Secure coding for web applications: Frameworks, challenges, and the role of LLMs

Kiana Kiashemshaki[1], Mohammad Jalili Torkamani[2], Negin Mahmoudi[3]
[1]Department of Computer Science, Bowling Green State University, Bowling Green, OH, USA
[2]School of Computing, University of Nebraska-Lincoln, Lincoln, Nebraska, USA
[3] Department of Civil Environmental and Ocean Engineering, Stevens Institute of Technology, New Jersey , USA
Emails: kkiana@bgsu.edu, mJaliliTorkamani2@huskers.unl.edu, nmahmoud1@stevens.edu

***Abstract*—Secure coding is a critical yet often overlooked practice in software development. Despite extensive awareness efforts, real-world adoption remains inconsistent due to organizational, educational, and technical barriers. This paper provides a comprehensive review of secure coding practices across major frameworks and domains, including web development, DevSecOps, and cloud security. It introduces a structured framework comparison and categorizes threats aligned with the OWASP Top 10. Additionally, we explore the rising role of Large Language Models (LLMs) in evaluating and recommending secure code, presenting a reproducible case study across four major vulnerability types. This paper offers practical insights for researchers, developers, and educators on integrating secure coding into real-world development processes.**

***Index Terms*—Secure Coding, Web Security, OWASP, NIST, SSDLC, SQL Injection, XSS, Authentication Security, LLM**

I. Introduction

The widespread adoption of web applications has dramatically reshaped digital service delivery across industries. From finance and e-commerce to healthcare and education, organizations increasingly rely on web based systems to streamline operations, deliver services, and engage with users. However, this growing dependency has made web applications one of the most targeted vectors for cyberattacks, exposing sensitive data and critical infrastructure to significant risk. As a result, embedding robust security mechanisms directly into the software development lifecycle has become an essential component of modern software engineering [2], [6].

In recent years, the frequency, scale, and sophistication of web based attacks have surged. Common vulnerabilities such as SQL injection (SQLi), cross-site scripting (XSS), broken authentication, and cross-site request forgery (CSRF) continue to dominate global threat landscapes [9], [12], [14]. These flaws are often the result of insecure coding practices, weak input validation, and a lack of secure software design principles. Empirical studies have shown that a large percentage of data breaches can be traced back to such preventable vulnerabilities [14].

To mitigate these risks, several secure coding frameworks and best practice guidelines have been proposed. The OWASP Top 10 identifies the most critical security risks for web applications and provides actionable strategies to address them [6]. The NIST Cybersecurity Framework offers a structured, high level approach to managing cybersecurity risk through its five core functions: identify, protect, detect, respond, and recover [7]. Similarly, the Secure Software Development Lifecycle (SSDLC) emphasizes the integration of security controls at every stage of the development process, from requirements gathering to deployment and maintenance [2], [4].

Despite the availability of such resources, the consistent implementation of secure coding practices remains a challenge in the real world. Many organizations face barriers such as inadequate developer training, pressure to meet tight deadlines, and limited security resources. This often results in a prioritization of feature delivery over security assurance, leaving applications vulnerable to exploitation [1], [3], [5].

Furthermore, the rapid evolution of software development paradigms—such as agile methodologies, DevOps pipelines, and cloud native environments has introduced new complexities that traditional security practices struggle to keep up with. These shifts necessitate a more adaptive, automated, and integrated approach to secure coding, aligning security measures with modern development workflows.

This paper presents a comprehensive review of secure coding frameworks and practices in the context of web application security. It explores established standards, analyzes key implementation challenges, and investigates emerging trends such as AI powered security tools, zero trust architectures, and security as code in DevSecOps environments. By synthesizing insights from academic literature, industry reports, and real-world practices, the paper aims to offer practical guidance for developers, security engineers, and organizational leaders striving to enhance software security.

As illustrated in Figure 1, injection flaws and XSS vulnerabilities continue to be among the most commonly reported security issues in modern web applications.

II. Literature Review

Secure coding has long been recognized as a critical discipline in the field of cybersecurity. Numerous studies have examined the causes of software vulnerabilities and proposed various frameworks, strategies, and methodologies to prevent them. This section presents a structured overview of the

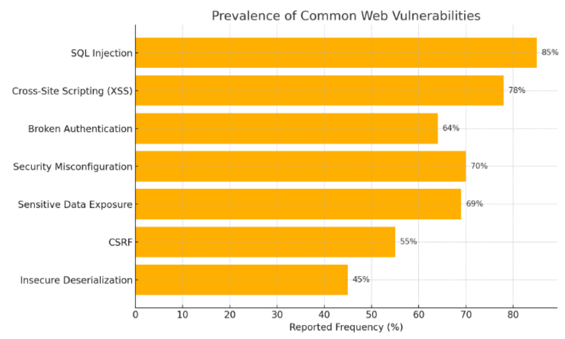

Fig. 1. Prevalence of Common Web Vulnerabilities (adapted from OWASP Top 10 and industry surveys).

key literature, organized around core contributions: proactive security integration, vulnerability identification and mitigation, automation, and emerging security technologies.

A. *Proactive Integration of Security in the Development Lifecycle*

Early efforts in secure software design emphasized the importance of integrating security from the initial stages of development. Shostack [17] introduced threat modeling as a proactive methodology, urging developers to assess risks during the design phase rather than waiting until deployment. His approach underscored that identifying and mitigating threats early reduces the cost and complexity of security interventions.

Similarly, McGraw [2] advocated for building security into the software lifecycle by adopting risk based security assessments, secure architecture principles, and developer education. He positioned security not as an add on but as a foundational element in software design. Lipner [18] demonstrated the practical implementation of these ideas through Microsoft's Security Development Lifecycle (SDL), showing that introducing security checkpoints throughout development can reduce vulnerabilities and enhance software reliability.

