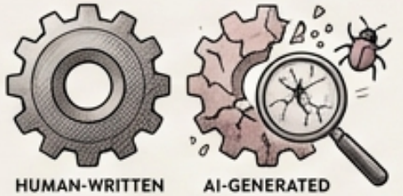
Module 3.1: Secure Coding Practices — Writing Defensible Code in an AI-Augmented World

Navigating Cybersecurity Challenges with Intelligent Tools and Timeless Principles

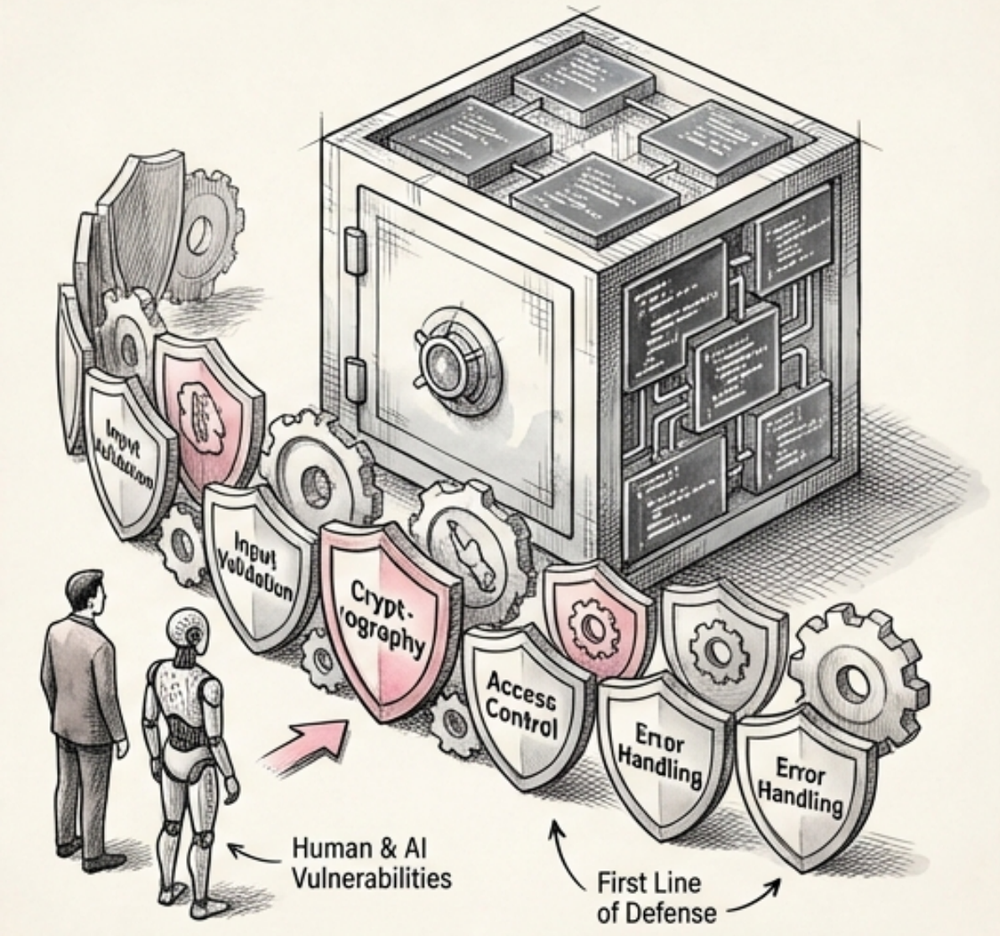
Introduction: Secure Coding in the Age of AI

- AI code generation tools are becoming increasingly prevalent in software development.
- AI-generated code presents a significantly higher risk of containing vulnerabilities compared to human-written code.
- Veracode's 2025 report indicates that AI-generated code is 2.74 times more likely to harbor security flaws.
- This module equips developers with the essential knowledge to write defensible and secure code, particularly in an AI-augmented development environment.
- We will cover OWASP secure coding practices, common weaknesses, language-specific vulnerabilities, and preventative measures.



