Research Article *Open Access*

Journal of Pioneering Artificial Intelligence Research

ISSN: 3069-0846

DOI: *doi.org/10.63721/26JPAIR0129*

AI-Assisted Software Development as a New Supply-Chain Attack Surface

Hritesh Yadav[1*], Varun Singh[2], Kshitij Sharma[3], Suyash Karmarkar[4], Surabhi Agarwal[5]

[1]Cyber security and Artificial Intelligence, Independent Researcher, USA
[2]Cellular Telecom and Artificial Intelligence, Independent Researcher, USA
[3]Ecommerce and Artificial Intelligence, Independent Researcher, USA
[4]Infra Virtualization and artificial intelligence, Independent Researcher, USA
[5]Generative and Enterprise Artificial Intelligence Independent Researcher, Mountain View, USA

Citation: *Hritesh Yadav, Varun Singh, Kshitij Sharma, Suyash Karmarkar, Surabhi Agarwal (2026) AI-Assisted Software Development as a New Supply-Chain Attack Surface. J of Poin Artf Research 2(2), 1-8 WMJ-JPAIR-129*

Abstract

The rapid adoption of artificial intelligence (AI)–assisted software development tools has introduced a new and largely unexamined attack surface within the modern software supply chain. AI-powered coding assistants integrated into integrated development environments (IDEs) can analyze repository context and autonomously generate or modify source code across multiple files, thereby significantly accelerating software development productivity. However, this capability simultaneously introduces security risks that challenge traditional assumptions of trust, intent verification, and code provenance in software development workflows. In particular, AI systems may generate insecure code patterns, recommend malicious or nonexistent dependencies, or propagate vulnerabilities through contextual code generation mechanisms. This paper examines the emerging threat landscape associated with AI-driven code generation and identifies several novel attack vectors including indirect prompt manipulation, cross- file context poisoning, and automated dependency injection attacks. We propose a hybrid real-time security architecture that combines static code analysis with AI-based semantic inspection to detect malicious or anomalous code at the moment it is generated. The architecture integrates directly into developer IDE environments and continuously monitors AI-generated code suggestions, providing immediate feedback to developers and centralized alerts to security teams. The findings demonstrate that AI-assisted development environments require a zero-trust security model in which AI- generated code is treated as untrusted until verified. Our work highlights the need for new defensive mechanisms to secure the software supply chain in the era of AI-driven software engineering.

***Corresponding author**: Hritesh Yadav, Cyber security and Artificial Intelligence, Independent Researcher, USA.

Submitted: 16.03.2026 **Accepted**: 20.03.2026 **Published**: 30.03.2026

Keywords: AI-Assisted Software Development, Software Supply Chain Security, Large Language Models (Llms), Prompt Injection Attacks, Dependency Confusion, Slop Squatting, Ai Code Generation Security, Devsecops; Zero-Trust Software Development, Secure Coding Assistants

Introduction

Artificial intelligence–driven coding assistants such as GitHub Copilot, ChatGPT, and other large language model (LLM)–based development tools are rapidly transforming modern software engineering practices. These tools leverage large-scale machine learning models trained on massive code corpora to generate code completions, recommend libraries, and automate repetitive programming tasks. As a result, developers increasingly rely on AI systems as active collaborators during software development. Industry reports indicate that AI-generated code now constitutes a significant portion of newly written source code in enterprise development environments [1]. While these systems offer clear productivity advantages, their integration into the software development lifecycle introduces a new class of supply-chain security risks that have not been fully explored.

Traditionally, software supply-chain threats focused on compromised third-party libraries, malicious open-source packages, or manipulated build pipelines. However, AI coding assistants fundamentally alter this paradigm by introducing automated code generation directly into the earliest stages of software development. Because AI systems often operate with broad access to repository context and development environments, they effectively function as automated contributors to the codebase. Without proper verification mechanisms, AI-generated code may introduce vulnerabilities, insecure dependencies, or malicious logic that bypasses traditional security checkpoints. Recent studies evaluating AI coding assistants have demonstrated that a substantial fraction of generated code snippets contain security vulnerabilities, including improper input validation, insecure cryptographic practices, and hardcoded credentials [2].