B. *Vulnerability Patterns and Defensive Coding Practices*

A Complementary body of research focused on understanding how specific coding flaws lead to exploitation. Howard and LeBlanc [1] analyzed real-world case studies of security failures and proposed coding strategies to avoid common mistakes. They emphasized consistency in applying secure coding principles across teams and projects.

Viega and McGraw [4] further highlighted typical software vulnerabilities such as poor input validation, weak error handling, and insecure authentication. They promoted defensive programming, encouraging developers to assume that any input or dependency might be malicious and to write code accordingly.

Whittaker and Thompson [19] contributed by mapping attack patterns to software defects. Their work offered insight into how attackers exploit insecure code and how developers can anticipate such attacks during development. This mindset shift from reactive defense to attacker aware development laid the groundwork for more resilient applications.

C. *Security Testing and Automation*

Another significant area of research examines tools and techniques for identifying security flaws during development and testing. Jones and Rastogi [20] compared static and dynamic analysis tools, showing that both approaches have strengths and limitations. Static analysis is valuable for early code level checks, while dynamic analysis is critical for detecting runtime vulnerabilities. The authors recommended a hybrid approach to maximize security coverage.

Sharma et al. [21] expanded on this by exploring AI powered vulnerability scanners that automate threat detection. Their study found that machine learning based tools improve scanning speed and efficiency, especially in large codebases. However, they also noted the risk of false positives, reinforcing the need for expert validation.

D. *Evaluating Frameworks and Emerging Technologies*

Rajput et al. [22] evaluated several secure coding frameworks, including OWASP and NIST, across diverse software development environments. Their findings revealed that while these frameworks offer robust guidelines, adoption is often hindered by limited expertise, resource constraints, and organizational resistance.

To address security challenges in decentralized and distributed systems, Ghobadi and Tavana [23] proposed blockchain based authentication mechanisms. They argued that decentralized identity management can reduce the risks of credential theft and unauthorized access in web applications.

E. *Summary of Literature Insights*

Taken together, the literature shows that secure coding is a multi dimensional field that evolves in response to both technical and organizational factors. Effective approaches combine early integration of security practices [2], [17], use of structured frameworks [6], [7], [22], automation tools [21], and support for emerging technologies such as blockchain [23]. While there is no one size fits all solution, the convergence of best practices from software engineering, cybersecurity, and AI research continues to shape the future of secure development.

To illustrate how secure coding concepts and frameworks have evolved over time, Figure 2 presents a timeline of key milestones. These include foundational works such as McGraw's secure development lifecycle, Shostack's introduction of threat modeling, and the iterative updates to the OWASP Top 10, as well as the formalization of NIST's cybersecurity framework and more recent AI driven developments.

Secure coding remains central to preventing critical threats like SQL injection, XSS, and broken authentication [2], [4], [18]. However, the literature also makes it clear that technical strategies alone are insufficient developer training, organizational buy in, and continuous adaptation to new risks are equally vital for lasting impact.

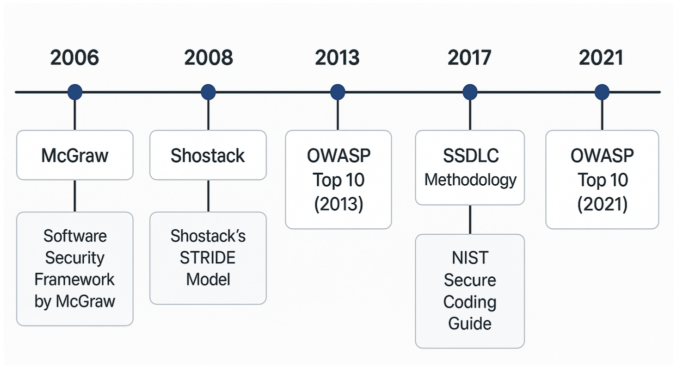

Fig. 2. Timeline of Secure Coding Frameworks and Milestones.

III. REMARKS ON LITERATURE REVIEW

The literature on secure coding offers a wealth of insights into foundational practices, tools, and frameworks. However, several key challenges and research gaps emerge that require further investigation to bridge the divide between theory and practice. This section highlights three major observations derived from the reviewed works: challenges in implementation, gaps in research coverage, and limitations of automated security testing.

A. Challenges in the Practical Adoption of Secure Coding

Despite the presence of structured guidelines and mature frameworks, the consistent and effective implementation of secure coding practices remains elusive in many real-world settings. Rajput et al. [22] identified several obstacles that hinder adoption, including insufficient developer training, lack of management commitment, and competing business priorities. These organizational factors often result in security being deprioritized in favor of faster delivery timelines.

Whittaker and Thompson [19] further emphasized that many security failures are not purely technical but arise from developers' limited understanding of attacker behaviors and threat vectors. Without adequate training and attacker centric thinking, even well intentioned teams may overlook exploitable flaws, leading to persistent vulnerabilities in deployed software.

B. Research Gaps in Secure Coding Education and Integration

While numerous studies propose technical solutions for improving code security, far fewer address the systemic barriers to their adoption particularly in educational and organizational contexts. Ghobadi and Tavana [23] observed that secure coding principles are not consistently embedded in software engineering curricula, leaving new graduates unprepared to handle security responsibilities in practice.

Moreover, Shostack [17] highlighted the lack of alignment between threat modeling approaches and modern agile or DevOps workflows. These fast paced development environments often omit early stage security planning due to time constraints and tooling complexity, which limits the effectiveness of traditional models like SDL and SSDLC.

C. Limitations of Automated Testing and AI based Security Tools

With the increasing complexity of software systems, security testing has evolved to incorporate automation and artificial intelligence. Sharma et al. [21] demonstrated that AI powered static and dynamic analysis tools can identify common vulnerabilities more quickly and comprehensively than manual methods, making them valuable components of modern development pipelines.