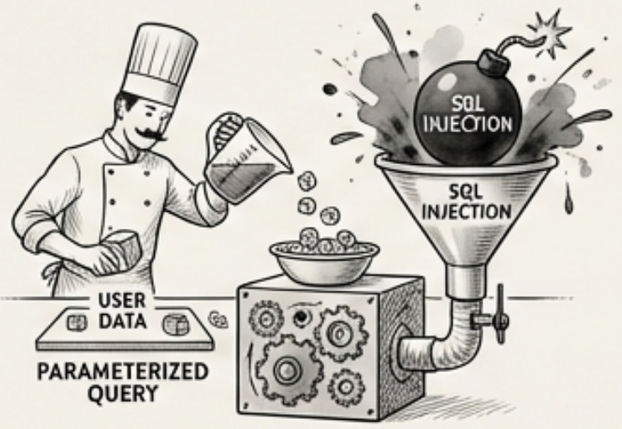
OWASP Secure Coding Practices: The Foundation of Defensible Code

- OWASP (Open Web Application Security Project) defines a set of crucial secure coding practices that every developer should follow.
- These practices cover 14 key areas, providing a comprehensive framework for mitigating common vulnerabilities.
- Key OWASP areas include Input Validation, Output Encoding, Authentication, Session Management, Access Control, Cryptography, and Error Handling.
- Other vital areas encompass Data Protection, Communication Security, System Configuration, Database Security, File Management, and General Coding Practices.
- Adopting OWASP practices is the first line of defense against both human-written and AI-generated vulnerabilities.

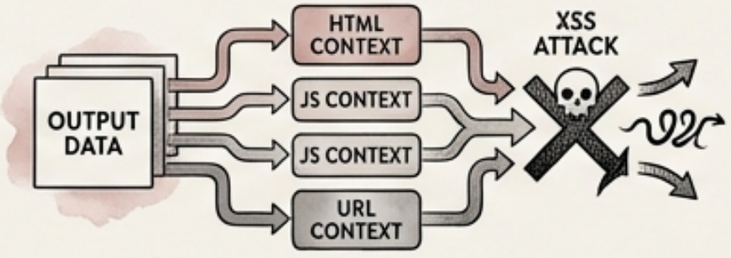


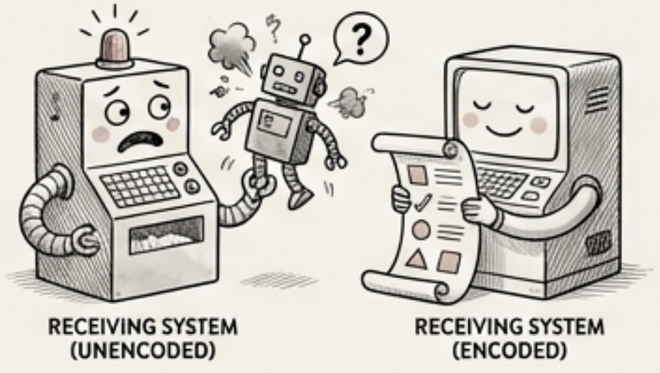
OWASP: Input Validation & Output Encoding

- Input Validation: Implement allowlists over denylists to restrict acceptable inputs, preventing malicious data from entering the system.
- Input Validation: Utilize parameterized queries to prevent SQL injection vulnerabilities, especially when dealing with user-supplied data.

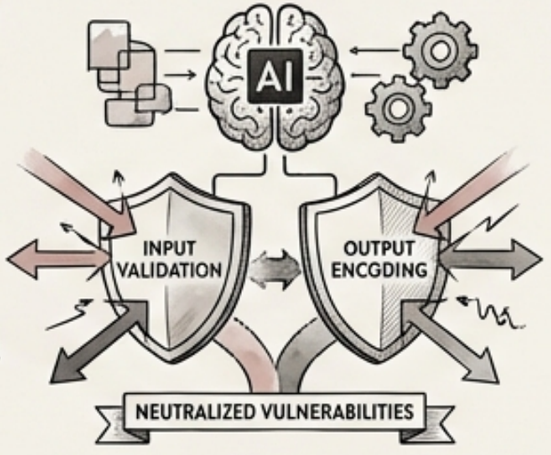


- Output Encoding: Encode output appropriately based on the context, such as HTML, JavaScript, URL, CSS, or SQL, to prevent cross-site scripting (XSS) and other injection attacks.





