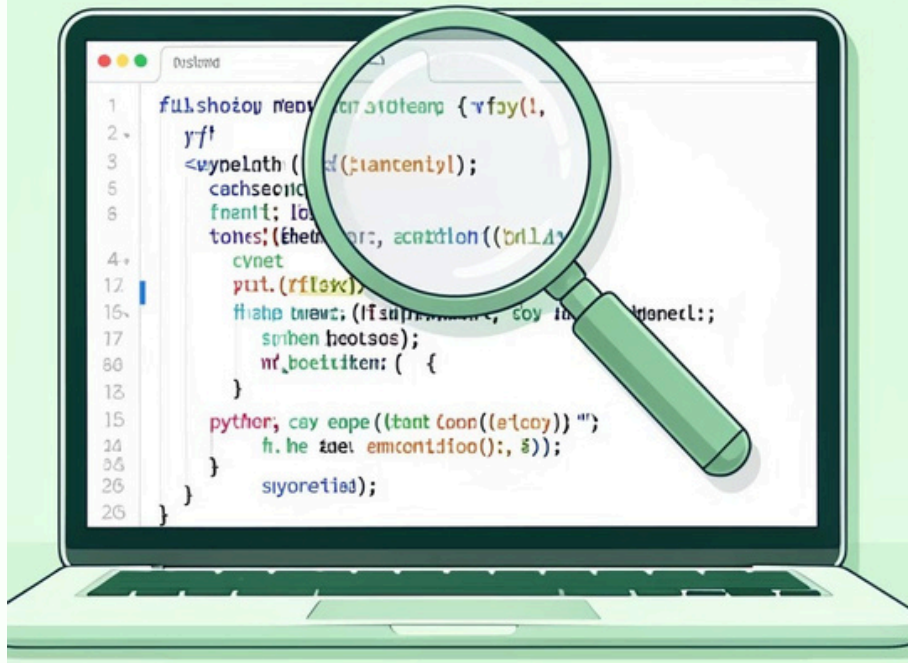
Secure Coding Review: Auditing a Python Web Application



Why Secure Coding Reviews Matter

1

Breach Prevention

70% of breaches originate from software vulnerabilities, highlighting the critical need for proactive security measures.

2

Cost Effinciecy

Detecting vulnerabilities early in the development lifecycle can save millions in remediation costs and preserve brand reputation.

3

Proactive Defence

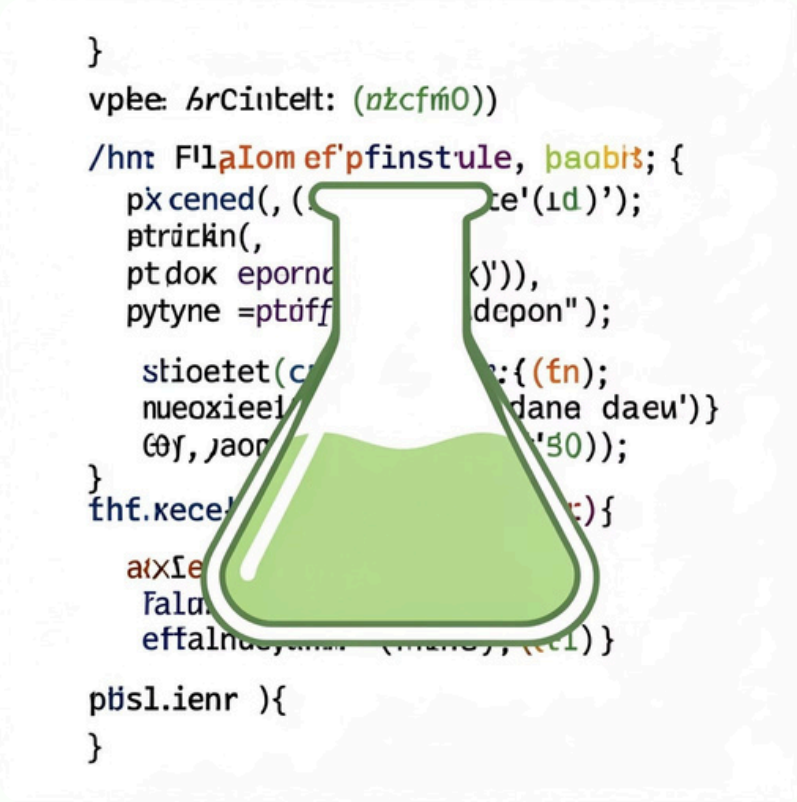
Secure coding acts as a front-line defence, shifting focus from reactive patching to building inherently secure applications.