Additionally, large language models have been shown to hallucinate software package names, which attackers can exploit by registering malicious packages under those names in public repositories a technique known as slop squatting [3]. These emerging risks illustrate how AI-assisted development introduces a new attack surface within the software supply chain. In this paper, we examine the security implications of AI-assisted development and present a threat model for adversarial exploitation of AI coding assistants. We analyze several emerging attack vectors and propose a defensive architecture designed to detect malicious AI-generated code in real time. Our contributions aim to provide a foundation for securing the next generation of AI-driven software development environments.

Background and Motivation

Software supply-chain security has become an increasingly critical concern as modern applications depend heavily on external components, open-source libraries, and complex development pipelines. High-profile incidents such as the SolarWinds supply-chain attack and malicious packages discovered in public repositories have demonstrated how attackers can compromise upstream components to impact thousands of downstream systems [4]. To mitigate such risks, the software industry has adopted several defensive mechanisms including Software Bills of Materials (SBOMs), dependency scanning tools, and integrity frameworks such as Supply-chain Levels for Software Artifacts (SLSA). SBOMs provide visibility into the components included in software systems, enabling organizations to track vulnerable dependencies and identify unauthorized libraries [5].

However, these mechanisms primarily focus on external components and build pipeline integrity rather than code generation processes occurring inside development environments. The integration of AI coding assistants fundamentally changes the threat landscape because AI systems can automatically generate code and introduce dependencies during the development phase itself. As a result, AI assistants effectively become automated participants in the software supply chain. Unlike human developers, these systems lack awareness of organizational security policies, internal dependency standards, or contextual threat models. Consequently, AI-generated code may inadvertently introduce insecure patterns, outdated libraries, or malicious dependencies into the codebase. Furthermore, developers often trust AI-generated suggestions due to perceived model expertise, which may reduce the rigor of code review processes. Empirical studies have shown that developers assisted by AI coding tools frequently overestimate the security of generated code and accept suggestions without thorough verification [6].

Another emerging challenge is dependency

proliferation caused by AI-generated solutions that introduce new libraries rather than leveraging existing project components. This behavior can significantly expand the attack surface by increasing the number of third-party dependencies and transitive packages included in software projects. In such environments, traditional supply-chain security mechanisms may be insufficient because vulnerabilities originate during code generation rather than through external dependency compromise. These challenges highlight the need for new security approaches that integrate verification mechanisms directly into AI-assisted development workflows.

Threat Model

In this work, we consider a software development environment in which AI coding assistants are deeply integrated into developer workflows through IDE plugins or automated code generation agents. These systems typically analyze repository context—including multiple source files, configuration files, and project documentation—to generate code suggestions that developers may accept and incorporate into the codebase. In many cases, AI assistants operate with privileges comparable to the developer's own environment, allowing them to recommend dependencies, modify existing files, or generate new functions across different modules. Under this model, the primary assets at risk include the integrity of the source code, the trustworthiness of dependencies introduced during development, and the overall security posture of the resulting software artifacts.

The adversary model assumes that attackers do not have direct commit access to the repository but instead attempt to exploit AI systems as indirect vectors for inserting malicious code. One potential adversary is an external attacker who manipulates the context consumed by the AI assistant by embedding malicious instructions within documentation, comments, or configuration files. These instructions may influence the AI model to generate insecure code or introduce malicious dependencies. Another adversarial scenario involves dependency manipulation attacks in which attackers publish malicious packages designed to be recommended by AI coding assistants. Because language models occasionally hallucinate package names, attackers can register packages matching those hallucinated names to compromise development environments when developers install them.

A third adversary model involves malicious insiders or compromised contributors who introduce subtle code modifications intended to manipulate the AI's contextual understanding of the project. Such modifications may influence subsequent AI-generated code to include vulnerabilities or insecure patterns. The ultimate goal of these adversaries is to inject malicious functionality or security vulnerabilities into the software supply chain without triggering traditional security defenses. Because AI-generated code may appear syntactically correct and functionally valid, such attacks may evade manual review processes and automated testing mechanisms. This threat model illustrates the necessity of treating AI-generated code as an untrusted input source within the development lifecycle.