- Correct output encoding ensures that data is interpreted as data, not as executable code, by the receiving system.
- Proper input validation and output encoding are crucial to neutralizing many AI-introduced vulnerabilities.



THE CWE TOP 25 MOST DANGEROUS SOFTWARE WEAKNESSES

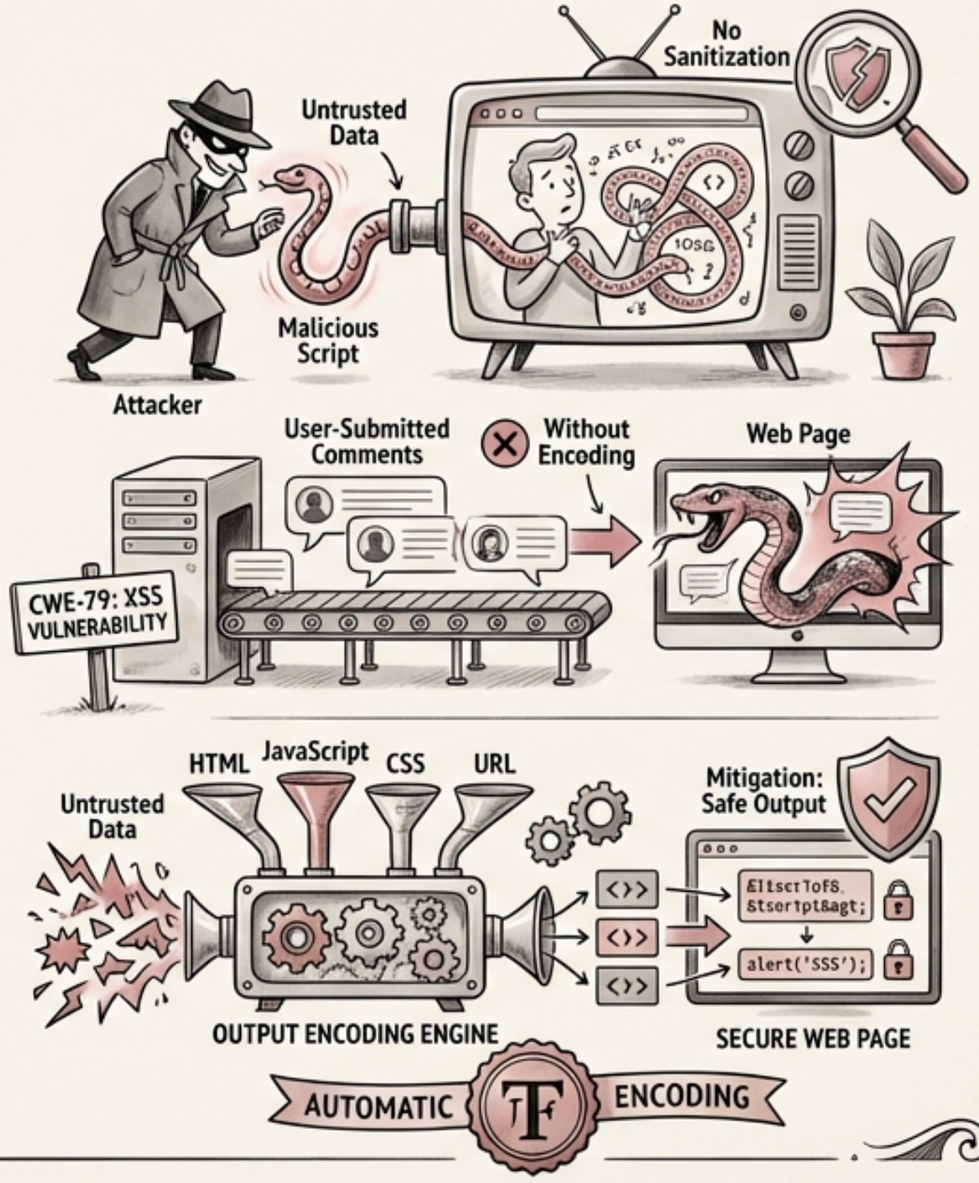
- The CWE Top 25 is a prioritized list of the most widespread and dangerous software weaknesses.
- Understanding the CWE Top 25 enables developers to focus their security efforts on the most critical risks.
- Knowing common CWEs and their fixes is critical when reviewing AI-generated code.
- This slide will examine several of these weaknesses in detail with code examples and potential solutions.