However, these tools are not without limitations. Automated scanners frequently generate false positives, lack contextual understanding, and may struggle with complex, logic based vulnerabilities. As a result, expert oversight is still essential to validate findings and guide remediation efforts. Over reliance on automation without human review can lead to misplaced confidence and overlooked risks.

D. Summary of Literature Remarks

Taken together, these observations highlight a clear disconnect between secure coding theory and its operationalization in practice. Technical tools and frameworks exist, but their real-world impact is often constrained by educational, cultural, and organizational factors. The literature suggests that future research should not only focus on advancing technical solutions but also prioritize improved training, better integration with agile methodologies, and hybrid approaches that balance automation with human expertise.

IV. COMPARISON OF SECURE CODING FRAMEWORKS

Several well established frameworks have emerged to guide secure coding efforts and improve the security posture of software systems. This section compares three of the most influential models: the OWASP Top 10, the NIST Cybersecurity Framework (CSF), and the Secure Software Development Lifecycle (SSDLC). Each serves a distinct role in promoting secure development, with unique strengths and limitations.

A. OWASP Top 10 and Secure Coding

The OWASP Top 10 is a widely recognized industry benchmark that outlines the most critical security risks to web applications [6]. It covers issues such as injection attacks, broken access control, and misconfigurations. The list is updated regularly most recently in 2021 to reflect emerging threats and trends, and it is mapped to Common Weakness Enumerations (CWEs), providing actionable guidance for developers.

Howard and LeBlanc emphasize that the OWASP Top 10 is particularly effective in encouraging secure input handling and validation, which helps mitigate frequent attack vectors [1]. Similarly, McGraw supports OWASP's emphasis on incorporating security earlier in the development process, reinforcing its role as a proactive risk reduction strategy [2].

One key strength of the OWASP Top 10 lies in its simplicity and accessibility it translates complex security concerns into ten high impact categories that are easily understandable even for non security professionals. However, its limitations are equally noteworthy. The Top 10 is focused exclusively on web application risks and lacks depth in areas such as infrastructure security or CMS specific threats [29]. Additionally, its high level nature may leave organizations without sufficient guidance for implementation.

B. NIST Cybersecurity Framework

The NIST CSF offers a broad, adaptable structure for managing cybersecurity risk across organizational and technical domains [7]. It defines six core functions govern, identify, protect, detect, respond, and recover that serve as a comprehensive roadmap for integrating security into both strategic planning and operational workflows. Seacord highlights that adopting this framework improves alignment between development and security teams while promoting robust software engineering practices [3].

The flexibility of the NIST framework is one of its greatest strengths. It can be customized to fit an organization's size, industry, and maturity level, making it suitable for both small businesses and large enterprises [30]. Moreover, its adoption supports regulatory compliance and improves stakeholder confidence.

However, the framework's high level of abstraction may pose challenges. It does not prescribe specific technical controls, which can hinder implementation in organizations lacking in house cybersecurity expertise [31]. This often leads to inconsistent adoption unless supported by strong governance and training programs.

Figure 3 provides a visual overview of the NIST CSF's core functions.

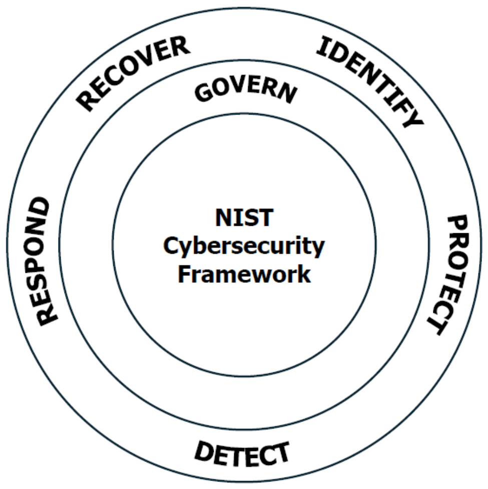

Fig. 3. NIST Cybersecurity Framework.

C. Secure Software Development Lifecycle (SSDLC)

The SSDLC emphasizes integrating security activities throughout all phases of software development from requirements gathering and design to implementation, testing, and maintenance. By embedding security into each phase, the SSDLC model promotes early identification and mitigation of vulnerabilities, leading to more secure and reliable software [4].

Viega and McGraw [4] and Long et al. [5] stress that this early integration significantly reduces downstream security costs and improves code quality. The SSDLC is particularly beneficial in reducing overlooked edge cases, ensuring more thorough validation and testing, and building a security conscious development culture [32].

Despite its strengths, SSDLC introduces additional complexity and demands greater coordination across teams. In agile and fast paced development environments, integrating security in every iteration can be seen as time consuming, which sometimes results in resistance from developers or project managers.

Figure 4 illustrates how security tasks are embedded within each stage of the SSDLC process.

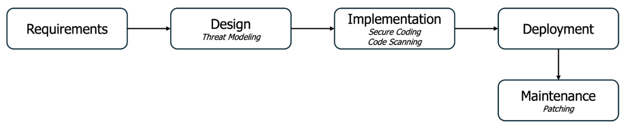

Fig. 4. Secure Software Development Life Cycle (SSDLC).

D. Comparative Analysis

Each of the three frameworks plays a vital role in enhancing software security, but they differ in scope, focus, and implementation difficulty. Table I summarizes their key characteristics, offering a side by side comparison of their purposes, advantages, and practical limitations.

V. CHALLENGES IN IMPLEMENTING SECURE CODING PRACTICES

Although secure coding frameworks and tools are increasingly available, organizations still face numerous barriers to consistent adoption. These challenges span organizational constraints, educational gaps, and technical limitations. This section categorizes the core difficulties into three key dimensions.