Emerging Attack Vector

The integration of AI assistants into software development workflows enables several new attack vectors that extend beyond traditional software supply-chain threats. One of the most significant threats is prompt injection, where adversaries manipulate the inputs provided to large language models in order to influence generated outputs. Prompt injection attacks may occur either directly or indirectly. In direct prompt injection, attackers attempt to override system constraints by crafting prompts that instruct the model to bypass security restrictions or generate malicious code.

However, more concerning are indirect prompt injection attacks, where adversarial instructions are embedded within external data sources such as code comments, configuration files, documentation pages, or external resources that the AI assistant may ingest as contextual input. When the model processes these inputs, it may unknowingly interpret them as instructions and generate code that violates security policies. Recent research has demonstrated that prompt injection attacks can cause AI systems to expose sensitive data, generate insecure code patterns, or introduce malicious functionality into otherwise legitimate software systems [7]. Yadav et al. further categorize adversarial prompt injection techniques and demonstrate how attackers can exploit LLM behavior through carefully crafted input manipulation

strategies designed to bypass model alignment safeguards [8]. Another emerging attack vector is dependency manipulation through AI-generated package suggestions. Large language models occasionally hallucinate the names of software libraries when generating code examples. Attackers can exploit this behavior by registering malicious packages under these hallucinated names in public repositories such as npm or PyPI. When developers attempt to install these suggested dependencies, the malicious package becomes integrated into the development environment, enabling code execution or data exfiltration. This attack pattern, known as slop squatting, represents a new variation of traditional typo squatting attacks adapted specifically for AI-assisted development environments [3].

A third attack vector arises from cross-file context poisoning, where attackers manipulate the contextual inputs provided to AI coding assistants. Modern development assistants often analyze multiple files simultaneously to improve code suggestions. By introducing subtle, semantically benign modifications to one file, attackers can influence how the model generates code in another file. These modifications may alter naming conventions, coding patterns, or data flows in ways that encourage the AI to generate vulnerable logic such as SQL injection–prone queries or unsafe command execution patterns. Because these contextual changes often appear harmless to human reviewers, such attacks can remain undetected until the vulnerable code is deployed. Collectively, these attack vectors illustrate how AI coding assistants introduce new pathways for supply-chain compromise that bypass traditional dependency security mechanisms and highlight the need for defensive controls that monitor code generation in real time.

Related Work

The security implications of AI-assisted software development intersect with multiple areas of existing research, including software supply-chain security, static code analysis, and adversarial machine learning. Traditional supply-chain security frameworks primarily focus on protecting build pipelines, verifying dependencies, and ensuring artifact provenance. For example, the Software Bill of Materials (SBOM) framework provides visibility into the components included within software products and enables organizations to track vulnerable dependencies across development environments [5]. Similarly, the Supply-chain Levels for Software Artifacts (SLSA) framework aims to ensure the integrity of build pipelines by establishing verifiable build provenance and tamper-resistant artifact generation processes [9]. While these frameworks significantly improve visibility and traceability within the software supply chain, they do not directly address vulnerabilities introduced during code generation itself. In parallel, research into AI-generated code security has begun to reveal the potential risks associated with large language models trained on publicly available code repositories. Pearce et al. demonstrated that a substantial proportion of code generated by GitHub Copilot contains security vulnerabilities when evaluated across common programming tasks [2].

Their study showed that AI-generated code often reproduces insecure coding practices present within training datasets, highlighting the limitations of current models in understanding secure programming principles. Other studies have explored the concept of package hallucination in AI- generated code, demonstrating how language models may suggest nonexistent libraries that attackers can exploit through slop squatting techniques [3]. Additionally, recent research has examined adversarial prompt injection attacks targeting large language models. Yadav et al. present a taxonomy of prompt injection strategies that categorize adversarial techniques based on input manipulation, jailbreak prompts, and context poisoning attacks, and propose mitigation frameworks that include input sanitization, output validation, and model alignment strategies [8]. Despite these advancements, most existing defensive approaches operate after code has already been generated and committed to the repository. Static analysis tools and software composition analysis platforms typically run during code review or continuous integration phases, meaning vulnerabilities introduced by AI assistants may persist within the development environment for extended periods before detection. This delay underscores the importance of integrating security verification mechanisms directly into AI- assisted development workflows, enabling vulnerabilities to be detected at the moment of code generation.