CWE-787: Out-of-bounds Write - Code Example

- CWE-787: Out-of-bounds Write occurs when software writes data beyond the allocated memory boundary.
- This can lead to crashes, data corruption, or even arbitrary code execution.
- **Example:** A buffer overflow occurs when a program attempts to write more data to a buffer than it can hold.
- **Mitigation:** Carefully validate the size of input data and use safe memory allocation functions.
- Consider using memory-safe languages or libraries to prevent out-of-bounds writes.

CWE-79: Cross-Site Scripting (XSS) - Code Example

- **CWE-79:** Cross-Site Scripting (XSS) occurs when an application injects untrusted data into a web page without proper sanitization.
- This allows attackers to execute malicious scripts in the context of the user's browser.
- **Example:** Displaying user-submitted comments directly without encoding them can lead to XSS.
- **Mitigation:** Use proper output encoding techniques to escape HTML, JavaScript, CSS, and URL contexts.
- Leverage templating engines and frameworks that provide automatic output encoding to minimize the risk of XSS.



CWE-89: SQL Injection - Code Example

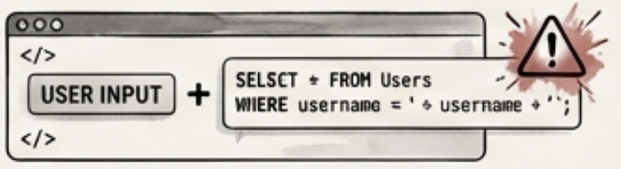
CWE-89: SQL Injection occurs when an application uses untrusted data to construct SQL queries without proper sanitization.

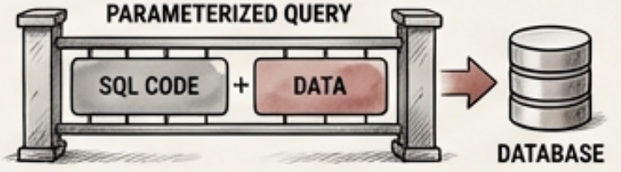
This allows attackers to inject malicious SQL code into the query, potentially gaining unauthorized access to the database.

Example: Concatenating user input directly into an SQL query can lead to SQL injection.

Mitigation: Use parameterized queries (also known as prepared statements) to separate data from SQL code.

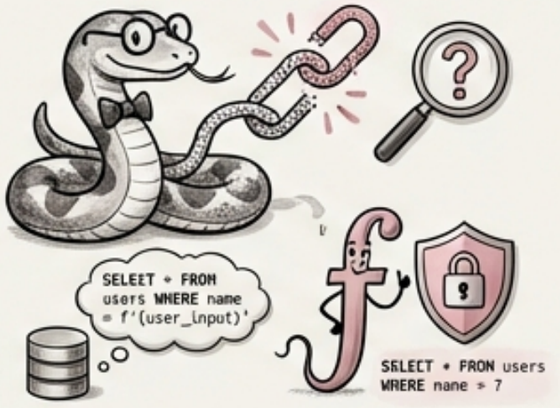
Avoid dynamic SQL query construction and rely on object-relational mapping (ORM) frameworks for database interactions.

