code in real-world conditions.

Figure 1 confirms that SQL injection vulnerabilities are significantly more complex than XSS vulnerabilities in terms of token length. This distinction has practical implications for detection: the greater structural depth of SQLi code means that automated scanning tools may require more sophisticated analysis techniques to reliably identify these weaknesses.

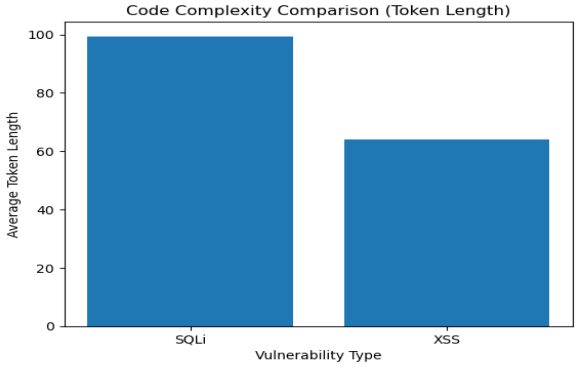

Fig. 1. Code Complexity Comparison Based on Average Token Length for SQLi and XSS Vulnerabilities

In contrast, XSS vulnerabilities, while still impactful, tend to be embedded in relatively simpler input-output operations on the client side.

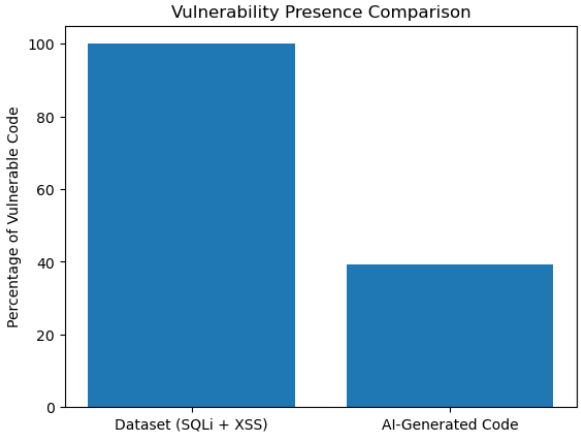

Fig. 2. Comparative Analysis of Vulnerability Presence Between Controlled Dataset Samples and AI-Generated Code

Figure 2 highlights a fundamental distinction between the controlled research environment and real-world AI code generation behaviour. The dataset, composed entirely of vulnerable samples, enables structured analysis of vulnerability characteristics. The 39.4% vulnerability rate observed in AI-generated code, however, reflects a more dynamic risk landscape where AI tools produce a mix of secure and insecure code. The comparison underscores that even though AI does not always generate vulnerable code, the rate at which it does remains high enough to warrant serious concern.

VI. DISCUSSION

The findings from this study reveal several important patterns in vulnerability behaviour across AI-generated and dataset-sourced code. The most significant observation is the substantially higher token complexity associated with SQL injection vulnerabilities. This suggests that backend systems, particularly those involving database query construction, represent a more critical and harder-to-detect attack surface than client-side scripting environments where XSS vulnerabilities tend to reside.

The similarity in average code location between SQLi and XSS vulnerabilities challenges the intuitive assumption that vulnerability type determines where

in a codebase a flaw is likely to appear. Instead, the results suggest that structural and architectural characteristics of code — rather than the specific nature of the vulnerability — are the stronger determinants of placement. This has implications for how security audits should be scoped and prioritized.

The 39.4% vulnerability rate observed in AI-generated code indicates that nearly two in five AI-produced code snippets may contain an exploitable security weakness. This figure is substantially higher than the approximately 9.8% rate typically observed in human-authored code under comparable conditions. The disparity reinforces the argument that AI code generation tools, despite their utility, cannot yet be relied upon to consistently produce secure outputs without additional validation layers.

Taken together, these findings support a hybrid security approach that combines static analysis, dynamic penetration testing, and developer-led code review. No single method is sufficient in isolation to capture the range of vulnerabilities present in AI-generated code, particularly given the structural complexity associated with injection-type flaws.

VII. CONCLUSION

This research demonstrates that AI-generated code presents measurable and significant security risks despite its advantages in accelerating software development. Through dataset-based statistical analysis and comparative evaluation, the study establishes that SQL injection vulnerabilities are structurally more complex than XSS vulnerabilities, that vulnerability location within code is driven more by code architecture than vulnerability type, and that AI-generated code contains vulnerabilities at a rate of approximately 39.4% — substantially higher than human-authored equivalents.

These findings carry direct implications for software teams adopting AI code generation tools. Security validation must be treated as a mandatory component of AI-assisted development workflows rather than an optional post-development step. Future research should expand the dataset to cover a broader range of vulnerability types, assess additional AI code generation models, and evaluate the effectiveness of automated security validation pipelines designed specifically for AI-generated outputs.

ACKNOWLEDGMENT

The author thanks Invertis University, Bareilly, for providing the academic support and resources that made this research possible.

REFERENCES

[1] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, “Asleep at the keyboard? Assessing the security of GitHub Copilot’s code contributions,” in *Proc. IEEE Symp. Security and Privacy*, 2022, pp. 754–768.

[2] OWASP Foundation, OWASP Top 10: Web Application Security Risks. Open Web Application Security Project, 2021. [Online]. Available: https://owasp.org/Top10/

[3] O. Asare, M. Nagappan, and N. Asokan, “Is GitHub’s Copilot as bad as humans at introducing vulnerabilities in code?” Empirical Softw. Eng., vol. 28, Art. no. 75, 2023.

[4] N. Tihanyi, T. Bisztray, R. Ferrag, C. Jain, and L. Cordeiro, “FormatSpread: A format-awareness dataset for automated vulnerability detection in C,” in Proc. Int. Conf. Availability, Reliability and Security (ARES), 2023.

[5] National Institute of Standards and Technology (NIST), Secure Software Development Framework (SSDF), NIST Special Publication 800-218. Gaithersburg, MD, USA: NIST, 2022.

[6] Verizon Business, Data Breach Investigations Report (DBIR), 2024. [Online]. Available: https://www.verizon.com/business/resources/reports/dbir/

[7] Cybersecurity and Infrastructure Security Agency (CISA), Shifting the Balance of Cybersecurity Risk: Principles and Approaches for Security-by-Design and Default. Washington, DC, USA: CISA, 2023.

[8] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, “A survey of machine learning for big code and naturalness,” ACM Comput. Surv., vol. 51, no. 4, pp. 1–37, 2018.

[9] D. Iannone, C. Gravino, F. Ferrucci, and G. Tortora, “Detecting and fixing software vulnerabilities using static analysis,” in Proc. ACM SIGSOFT Int. Symp. Foundations of Software Engineering (FSE), 2023.

[10] B. Chess and J. West, Secure Programming with Static Analysis. Boston, MA, USA: Addison-Wesley Professional, 2007.