Proposed Architecture

To address the emerging security risks introduced by

AI- assisted development environments, we propose a secure architecture designed to monitor and validate AI-generated code in real time. The architecture is based on the principle that AI-generated code should be treated as an untrusted input until it has been verified through automated security checks and contextual validation mechanisms. Unlike traditional software security tools that operate during later stages of the development lifecycle such as code review or continuous integration pipelines, the proposed architecture integrates security analysis directly into the development workflow. This approach enables vulnerabilities introduced by AI coding assistants to be detected immediately at the moment of generation rather than after the code has already been committed to the repository. The architecture consists of three primary components: a hybrid scanning engine that combines static and AI-driven analysis techniques, an IDE-integrated monitoring layer that intercepts AI-generated suggestions, and a centralized security intelligence platform that aggregates alerts and continuously improves detection models.

The hybrid scanning engine forms the core of the architecture and is responsible for analyzing all AI-generated code suggestions before they are accepted into the codebase. This component incorporates traditional static analysis techniques that detect known vulnerability patterns using rule- based detection mechanisms derived from secure coding standards and vulnerability taxonomies such as the Common Weakness Enumeration (CWE). Static analysis is particularly effective at identifying well-known vulnerabilities such as insecure cryptographic implementations, unsafe deserialization, command injection patterns, and improper input validation. However, static rules alone are insufficient for detecting the more subtle attack vectors associated with AI- generated code, particularly those resulting from prompt injection or contextual poisoning attacks. To address this limitation, the architecture augments rule-based analysis with an AI-driven semantic inspection module. This component leverages machine learning techniques to analyze the behavior and intent of generated code rather than relying solely on syntactic signatures. By examining contextual relationships between functions, variables, and external dependencies, the semantic inspection module can identify anomalous patterns that may indicate malicious behavior or insecure coding practices.

The architecture is implemented through an IDE-integrated plugin that operates transparently within the developer's environment. When an AI coding assistant generates a code suggestion, the plugin intercepts the proposed code and extracts the relevant code diff along with surrounding contextual information. This diff is then forwarded to the hybrid scanning engine for analysis. The use of diff-based analysis significantly improves performance by focusing security checks only on newly generated or modified code rather than scanning the entire repository. If the scanning engine detects suspicious patterns or insecure dependencies, the system immediately notifies the developer through inline warnings displayed within the IDE interface. These warnings provide explanations of the detected security risks and may recommend safer alternatives or mitigation strategies. In cases where high-risk vulnerabilities are detected, such as the introduction of unknown external dependencies or execution of system commands using unsensitized input, the system may temporarily block the suggestion from being accepted until the developer resolves the issue or explicitly overrides the warning.

In addition to providing immediate feedback to developers, the proposed architecture includes a centralized monitoring platform that aggregates security alerts generated across development environments. This platform allows security teams to monitor trends in AI-generated vulnerabilities and identify potential adversarial campaigns targeting development workflows. For example, if multiple developers encounter suggestions involving the same unknown dependency, the centralized platform can flag the dependency as potentially malicious and distribute updated detection rules across all development environments. This capability enables rapid response to emerging threats such as slop squatting attacks or coordinated prompt injection campaigns. The centralized platform also supports continuous improvement of the semantic inspection model by incorporating feedback from developers and security analysts. Over time, this feedback loop enables the system to refine detection accuracy and reduce false positives while maintaining strong protection against emerging attack vectors. By combining real-time monitoring, hybrid security analysis, and centralized threat intelligence, the proposed architecture provides

a comprehensive defense framework for securing AI-assisted software development environments.

Evaluation Consideration

Evaluating the effectiveness of security mechanisms designed for AI-assisted development environments requires careful consideration of both security performance and developer usability. One of the primary evaluation metrics for the proposed architecture is detection coverage, which measures the system's ability to identify vulnerabilities introduced through AI-generated code suggestions. Detection coverage can be evaluated by simulating various attack scenarios within controlled development environments, including prompt injection attacks, dependency manipulation attacks, and context poisoning attacks. In these experiments, AI coding assistants are intentionally exposed to adversarial inputs designed to generate insecure code patterns. The effectiveness of the hybrid scanning engine can then be measured based on its ability to detect these vulnerabilities before the code is accepted into the repository. Prior studies have demonstrated that a significant proportion of AI- generated code contains security vulnerabilities, suggesting that real-time security verification mechanisms could significantly reduce the introduction of insecure code into production systems [2].