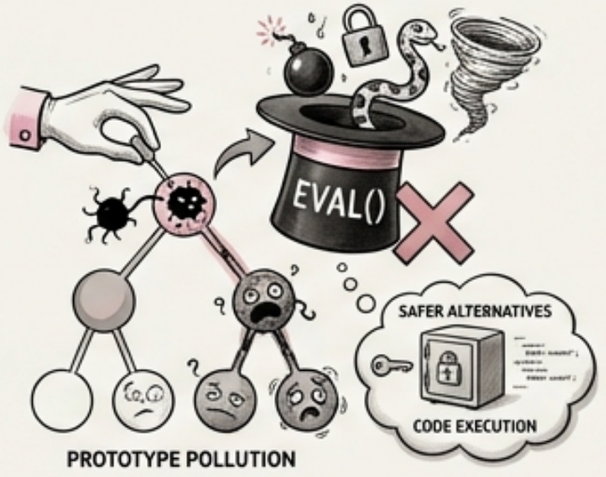




Language-Specific Vulnerabilities: Python, Javascript/Typescript

- **Python:** Avoid SQL injection through f-strings by using parameterized queries instead.
- **Python:** Pickle deserialization can be exploited to execute **arbitrary code**; use it with **extreme caution** and only with **trusted data**.
- **Python:** Avoid the `eval()` function, which can execute arbitrary code from user input; use **safer alternatives** for dynamic code execution.
- **JavaScript/TypeScript: Prototype pollution** can be exploited to modify object properties, leading to unexpected behavior or security vulnerabilities.
- **JavaScript/TypeScript:** Avoid using `innerHTML` with untrusted data, as it can lead to XSS vulnerabilities.





Language-Specific Vulnerabilities: Java, Go, and Rust

- Java: Deserialization attacks can occur when deserializing untrusted data; use object streams carefully and avoid deserializing data from unknown sources.
- Java: XML External Entity (XXE) injection can occur when parsing XML documents; disable external entities processing to prevent attacks.
- Java: JNDI injection can occur when using the Java Naming and Directory Interface (JNDI); validate and sanitize JNDI lookups to prevent remote code execution.
- Go: Goroutine leaks can occur if goroutines are not properly terminated; ensure that all goroutines exit when they are no longer needed.
- Go: Always check error returns from functions to handle errors properly and prevent unexpected behavior.

AI Code Vulnerability Data: A Stark Reality

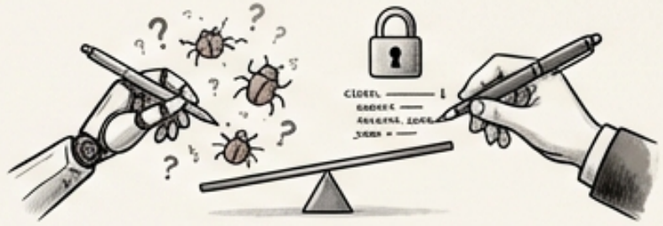
1. **Veracode:** AI-generated code is 2.74x more likely to contain security flaws compared to human-written code (2025).

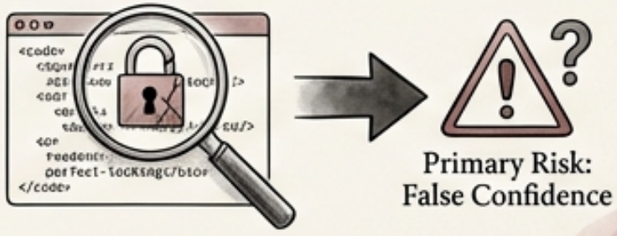
2. **CodeRabbit:** AI code scores 10.83 issues per thousand lines of code (KLOC), versus 6.45 for human-written code.

3. **Stanford Study:** Developers using AI assistants produced significantly less secure code than those who did not.

4. The Stanford study also found that developers using AI assistants were more confident in the security of their code, despite its lower security.

