Mohamed Elhassan

CS 405 - Secure Coding

Project Two: Security Policy Presentation

4/20/2025

Southern New Hampshire University

<https://youtu.be/BlLUx_NEtMA>

| **Slide** | **Narrative** |
| --- | --- |
| **Slide 1: Title Page** | Hello everyone, and thank you for being here today. In this presentation, I'll be walking you through the Green Pace security policy guide and its implementation. My name is Mohamed Elhassan, and I’ll be discussing how we can ensure secure coding practices as we continue to grow as a development team. The goal is to integrate security into every phase of our development lifecycle. |
| **Slide 2: Overview** | The Green Pace security policy has been created to ensure that as our team grows, we maintain secure coding practices and robust system architectures. This policy was needed as our projects have scaled, making security more complex and crucial. It will help guide our team in adhering to best practices and will support our defense-in-depth strategy, where multiple layers of security are applied to protect our systems. |
| **Slide 3: Threats Matrix** | Here we have the **Threat Matrix**, which categorizes the security risks we’ve identified and ranks them by severity, likelihood, and impact. For example, **SQL Injection** is rated high due to its potential to compromise sensitive data, while **Buffer Overflow** and **Cross-Site Scripting** are also key concerns. By analyzing these risks, we can prioritize mitigation efforts effectively. |
| **Slide 4: 10 Principles** | These are the 10 **security principles** that guide our development practices. Each principle focuses on a critical aspect of secure coding. For instance, **Secure Input Validation** ensures that data is sanitized before being processed, while **Fail-Safe Defaults** ensures that the system operates securely by default. Adhering to these principles will help maintain a secure development process as we scale. |
| **Slide 5: Coding Standards** | Here are our **coding standards**, ranked in order of priority based on their potential risk. For example, **SQL Injection Prevention** is the highest priority because a vulnerability in this area could lead to severe data breaches. Other priorities like **Memory Protection** and **String Correctness** are critical for ensuring the stability and security of the system. |
| **Slide 6: Encryption Policies** | **Encryption** plays a critical role in securing our data. We have three primary encryption policies in place: **Encryption in Flight**, **Encryption at Rest**, and **Encryption in Use**. These policies ensure that our data remains secure at every stage—whether it’s being transmitted, stored, or processed, preventing unauthorized access and data theft. |
| **Slide 7: Triple-A Policies** | The **Triple-A Framework** focuses on Authentication, Authorization, and Accounting. **Authentication** ensures that only verified users can access our systems, while **Authorization** controls what users can do once they’re logged in. **Accounting** monitors user actions, helping us detect suspicious behavior and maintain a secure environment. |
| **Slide 8: Unit Testing – SQL Injection Prevention** | Now, let's look at **Unit Testing**, which ensures that critical components of the application are secure. The first test is focused on **SQL injection prevention**. This test checks whether the application properly sanitizes user inputs before using them in SQL queries. Positive Test: A safe input, like username=admin&password=1234, is entered and processed correctly. Negative Test: A malicious SQL injection attempt, such as username=admin' OR '1'='1&password=anything, is rejected to prevent unauthorized access. By running this test, we ensure that the application doesn't allow dangerous input to compromise the database. |
| **Slide 9: Unit Testing – Parameterized Queries** | The second test ensures the application uses **parameterized queries**, which is a key defense against SQL injection. Positive Test: When valid input is entered (e.g., username=admin&password=1234), the application should safely process it using parameterized queries. Negative Test: If malicious SQL injection code is entered, such as username=admin' OR '1'='1&password=anything, the system should reject it and prevent execution. This test guarantees that our system safely handles user input, preventing SQL injection attacks by avoiding string concatenation. |
| **Slide 10: Unit Testing – Logging Failed Login Attempts** | The third test verifies that the application logs failed login attempts, which helps in detecting potential security threats. Positive Test: After several invalid login attempts, including SQL injection attempts, the system should log each failure with appropriate details for monitoring. Negative Test: If the system fails to log any failed login attempt or if logging is inaccurate, it indicates a security oversight and failure to detect suspicious activity. This test ensures that we have visibility into failed authentication attempts, helping us track and respond to potential attacks. |
| **Slide 11: Unit Testing – Union-Based SQL Injection** | The fourth test focuses on detecting **union-based SQL injection** vulnerabilities, which allow attackers to retrieve unauthorized data by exploiting UNION SELECT statements. Positive Test: The system should sanitize inputs, blocking union-based injection attempts like username=admin UNION SELECT \* FROM users. Negative Test: If the system fails to block such an injection, it could potentially leak sensitive data. This test ensures that our system is secure against more advanced SQL injection techniques. |
| **Slide 12: Automation and DevSecOps** | In our **DevSecOps pipeline**, security tools like *SonarQube*, *CppCheck*, and *OWASP ZAP* will be integrated into various stages of the pipeline—during development, testing, and deployment. These tools will automatically detect vulnerabilities such as SQL injection and memory leaks, ensuring that security is continuously monitored and maintained throughout the development lifecycle. |
| **Slide 13: Risks and Benefits** | By addressing security vulnerabilities early in the development process, we minimize the risk of data breaches and system failures. **Acting now** ensures that we are proactive in securing our applications, while waiting can lead to costly fixes and a higher risk of exploitation. It’s essential to integrate security at every phase to protect our systems and data. |
| **Slide 14: Recommendations for Future Security Improvements** | While our current security policies cover the essentials, we recommend further integrating advanced tools and predictive analytics, like AI-based threat detection. Enhancing our encryption practices and improving real-time monitoring will also help us stay ahead of emerging threats. Continuous refinement of these practices is key to maintaining a secure environment. |
| **Slide 15: Conclusions** | In conclusion, securing our applications from the start is crucial for mitigating risks and protecting our data. The policies and practices outlined today are designed to ensure that we remain secure as our team and projects scale. Moving forward, we must continue to adapt our security strategy to meet new challenges, incorporating emerging technologies and maintaining vigilance against evolving threats. |