Our Target: Python Flask Web Application

We focused our audit on a Python Flask web application, a popular choice for developing web APIs and dynamic web services. Flask's lightweight nature makes it a favourite among startups and established enterprises for its flexibility and ease of use.

Common Vulnerabilitie :

- Injection flaws (e.g., SQL, XSS)
- Authentication and authorization weaknesses
- Insecure configurations and default settings
- Sensitive data exposure



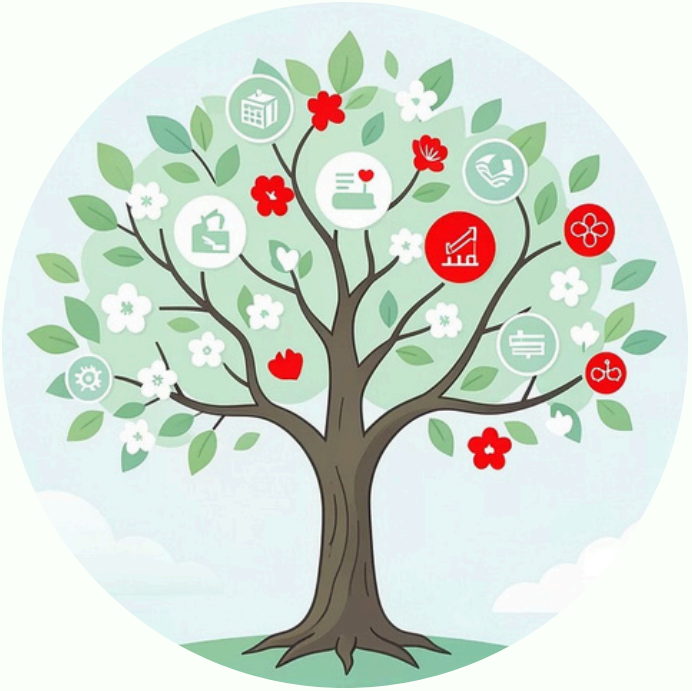
Tools & Method Used for Review

SAST Tool

Utilised Bandit for Python-specific security linting and SonarQube for comprehensive static analysis across the codebase.

Manual Inspection

Focused on critical areas: authentication mechanisms, user input handling, and robust error management practices to identify subtle logical flaws.



Dependency Analysis

Scanned for vulnerable third-party libraries and packages, including outdated versions with known Common Vulnerabilities and Exposures (CVEs).

Key Vulnerabilities Discovered

- **SQLInjectionRisk:** Unsanitised user input wasfound in a login endpoint, creating a clear pathway for SQL injection attacks.
- **Hardcoded Secrets:** Cryptographic secrets and API keys were directly embedded in the source code, posing a significant exposure risk.
- **Insecure Session Management:** Session cookies lacked critical **HttpOnly** and **Secure** flags, making them vulnerable to client-side attacks.
- **Outdated Dependencies:** The application relied on an old version of Flask (1.1.1), containing several publicly known CVEs.