5. This false sense of security is a primary risk associated with using AI code generation tools.



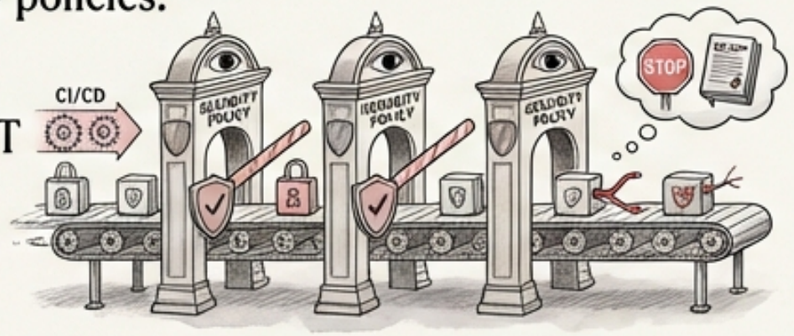


PRE-COMMIT HOOKS: PROACTIVE VULNERABILITY PREVENTION

- Pre-commit hooks are automated scripts that run before code is committed to a repository, providing an early warning system for potential issues.
- Implement pre-commit hooks for Static Application Security Testing (SAST) scanning to identify code vulnerabilities early in the development process.
- Use pre-commit hooks for secret detection to prevent sensitive information like API keys and passwords from being committed to the repository.
- Implement pre-commit hooks for license checking to ensure that code dependencies comply with licensing requirements.
- Configure pre-commit hooks to automatically format code and enforce coding style guidelines, reducing inconsistencies and improving code quality.

CI Pipeline Gates: – Enforcing Security Policies

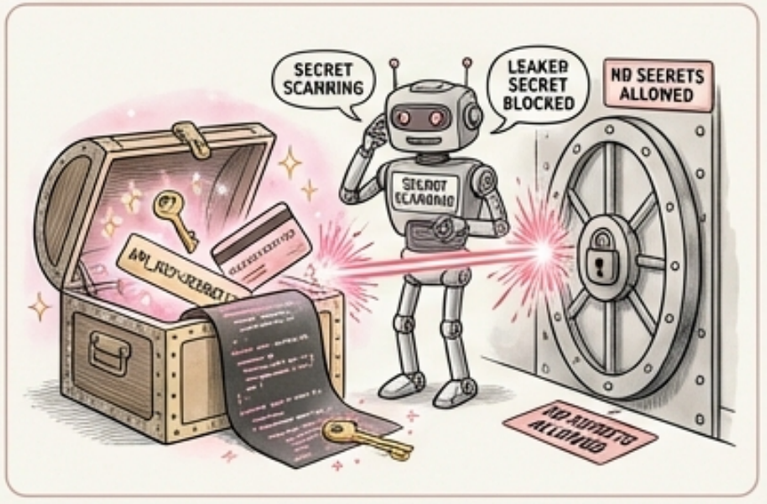
- CI (Continuous Integration) pipeline gates are automated checks that are executed as part of the CI/CD pipeline to enforce security policies.



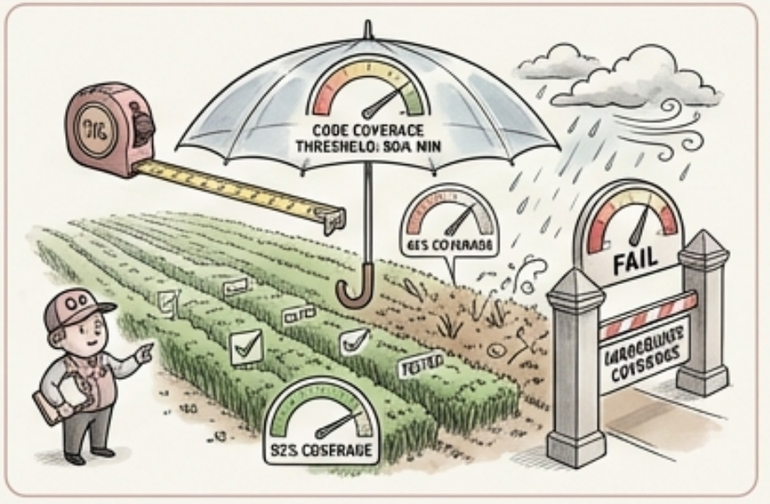
- Implement SAST pipeline gate vulnerabilities

- Use SCA (Software Composition Analysis) as a CI pipeline gate to block code with known exploited vulnerabilities from being deployed.

- Implement secret scanning as a CI pipeline gate to always block code with leaked secrets from being deployed.



- Establish a minimum code coverage threshold as a CI pipeline gate to ensure adequate testing coverage.