A. Organizational Challenges

1) Resource and Budget Constraints: Small to mid sized enterprises often operate under limited budgets and with minimal personnel, making it difficult to prioritize secure coding practices. These constraints hinder investment in advanced security tools, code review processes, or developer training [9]. Howard and LeBlanc [1] note that organizations frequently underestimate the long term costs of insecure development, failing to allocate adequate resources for prevention strategies.

2) Limited Management Support: The successful implementation of secure development policies often requires buy in from senior leadership. However, decision makers frequently prioritize speed, feature delivery, and cost efficiency over long term security [24]. This strategic misalignment causes security to be treated as a secondary concern, leading developers to deprioritize security in their day to day workflows.

TABLE I
COMPARISON OF SECURE CODING FRAMEWORKS.

Framework	Purpose, Strengths, and Limitations
OWASP Top 10	**Purpose:** Identify and prioritize top web application vulnerabilities. **Strengths:** Easy to understand, regularly updated, widely adopted in training. **Limitations:** High level; limited guidance for implementation and infrastructure threats.
NIST CSF	**Purpose:** Provide a flexible framework for managing cybersecurity risk. **Strengths:** Broad applicability; supports governance, compliance, and scalability. **Limitations:** Too abstract for direct use by developers; requires security expertise.
SSDLC	**Purpose:** Integrate security into each stage of software development. **Strengths:** Proactive, improves code robustness, identifies flaws early. **Limitations:** Resource intensive; may slow development and be resisted in agile environments.

3) "Speed to Market" Culture: In highly competitive markets, the pressure to rapidly release software often supersedes the focus on building secure systems [17]. This urgency fosters a reactive rather than proactive approach to security, where controls are added post deployment if at all. Such practices increase the likelihood of releasing software with untested vulnerabilities [6], [17].

B. Educational and Knowledge Gaps

1) Inadequate Formal Training: A major barrier to secure software development is the lack of formal education in security principles among developers. Many computer science and software engineering programs continue to overlook secure coding as a core component of the curriculum [3], [10]. As McGraw argues, developers without foundational knowledge in secure development may inadvertently introduce exploitable flaws [2].

2) Lack of Developer Awareness: Even when resources are available, many developers remain unaware of best practices or credible training programs. The Open Source Security Foundation (OpenSSF) reports that 53% of developers have never received training in secure coding practices [25]. This is especially common among self taught developers, who may not have been exposed to security standards in a structured way.

3) Limited On the Job Training: Beyond initial education, there is a lack of continuous professional development in application security. Many organizations do not offer routine security training or allocate time for developers to stay informed about evolving threats. According to recent surveys, over half of developers attribute their inability to build secure applications to insufficient in house training programs.

C. Technical Challenges

1) Legacy Systems and Technical Debt: Large organizations often maintain legacy systems with outdated architectures and codebases that were not designed with modern security in mind [26]. Modifying such systems to align with current best practices is resource intensive and risky, especially when existing functionality is poorly documented. This technical debt discourages developers from making necessary security updates, leaving the systems exposed.

2) Inadequate Testing and Scanning: Comprehensive security testing is frequently overlooked under time constraints. Dynamic analysis, penetration testing, and manual code reviews are time consuming and are often omitted to meet delivery deadlines. In a 2024 survey by Contrast Security, 53% of organizations admitted to skipping security scans to expedite releases [27].

3) Architectural Complexity: Modern applications are composed of interconnected microservices, third party APIs, cloud native resources, and open source dependencies. This heterogeneous architecture significantly increases the attack surface and makes it difficult to enforce consistent security practices across all components [26]. Managing security across such diverse and distributed environments requires advanced tooling, centralized policies, and cross team coordination all of which are often lacking.

D. Summary of Challenges

In summary, the challenges in implementing secure coding are multifaceted. Organizational inertia, gaps in developer knowledge, and the inherent complexity of modern software systems all contribute to inconsistent adoption of best practices. Addressing these barriers requires a holistic strategy that combines education, tooling, process redesign, and executive level commitment.

To better visualize the prevalence and grouping of implementation barriers, Figure 5 presents a summary of root causes across organizational, educational, and technical domains.

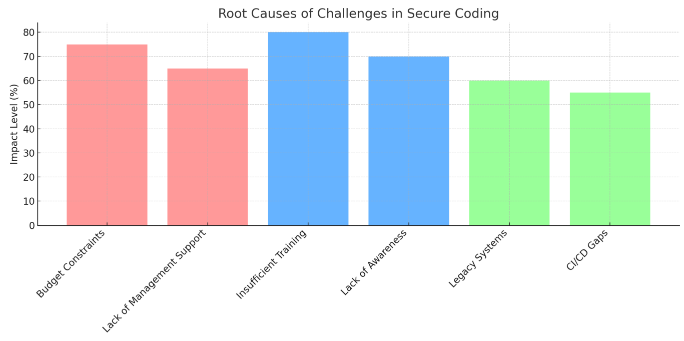

Fig. 5. Root Causes of Challenges in Secure Coding, grouped by Organizational, Educational, and Technical categories.

VI. Emerging Trends in Secure Coding

As cyber threats grow more sophisticated and development environments become increasingly agile and distributed, new trends in secure coding are emerging to bridge the gap between traditional practices and modern software demands. This section explores three of the most influential developments: AI driven security automation, the zero trust security model, and the rise of Security as Code in DevSecOps.

A. AI Driven Security Automation

Artificial Intelligence (AI) is playing an increasingly vital role in strengthening software security by enabling automation throughout the development lifecycle. AI powered tools can scan codebases for vulnerabilities, detect anomalous patterns, and even suggest or implement patches in near real time [12]. By automating routine security checks, AI reduces the manual burden on developers and enhances the speed and scale of vulnerability discovery.