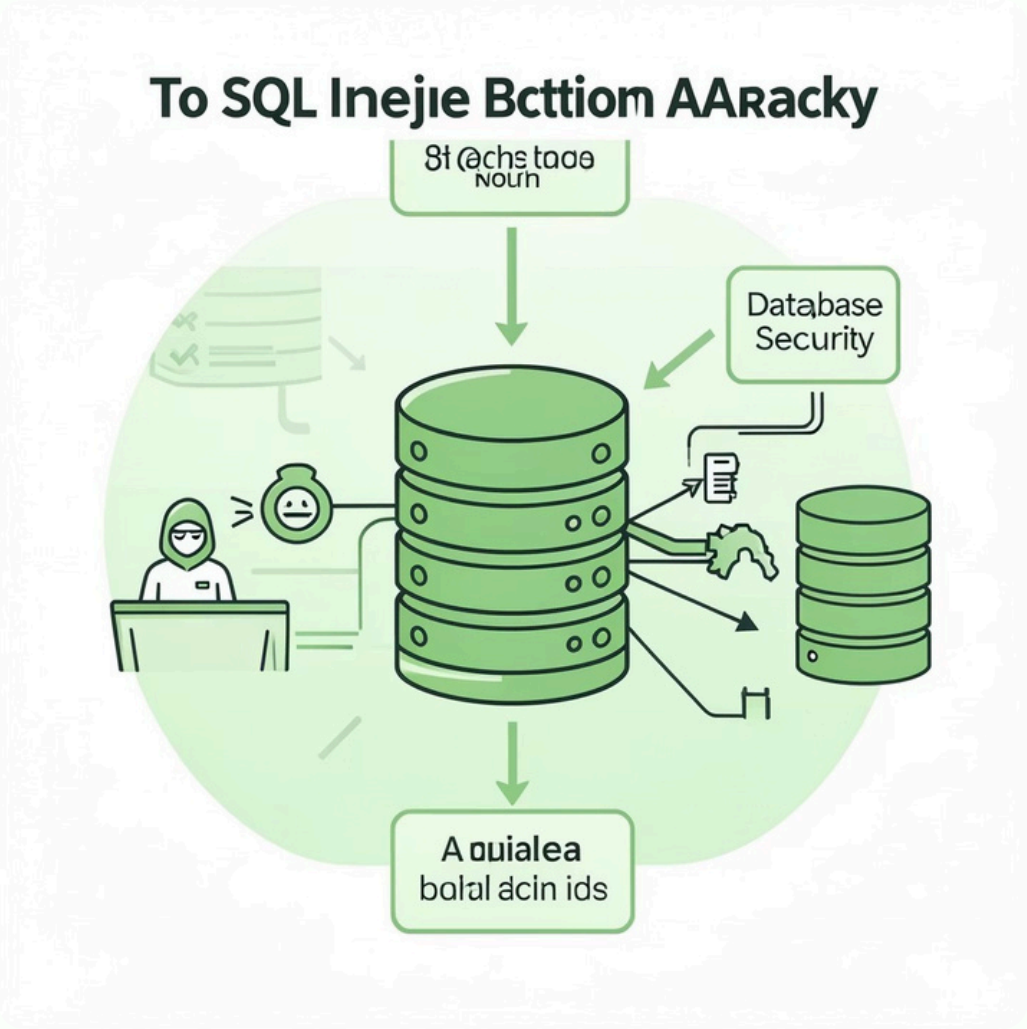
Demonstration : SQL Injection Risk

Vulnerable Code Snippet:

```
query = "SELECT * FROM users WHERE username = '" +
user_input + "'"
```

This example illustrates how a raw SQL query, constructed with simple string concatenation, is susceptible to SQL Injection.

A malicious actor could inject SQL commands via `user_input` to bypass authentication or extract sensitive data.



Bandit security tool correctly identified this as a **high-severity issue** during static analysis.

Recommendations & Best Practices

Prevent Injection

Use **parameterised queries** or Object-Relational Mapping (ORM) frameworks to automatically sanitise input.

Secure Secret

Store sensitive data in **environment variables** or dedicated secret management vaults; never hardcode them.

Enhance Cookies

Enforce **HttpOnly** , **Secure**, and **SameSite** attributes for all session cookies.

Update Dependencies

Regularly **update all dependencies** and actively monitor for new CVEs.

Validate Input

Implement **strict input validation** and output encoding for all user-supplied data.

Secure Coding Principle to Adopt

Input Sanitisation

All external input must be rigorously sanitised and validated to prevent malicious data from compromising the system.

Least Privilege

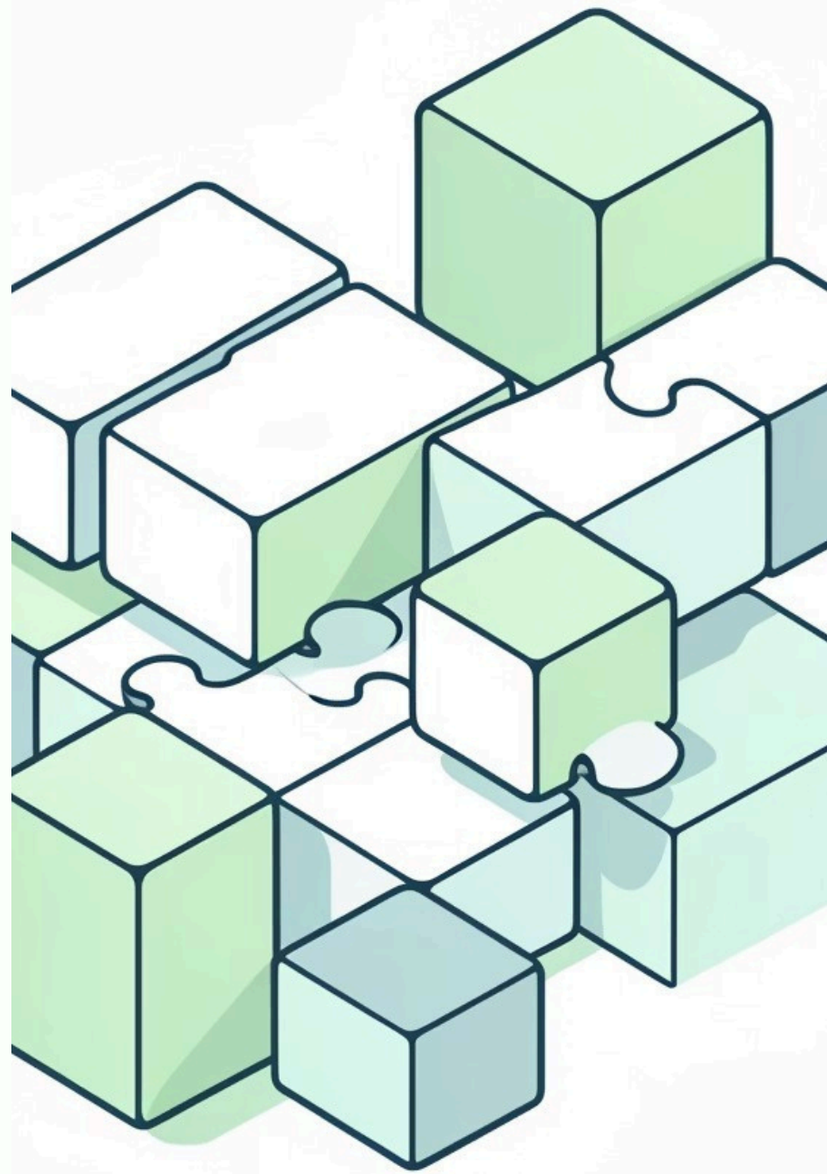
Apply the principle of least privilege to both code execution and infrastructure access, limiting potential damage.