Zero-Tolerance Enforcement for AI-Generated Code

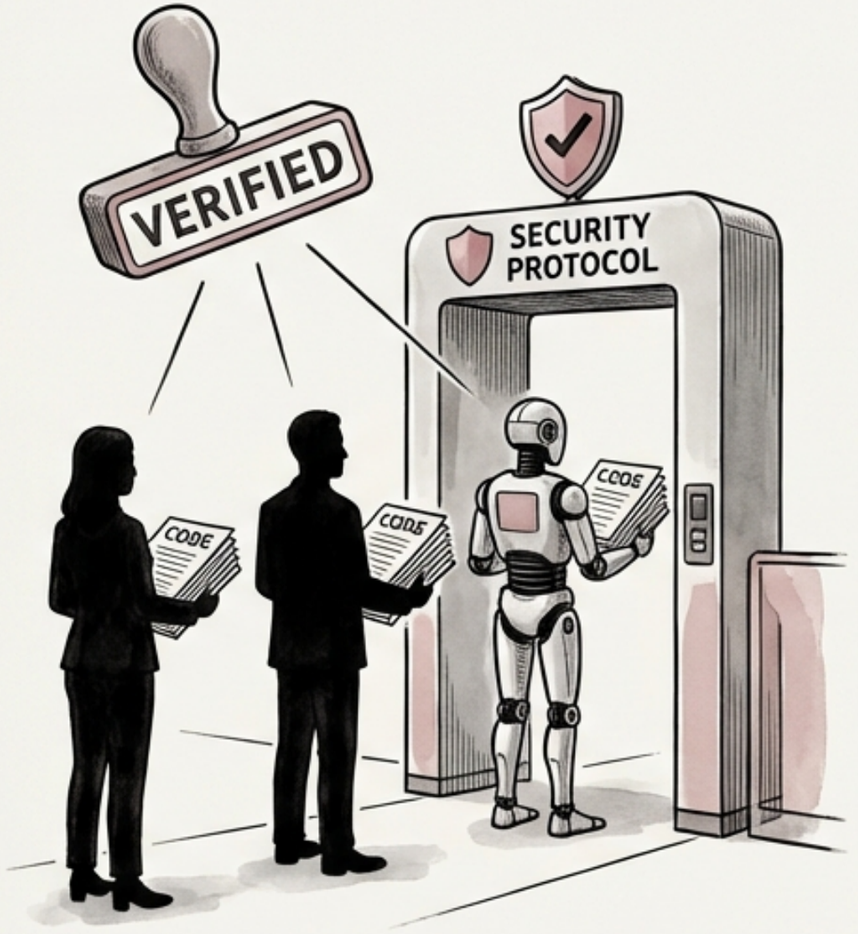
- All pre-commit hooks and CI pipeline gates should be applied uniformly, regardless of whether the code was written by a human or generated by AI.

- AI-generated code should not receive any special treatment or exemptions from security checks.

- Enforce the same standards for SAST, SCA, secret scanning, code coverage, and other security measures for all code, including AI-generated code.