Another important evaluation metric is the false positive rate, which reflects the frequency with which legitimate code is incorrectly flagged as insecure by the scanning engine. Excessive false positives can negatively impact developer productivity by interrupting workflows and reducing confidence in the security tool. Therefore, the semantic analysis component of the architecture must be carefully tuned to balance detection sensitivity with practical usability. Machine learning–based analysis models may require continuous retraining using labeled datasets that include both vulnerable and secure code examples. Incorporating feedback from developers can further improve accuracy by identifying scenarios in which alerts should be suppressed or reclassified. Maintaining an acceptable false positive rate is particularly important in interactive development environments where security alerts must be delivered in real time without creating unnecessary friction.

Performance overhead is another critical factor when evaluating real-time security mechanisms integrated into development tools. Because developers interact with AI assistants frequently during coding sessions, the scanning process must complete within milliseconds in order to avoid disrupting productivity. Experimental evaluation of the proposed architecture should measure the time required to analyze code suggestions of varying sizes and complexity. Diff-based analysis provides a significant performance advantage by limiting the scope of analysis to newly generated code rather than the entire project. Additionally, the architecture may employ asynchronous scanning strategies in which static analysis results are delivered immediately while more computationally intensive semantic analysis occurs in parallel. This approach ensures that developers receive rapid feedback while still benefiting from deeper security analysis.

Finally, evaluation should consider the broader impact of the architecture on software development practices. Longitudinal studies may measure the number of vulnerabilities detected during early development stages compared with traditional security pipelines. If vulnerabilities are detected earlier in the development lifecycle, organizations can significantly reduce remediation costs and improve overall security posture. User studies may also evaluate how developers respond to security warnings generated by the system and whether the architecture encourages more secure coding practices when interacting with AI assistants. By analyzing these factors collectively, researchers can determine whether the proposed architecture effectively mitigates the risks associated with AI-assisted development while maintaining a positive developer experience.

Discussion

The integration of AI coding assistants into software development workflows represents a significant shift in how software systems are created, reviewed, and maintained. While these tools offer substantial productivity benefits, they also introduce new challenges related to trust, accountability, and security verification. Traditional development practices assume that code written by developers can be trusted once it has passed through peer review and automated testing processes. However, AI-generated code introduces an additional layer of complexity because the logic produced by large language models may originate from training data rather than deliberate

human reasoning. As a result, organizations must reconsider how trust boundaries are defined within the software development lifecycle. One potential approach involves adopting zero-trust principles for AI-generated code, where all suggestions produced by AI assistants are treated as untrusted inputs until they have been verified through automated security checks and developer review.

Another important aspect of securing AI-assisted development environments involves improving transparency and traceability of AI-generated code. Development tools may need to incorporate mechanisms for tracking the provenance of code suggestions generated by AI assistants. Such mechanisms could record metadata describing which model generated a particular code segment, the prompt used to generate the code, and the contextual inputs provided to the model. This information could prove valuable for auditing purposes and incident response investigations. For example, if a vulnerability is discovered in a deployed system, security analysts could trace the origin of the vulnerable code and determine whether it was generated by an AI assistant or written manually by a developer. Provenance tracking may also help identify patterns in adversarial attacks targeting AI-assisted development environments.

Future research should also explore the potential for automated verification techniques to validate AI-generated code. Formal verification methods, while traditionally applied to safety-critical systems, may be adapted to verify certain properties of AI-generated code segments. For example, automated reasoning tools could be used to ensure that generated functions satisfy specific security constraints or adhere to defined input validation requirements. Another promising direction involves multi-model validation frameworks in which multiple AI systems independently analyze generated code to identify potential vulnerabilities. Such approaches could reduce the likelihood that vulnerabilities introduced by one model remain undetected. Additionally, improving model alignment during training may help reduce the frequency with which AI assistants generate insecure code patterns. However, even with improved model training and alignment techniques, it is unlikely that AI assistants will completely eliminate security risks associated with automated code generation. Consequently, organizations must continue to develop complementary security mechanisms that monitor and validate AI-generated outputs throughout the development lifecycle.