Error Handling

Implemen robust error handling that provides useful information without leaking sensitive system details to attackers.

Automated Scans

Integrate automated security scans directly into CI/CD pipelines for continuous security assurance.



Documenting Findings & Remediation Steps

Comprehensive Reporting

- Detailed vulnerability descriptions , including impact and likelihood assessments.
- Clear risk ratings to prioritise critical issues.
- Code snippets highlighting the exact location of vulnerabilities.

Actionable Remediation

- Prioritised remediation plan with assigned owners and realistic timelines.
- Integration of security checkpoints into the software development lifecycle.
- Ongoing training sessions for developers on secure coding standards and best practices.

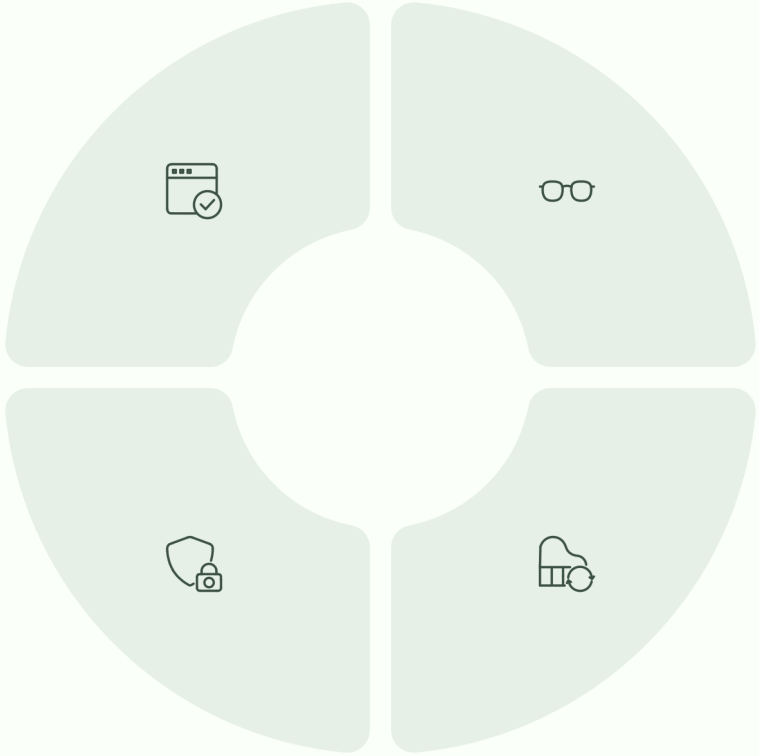
Conclusion: Building Security Into Every Line of Code

Essential Reviews

Regular secure code reviews are in dispensable for creating resilient and trustworthy applications.

Security-First

Commitment to a culture of security-first development is crucial for building safer software.



Hybrid Approach

A blend of automated tools and expert manual review uncovers a broader range of hidden security risks.

Mitigate Risks

Adhering to best practices significantly reduces the attack surface and enhances user protection.

THANK YOU

presented by: Poorna sri A