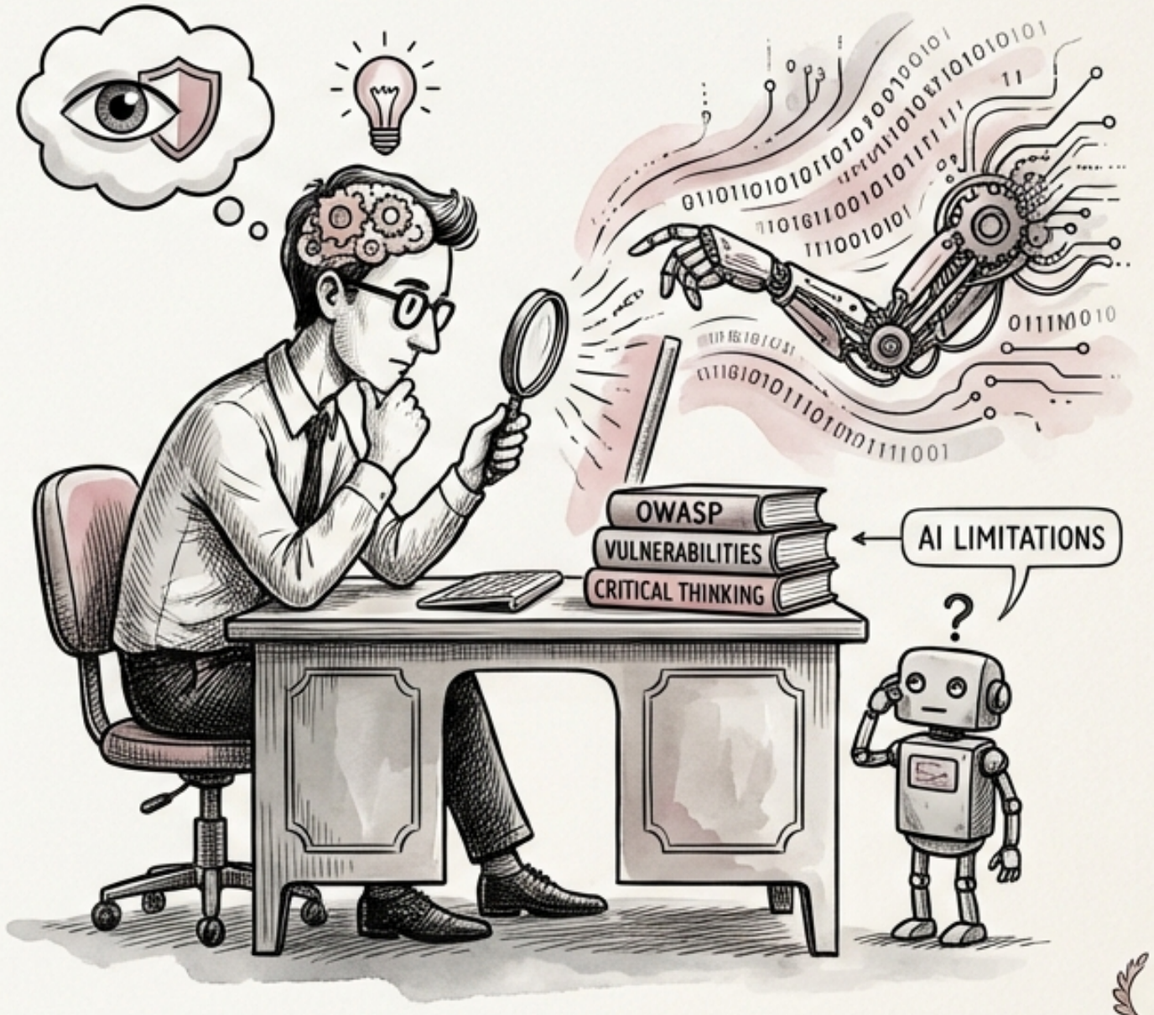
- Automated security tools are essential for identifying vulnerabilities in AI-generated code, as manual review may be insufficient.

- Developers should be trained to critically review AI-generated code and understand the potential security implications.

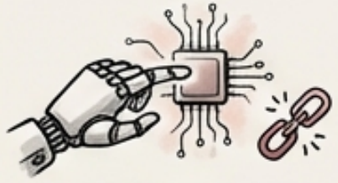


DEVELOPER TRAINING: THE HUMAN ELEMENT

- Even with automated tools, developer training remains crucial for writing secure code and reviewing AI-generated code effectively.
- Developers should be trained on OWASP secure coding practices, common vulnerabilities, and language-specific security considerations.
- Training should cover how to critically evaluate AI-generated code and identify potential security flaws.
- Developers need to understand the limitations of AI code generation tools and avoid over-reliance on them.
- Encourage developers to continuously update their knowledge of security best practices and emerging threats.



Conclusion: Embracing AI Responsibly Through Secure Coding



- AI code generation tools offer potential benefits, but they also introduce new security risks.

- AI-generated code is demonstrably more vulnerable, necessitating rigorous security practices.

- Implementing OWASP secure coding practices, addressing CWE Top 25 weaknesses, and understanding language-specific vulnerabilities are essential.

- Pre-commit hooks and CI pipeline gates provide automated enforcement of security policies.

- Zero-tolerance enforcement is necessary for AI-generated code; apply the same stringent security checks as for human-written code.

Thank You

- Questions?