Sutton et al. [15] note that AI significantly improves the efficiency of testing workflows and broadens coverage, making it especially valuable in large, complex systems. However, McGraw [2] cautions against over reliance on automated tools. AI systems may generate false positives or misinterpret context sensitive logic, necessitating human review to ensure accuracy and prevent the overlooking of subtle vulnerabilities.

Although promising, AI based security is still maturing. Successful adoption requires continuous validation, human oversight of high risk decisions, and regular model updates to align with evolving threat landscapes.

B. Zero-Trust Security Model

The zero-trust model operates on the principle that no user or system whether inside or outside an organization's perimeter should be automatically trusted. Every access request must be authenticated, authorized, and verified before granting permission [7]. This philosophy is well suited to the modern development landscape, where applications operate in distributed, cloud native environments that manage sensitive user data.

Shostack [17] emphasizes that combining secure coding with zero-trust principles such as role based access control, network segmentation, and continuous verification can significantly limit an attacker's lateral movement post compromise. As a result, zero-trust architectures create a more resilient and tamper resistant system.

Despite its advantages, implementing zero-trust introduces operational complexity. Continuous validation mechanisms, like expiring tokens and strict authentication policies, can impact application flow and performance [33]. Moreover, effective adoption requires skilled personnel and well integrated tooling, which may be lacking in many development teams.

C. Security-as-Code (SaC) in DevSecOps

Security-as-Code (SaC) refers to embedding security policies, validation rules, and configurations directly into source code and infrastructure-as-code artifacts. This practice is a cornerstone of DevSecOps, where development, operations, and security teams collaborate to integrate security throughout the continuous integration and deployment (CI/CD) pipeline [4], [5].

By codifying security controls, organizations can enforce policies consistently during every build, test, and deployment cycle. As discussed in our previous work [44], remote agile tools significantly influence communication workflows, which are tightly coupled with the adoption and integration of secure development practices. SaC enables real time enforcement of access policies, vulnerability scans, and compliance checks, helping to catch and remediate issues early in the lifecycle [34].

Howard and LeBlanc [1] highlight that early integration of security reduces the cost of post release fixes and fosters a culture of secure development. SaC implementations often include deployment restrictions, role based access control definitions, and automated security testing frameworks [34].

D. Summary of Trends

Together, these emerging trends signal a shift from reactive security practices toward proactive, embedded, and scalable approaches. AI-based automation accelerates detection, zero-trust redefines access management, and Security-as-Code operationalizes security within CI/CD workflows. When combined, these paradigms offer a robust foundation for securing modern software applications provided they are implemented with awareness of their limitations and a commitment to continuous improvement.

VII. Evaluating LLMs for Secure Code Review: A Case Study

Large Language Models (LLMs) have found applications across diverse domains such as software development [41], [42], healthcare [39], [40], and education [43]. In software engineering, they are increasingly being explored for tasks like code generation [35], [36], documentation, and review. In the context of secure coding, these models may also offer potential as lightweight vulnerability detectors during early development stages [37], [38]. To assess this capability, we conducted a case study to evaluate how effectively an LLM can identify common security issues in code.

A. Methodology

We selected three intentionally vulnerable code snippets in commonly used web development languages (Python and JavaScript), each representing a typical flaw: SQL injection (SQLi), cross-site scripting (XSS), and broken authentication. The prompts were submitted to publicly accessible interface of GPT (ChatGPT with GPT-4), asking:

"Analyze this code and describe any security vulnerabilities."

The LLM's responses were analyzed for correctness, depth, and completeness.

B. Results

Table II summarizes the results of the LLM based vulnerability analysis. As shown, the model was able to accurately identify and explain a classic SQL injection flaw in Python code. For the XSS example in JavaScript, the LLM correctly flagged the issue but provided only a partial explanation, lacking detail on encoding or proper output sanitization. In contrast, the model failed to recognize a broken authentication scenario, highlighting current limitations in detecting logic based vulnerabilities that require deeper contextual understanding.

C. Discussion

The results show that LLMs can reliably detect certain syntactic vulnerabilities, such as unsanitized SQL queries and basic XSS vectors. However, more nuanced and contextual issues like broken authentication logic were not accurately identified. The LLM often provided generic suggestions without fully understanding the logic or flow of the code. These findings highlight both the strengths and current limitations of LLMs in the secure coding domain.

While promising as support tools for early review, LLMs should not be relied upon as primary security evaluators. They are best viewed as complementary aids that may assist developers, particularly those with limited security experience, in identifying obvious flaws before formal testing or peer review.

D. Comparison with Traditional Security Tools

To contextualize the strengths and limitations of large language models in secure code analysis, Table III presents a structured comparison between LLMs and traditional static and dynamic analysis tools. The comparison covers setup complexity, detection capabilities, explainability, and integration potential in development workflows.

VIII. Future Research Directions

While secure coding practices have significantly evolved, the dynamic nature of software development, evolving threat landscapes, and the rise of distributed systems continue to expose new vulnerabilities. As such, there is a pressing need for further research to address the limitations of current frameworks and to enhance their applicability in modern environments. This section outlines three primary areas for future exploration: education, automation, and adaptation to emerging technologies.

A. Enhancing Secure Coding Education

A fundamental yet underexplored area is the improvement of secure coding education at both academic and professional levels. Many graduates enter the workforce with limited exposure to secure development concepts, increasing the likelihood of introducing vulnerabilities early in the software lifecycle [3], [10]. McGraw [2] emphasizes that security must be viewed as a design principle rather than a post development fix, advocating for its inclusion from the first line of code.

Future research should explore the development of standardized secure coding curricula for undergraduate and graduate programs, possibly aligned with industry frameworks such as OWASP and NIST. Additionally, empirical studies are needed to evaluate the effectiveness of hands on training environments, such as cyber ranges and CTF style exercises, in promoting long term security awareness among students and professionals.