Conclusion

AI-assisted software development represents a transformative advancement in modern software engineering, enabling developers to write code more efficiently through automated code generation and intelligent development assistance. However, the integration of AI coding assistants into development environments introduces a new category of supply-chain security risks that extend beyond traditional dependency management and build pipeline protections. This paper examined several emerging attack vectors associated with AI-assisted development, including prompt injection attacks, dependency manipulation through slopsquatting, and cross-file context poisoning attacks. These threats demonstrate how adversaries can exploit the behavior of large language models to introduce vulnerabilities or malicious code into the software supply chain. To address these challenges, we proposed a hybrid security architecture that integrates static analysis and AI-based semantic inspection directly into developer workflows through an IDE-integrated monitoring system. By performing real-time analysis of AI-generated code suggestions, the proposed architecture can detect potentially malicious behavior before code enters the repository. This approach enables organizations to adopt a zero-trust model for AI-generated code while preserving the productivity advantages offered by AI-assisted development tools. As generative AI continues to reshape the software development landscape, securing AI-driven development environments will become an increasingly critical component of software supply-chain security. Future research should focus on improving automated verification techniques, strengthening model alignment mechanisms, and developing industry standards for secure AI-assisted development practices [10-15].

References

1. Djalali H, Aljedaani W, Ludi S (2025) The Evolution of Software Usability in Developer Communities: An Empirical Study on Stack Overflow. Software 4: 27.
2. H Pearce, B Ahmad, B Tan, B Dolan-Gavitt and R Karri (2022) Asleep at the Keyboard?

Assessing the Security of GitHub Copilot's Code Contributions," in Proc. IEEE Symp. Security and Privacy Workshops 1-9.

3. J Spracklen, R Wijewickrama, AHMN Sakib, A Maiti, et al. (2024) "We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs," arXiv preprint arXiv:2406.10279.
4. P Ladisa, H Plate, M Martinez and O Barais (2023) "Sok: Taxonomy of Attacks on Open Source Software Supply Chains," in Proc. IEEE European Symp. Security and Privacy 150-166.
5. O'Donoghue EJ (2024) Using software bill of materials for software supply chain security and its generation impact on vulnerability detection https://www.cs.montana.edu/izurieta/thesis/ODonoghue.pdf.
6. Kudriavtseva A, Hotak NA, Gadyatskaya O (2025) My Code Is Less Secure with Gen AI: Surveying Developers' Perceptions of the Impact of Code Generation Tools on Security https://www.scribd.com/document/1003182810/3-3672608-3707778.
7. OWASP OT (10) for Large Language Model Applications, OWASP Foundation. URL: https://owasp. org/www-project-top-10-for-largelanguage-model-applications.
8. Yadav H, Singh V, Sharma K (2025) Adversial Prompt Injection in Large Language Models: Taxonomy, Exploits, and Mitigation Frameworks. In 2025 Seventh International Conference on Research in Computational Intelligence and Communication Networks 244-251.
9. Williams L, Benedetti G, Hamer S, Paramitha R, Rahman I, et al. (2025) Research directions in software supply chain security. ACM Transactions on Software Engineering and Methodology 34: 1-38.
10. S Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, et al. (2023) "Sparks of Artificial General Intelligence: Early Experiments with GPT-4," arXiv preprint arXiv:2303.12712.
11. Achiam J, Adler S, Agarwal S, Ahmad L, Akkaya I, et al. (2023) Gpt-4 technical report. arXiv preprint arXiv:2303.08774.
12. N Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, et al. (2021) "Extracting Training Data from Large Language Models," in Proc. USENIX Security Symposium 2633-2650.
13. A Gupta, S Jana, P Srivastava (2024) "Prompt Injection Attacks and Defenses in LLM-Integrated Systems," in Proc. ACM Conf. Computer and Communications Security Workshops https://www.semanticscholar.org/paper/Prompt-Injection-Attacks-and-Defenses-in-Liu-Jia/04fa3ebc4c0c0b4a2d2d1a3fc612134a05057696.
14. A Birsan (2021) "Dependency Confusion: How I Hacked into Apple, Microsoft, and Dozens of Other Companies," Medium Security Blog https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610.
15. M Howard, D LeBlanc (2003) Writing Secure Code, 2nd ed. Redmond, WA, USA: Microsoft Press https://www.microsoftpressstore.com/store/writing-secure-code-9780735617223.