B. Developing Advanced Security Automation Tools

Although current automated tools assist in identifying common vulnerabilities, they are often limited by high false positive rates and an inability to interpret contextual logic. Enhancing these tools with more sophisticated machine learning and natural language processing capabilities could improve their precision, especially for detecting semantic flaws or business logic errors [12], [15].

Sutton et al. [15] highlight that fuzzing and automated scanning can uncover obscure vulnerabilities, but expert oversight remains essential to interpret results accurately. Future research could focus on developing explainable AI models in security automation tools that not only detect flaws but also provide human understandable justifications. Moreover, integrating such models seamlessly into CI/CD pipelines could facilitate real time vulnerability detection during rapid deployment cycles.

C. Adapting Secure Coding Practices for Emerging Technologies

The rapid adoption of modern architectures such as microservices, serverless computing, and cloud native platforms has introduced novel security challenges. Traditional secure coding frameworks often fall short when applied to highly dynamic and decentralized environments [8]. These platforms introduce complexities related to ephemeral services, infrastructure-as-code, and third party integrations, which require more adaptive security methodologies.

Shostack [17] and others advocate for deeper integration of threat modeling and security automation tailored to cloud environments. Future work should investigate how to extend the SSDLC model to accommodate DevSecOps workflows, leveraging Security-as-Code principles to ensure continuous compliance and enforcement. Furthermore, researchers should explore how secure coding can be embedded at the infrastructure layer, particularly in container orchestration platforms such as Kubernetes.

D. Toward an Evolving Research Agenda

In conclusion, future research in secure coding must balance foundational education, intelligent automation, and architectural adaptability. Addressing these gaps will not only improve the security of web applications but also ensure that secure coding evolves alongside modern software engineering practices. A collaborative effort between academia, industry, and open-source communities will be essential to advance practical, scalable, and effective security solutions for next generation software systems.

TABLE II
LLM DETECTION OF CODE VULNERABILITIES.

Vulnerability Type	Language	LLM Detected?	Explanation Accurate?
SQL Injection	Python	Yes	Yes
Cross-Site Scripting (XSS)	JavaScript	Yes	Partially
Broken Authentication	JavaScript	No	N/A

TABLE III
COMPARISON OF LLMS AND TRADITIONAL STATIC/DYNAMIC ANALYSIS TOOLS

Feature	LLMs (e.g., ChatGPT)	Traditional Tools
Setup Time	None (zero setup)	Requires installation and configuration
Input Format	Natural language or code snippets	Source code, binaries, or compiled artifacts
Output Type	Human-readable explanation	Structured reports with warnings/errors
Context Understanding	Partial, limited to prompt scope	Strong syntactic analysis, limited semantic depth
Detection of Logic Flaws	Limited	Possible with dynamic tools, but not guaranteed
False Positive Rate	Low to Medium	Medium to High (especially in static tools)
CI/CD Integration	Manual or external scripting required	Built-in support in modern DevSecOps pipelines
Learning Curve	Low (conversational interface)	Medium to High, depending on tool complexity
Explainability of Results	High (natural language reasoning)	Often limited to error codes or technical traces

IX. CONCLUSION

Secure coding remains a foundational discipline in protecting web applications from increasingly sophisticated cyber threats. This paper provided a comprehensive review of key secure coding frameworks including the OWASP Top 10, the NIST Cybersecurity Framework, and the Secure Software Development Lifecycle (SSDLC) all of which offer structured methodologies for reducing software vulnerabilities [4], [6], [7]. Despite their widespread availability, consistent adoption across organizations continues to face obstacles such as inadequate training, limited resources, and the rapid pace of software delivery [1], [3], [9].

To address these challenges, emerging trends such as AI-driven vulnerability detection, zero-trust security models, and Security-as-Code practices within DevSecOps pipelines offer promising directions [5], [12], [17]. However, these approaches must be further refined and operationalized to ensure scalability and effectiveness in production environments.

Future research should focus on embedding secure coding principles into formal education, enhancing the intelligence and interpretability of automated security tools, and evolving traditional models to align with modern cloud native and distributed systems [8], [10], [15]. Bridging these gaps will be essential for developing software systems that are not only functional and performant but also inherently secure by design.

By synthesizing foundational frameworks, current implementation challenges, and cutting edge trends, this work contributes a holistic perspective to the secure coding landscape offering valuable guidance to researchers, educators, and practitioners seeking to advance the state of secure software development.

- Secure coding remains foundational, yet under-adopted.
- LLMs provide helpful early feedback but cannot replace expert review.
- Embedding secure practices in education, automation, and cloud-native workflows is critical for future resilience.

REFERENCES

[1] M. Howard and D. LeBlanc, ”Writing Secure Code,” 2nd ed., Microsoft Press, 2003.
[2] G. McGraw, ”Software Security: Building Security In,” Addison-Wesley Professional, 2006.
[3] R. C. Seacord, ”Secure Coding in C and C++,” 2nd ed., Addison-Wesley Professional, 2013.
[4] J. Viega and G. McGraw, ”Building Secure Software: How to Avoid Security Problems the Right Way,” Addison-Wesley Professional, 2001.
[5] F. Long, D. Mohindra, R. C. Seacord, D. F. Sutherland, and D. Svoboda, ”Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs,” Addison-Wesley Professional, 2014.
[6] OWASP Foundation, ”OWASP Top Ten: The Ten Most Critical Web Application Security Risks,” 2021. [Online]. Available: https://owasp.org/www-project-top-ten/
[7] NIST, ”Framework for Improving Critical Infrastructure Cybersecurity,” Version 1.1, 2018. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.04162018.pdf
[8] S. T. King, G. W. Dunlap, and P. M. Chen, ”Operating System Support for Virtual Machines,” USENIX ATEC, 2003.
[9] W. G. J. Halfond, J. Viegas, and A. Orso, ”A Classification of SQL Injection Attacks and Countermeasures,” IEEE ISSSE, 2006.
[10] Y. Xie and A. Aiken, ”Static Detection of Security Vulnerabilities in Scripting Languages,” 15th USENIX Security Symposium, 2006.
[11] C. Anley, “Advanced SQL Injection in SQL Server Applications,” NGSSoftware, 2002. [Online]. Available: http://www.ngssoftware.com/papers/advanced_sql_injection.pdf
[12] P. Bisht and V. N. Venkatakrishnan, ”XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks,” in Proc. of the 5th Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), 2008.
[13] D. Scott and R. Sharp, ”Abstracting Application-Level Web Security,” in Proc. of the 11th Int. Conf. on World Wide Web (WWW), 2002.
[14] S. Christey and R. A. Martin, ”Vulnerability Type Distributions in CVE,” Mitre Corporation, 2007.
[15] M. Sutton, A. Greene, and P. Amini, ”Fuzzing: Brute Force Vulnerability Discovery,” Addison-Wesley Professional, 2007.
[16] D. Crockford, ”JavaScript: The Good Parts,” O'Reilly Media, 2008.
[17] A. Shostack, ”Threat Modeling: Designing for Security,” Wiley, 2014.
[18] S. Lipner, ”The Trustworthy Computing Security Development Lifecycle,” ACSAC, 2004.
[19] J. A. Whittaker and H. H. Thompson, ”How to Break Software Security: Effective Techniques for Security Testing,” Addison-Wesley, 2004.
[20] M. Jones and S. Rastogi, ”A Comparative Analysis of Static and ”Dynamic Analysis Tools,” *IEEE Security & Privacy*, vol. 10, no. 3, pp. 32-45, 2021.
[21] R. Sharma et al., ”Automated Security Scanning Tools: Benefits and Limitations,” Elsevier, 2022.
[22] A. Rajput et al., ”Comparative Study of Secure Coding Frameworks,” Springer, 2023.
[23] S. Ghobadi and M. Tavana, ”Blockchain Security for Web Applications,” Journal of Information Technology, 2023.

[24] T. A. Nidecki, ”7 Reasons Why Development Teams Skip Security Steps,” Invicti, 2023. [Online]. Available: https://www.invicti.com/blog/web-security/7-reasons-why-development-teams-skip-security-steps/

[25] OpenSSF, ”Why Are Organizations Struggling to Implement Secure Software Development?,” 2024. [Online]. Available: https://openssf.org/blog/2024/07/05/why-are-organizations-struggling-to-implement-secure-software-development/

[26] Upcore, “2024 Application Security: Tools, Trends, Practices and Challenges,” [Online]. Available: https://www.upcoretech.com/insights/application-security-tools-trends-best-practices-challenges/

[27] Contrast Security, ”The State of DevSecOps Report,” 2024. [Online]. Available: https://www.contrastsecurity.com/hubfs/DocumentsPDF/The-State-of-DevSecOps

[28] OWASP Foundation, ”OWASP Top Ten — OWASP Foundation,” [Online]. Available: https://owasp.org/www-project-top-ten

[29] G. V. de Ven, ”Understanding the Limitations of OWASP Top 10 Assessment,” Conquer Your Risk, 2022. [Online]. Available: https://www.conquer-your-risk.com/2022/11/15/understanding-the-limitations-of-owasp-top-10-assessment/

[30] A. Peak, ”Benefits and Challenges in Implementing NIST CSF,” Audit Peak, 2023. [Online]. Available: https://www.auditpeak.com/challenges-in-implementing-nist-csf/

[31] NIST, ”The NIST Cybersecurity Framework (CSF) 2.0,” 2024. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.29.pdf

[32] Palo Alto Networks, ”What is Secure Software Development Lifecycle (secure SDLC)?,” [Online]. Available: https://www.paloaltonetworks.com/cyberpedia/what-is-secure-software-development-lifecycle

[33] L. Fisher, ”Zero-Trust Security in Software Development,” Communications of the ACM, 2024. [Online]. Available: https://cacm.acm.org/blogcacm/zero-trust-security-in-software-development/

[34] Z. Ghalleb, ”What is Security as Code (SAC)?,” Wiz.io, 2024. [Online]. Available: https://www.wiz.io/academy/security-as-code-sac

[35] Wang, Jianxun, and Yixiang Chen. ”A review on code generation with llms: Application and evaluation.” 2023 IEEE International Conference on Medical Artificial Intelligence (MedAI). IEEE, 2023.

[36] Torkamani, Mohammad Jalili, et al. ”Assertify: Utilizing Large Language Models to Generate Assertions for Production Code.” arXiv preprint arXiv:2411.16927 (2024).

[37] Zhou, Xin, et al. ”Large language model for vulnerability detection and repair: Literature review and the road ahead.” ACM Transactions on Software Engineering and Methodology 34.5 (2025): 1-31.

[38] Torkamani, Mohammad Jalili, et al. ”Streamlining Security Vulnerability Triage with Large Language Models.” arXiv preprint arXiv:2501.18908 (2025).

[39] Cascella, Marco, et al. ”Evaluating the feasibility of ChatGPT in healthcare: an analysis of multiple clinical and research scenarios.” Journal of medical systems 47.1 (2023): 33.

[40] Kermani, Arshia, Veronica Perez-Rosas, and Vangelis Metsis. ”A Systematic Evaluation of LLM Strategies for Mental Health Text Analysis: Fine-tuning vs. Prompt Engineering vs. RAG.” arXiv preprint arXiv:2503.24307 (2025).

[41] Hou, Xinyi, et al. ”Large language models for software engineering: A systematic literature review.” ACM Transactions on Software Engineering and Methodology 33.8 (2024): 1-79.

[42] Jin, Haolin, et al. ”From llms to llm-based agents for software engineering: A survey of current, challenges and future.” arXiv preprint arXiv:2408.02479 (2024).

[43] Stamper, John, Ruiwei Xiao, and Xinying Hou. ”Enhancing llm-based feedback: Insights from intelligent tutoring systems and the learning sciences.” International Conference on Artificial Intelligence in Education. Cham: Springer Nature Switzerland, 2024.

[44] Kiashemshaki, Kiana. ”Remote Agile Tools and Technologies: A Comparative Analysis of Communication Methods.” *International Journal of Computer Applications Technology and Research* 10.14 (2024): 26–32. DOI: 10.7753/IJCATR1304.1004.

Appendix

Appendix: LLM Prompts and Responses

This appendix presents the full prompts and LLM-generated responses used in the case study described in Section VI. Each prompt was submitted to ChatGPT (GPT-4), and the model's output is shown below without modification.

Case 1: SQL Injection

Prompt

```
def login():
    username = request.args.get("user")
    password = request.args.get("pass")
    query = "SELECT * FROM users WHERE name =
        '" + username + "' AND pwd = '" +
        password + "'"
    cursor.execute(query)
```

Listing 1. Prompt for SQL Injection

LLM Response

```
This code is vulnerable to SQL Injection
    attacks.

SQL Injection is a code injection technique
    that attackers can use to insert malicious
     SQL statements into input fields for
    execution. This can allow the attacker to
    view data that they are not authorized to
    access, manipulate or delete data, or even
     execute administration operations on the
    database.

In this code, the username and password are
    directly concatenated into the SQL query
    without any form of sanitization or
    parameterization. This means that an
    attacker could provide a username or
    password that contains SQL code, which
    would then be executed by the database.

For example, an attacker could use the
    username `admin' --` . This would cause
    the SQL query to become `SELECT * FROM
    users WHERE name = 'admin' --' AND pwd = '
    '`, effectively commenting out the
    password check and allowing the attacker
    to login as the admin user without knowing
     the password.

To mitigate this vulnerability, you should use
     parameterized queries or prepared
    statements, which can ensure that user
    input is always treated as literal data
    and not executable code. Here is an
    example of how you could modify the code
    to use parameterized queries in Python
    with a SQLite database:

def login():
    username = request.args.get("user")
    password = request.args.get("pass")
    query = "SELECT * FROM users WHERE name =
        ? AND pwd = ?"
    cursor.execute(query, (username, password)
        )

In this version of the code, the `?`
    characters are placeholders that the `
    execute` method replaces with the `
    username` and `password` variables in a
    safe manner that prevents SQL Injection.
```

Listing 2. LLM Response for SQL Injection

Case 2: Cross-Site Scripting (XSS)

Prompt

```
<!DOCTYPE html>
<html>
  <body>
    <form method="GET">
      <input name="msg" />
      <button type="submit">Submit</button>
    </form>
    <p>Your message: <script>document.write(
        location.search.split('=')[1])</script
        ></p>
  </body>
</html>
```

Listing 3. Prompt for XSS

LLM Response

```
The code above is a simple HTML form that
    takes a user's input and displays it on
    the page. However, it has a significant
    security vulnerability related to Cross-
    Site Scripting (XSS).

The vulnerability lies in the line where it
    writes the user's input directly to the
    document without any sanitization or
    encoding:

<script>document.write(location.search.split
    ('=')[1])</script>

This line takes the value from the URL
    parameter 'msg' and writes it directly to
    the page. If a user enters a string that
    includes JavaScript code, that code will
    be executed when the page loads. This is a
     classic example of a reflected XSS
    vulnerability.

For example, an attacker could craft a URL
    like http://example.com/?msg=<script>
    malicious_code_here</script>. If someone
    visits this URL, the malicious JavaScript
    code will be executed.

To fix this vulnerability, the application
    should properly encode or sanitize user
    inputs before displaying them on the page.
     This can be done using various methods
    depending on the programming language and
    framework used. For JavaScript, one could
    use the `textContent` property or `
    innerText` property instead of `document.
    write()`, as these properties do not
    interpret the input as HTML or JavaScript
    code.
```

Listing 4. LLM Response for XSS

Case 3: Broken Authentication

Prompt

```
def login():
    username = request.form['username']
    user = db.find_user(username)
    if user:
        session['logged_in'] = True
        return "Welcome!"
    else:
        return "User not found"
```

Listing 5. Prompt for Broken Authentication

LLM Response

```
The code provided is a simple login function
    that checks if a user exists in the
    database and then logs them in if they do.
     However, there are several security
    vulnerabilities present:

1. No Password Verification: The function does
     not check for a password. It only checks
    if the username exists in the database.
    Anyone who knows a valid username can log
    in.

2. No Input Sanitization: The function does
    not sanitize the input from the user. This
     leaves the application vulnerable to SQL
    Injection attacks. An attacker could
    potentially input malicious SQL code as
    the username to manipulate the database.

3. No Encryption: The function does not
    encrypt the username before storing it in
    the session. This could potentially expose
     sensitive user information.

4. No Brute Force Protection: The function
    does not limit the number of login
    attempts. This makes it vulnerable to
    brute force attacks, where an attacker
    tries many different usernames until they
    find one that works.

5. No Error Handling: The function does not
    handle any errors that may occur during
    the login process. This could potentially
    expose sensitive information about the
    system or the database.

6. No Use of HTTPS: If this function is used
    in a web application that does not use
    HTTPS, the username could be intercepted
    by an attacker.

7. Session Fixation: The function does not
    regenerate the session ID after successful
     login. This makes it vulnerable to
    session fixation attacks, where an
    attacker sets a victim's session ID before
     they log in.

8. No Logging: The function does not log any
    events. This makes it difficult to detect
    and respond to security incidents.
```

Listing 6. LLM Response for Broken Authentication