# Martin Garza

# CS-405- Project Two: Security Policy Presentation

# 4-16-25 YouTube Link: <https://www.youtube.com/watch?v=7sqWuWnyt-w>

# CS 405 Project Two Script Template

Complete this template by replacing the bracketed text with the relevant information.

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hello, my name is Martin Garza, and today I’ll be presenting the Green Pace Secure Development Policy.  In this presentation, I’ll walk you through our updated security standards, how we address vulnerabilities, how we use automation, and how we ensure compliance with defense-in-depth best practices as our team continues to grow. |
| **2** | The Green Pace Security Policy was created to make sure all development practices follow secure coding principles.  As our team grows, we needed a standardized approach to stay aligned with best practices.  This policy helps us identify and mitigate vulnerabilities early, keeping our efforts consistent and secure.  In terms of defense-in-depth, we focus on layering security measures at every stage of development — from coding to deployment. This reduces the chances of a successful breach, making sure vulnerabilities are caught before they can be exploited. |
| **3** | Now, let's look at the Threat Matrix and Automation Overview. We’ve categorized our vulnerabilities into four areas.  In the Likely category, we find major risks like SQL Injection, Unsafe String Functions, and Improper Input Validation — the vulnerabilities attackers exploit most often.  In the Priority category, we include Hardcoded Values, Deprecated Functions, and Poor Exception Handling — serious issues, but slightly less common.  For Low Priority, we have things like Lack of Explicit Data Types and Poor Use of Assertions — still important, but less likely to lead to a breach.  Finally, in the Unlikely category, we see Secure File Handling Issues and Memory Management Mistakes, which are rare when best practices are followed. To detect these vulnerabilities early, we use automation tools like Cppcheck and Visual Studio Static Analysis.  By integrating these into our DevSecOps pipeline — especially during coding and verification phases — we strengthen our defense-in-depth approach and ensure higher security compliance. |
| **4** | This slide shows how each security principle ties to specific coding standards.  For example, 'Validate Input Data' connects to input validation standards. 'Heed Compiler Warnings' is supported by using assertions.  Secure design is addressed through prepared statements to prevent SQL injection.  Keeping things simple is backed by using explicit data types.  Principles like 'Default Deny' and 'Least Privilege' align with avoiding hardcoded values and handling exceptions properly.  Defense-in-depth is reinforced by secure file handling and memory management standards.  This ensures that our principles aren’t just guidelines — they’re enforced through how we code every day. |
| **5** | Here, we show the 10 coding standards from our policy, ordered by threat severity and potential impact.  Prepared statements and validating input are ranked highest because they defend against the most critical vulnerabilities like SQL injection. Safe string functions and memory management come next, preventing common but dangerous issues like buffer overflows. Using modern functions and secure file handling methods keeps our systems updated and safe. Explicit data types and avoiding hardcoded values improve security and maintainability.  Finally, assertion use and proper exception handling round out the list — still important, but slightly lower risk. By addressing the most critical threats first, we create a stronger, more secure foundation. |
| **6** | Encryption is a critical part of our policy at Green Pace. **Encryption at rest** protects data stored on devices like hard drives and backup servers, even if those devices are stolen.  **Encryption in flight** protects data as it moves across networks, using secure protocols like TLS.  **Encryption in use** protects sensitive data that's actively being processed in memory — an often overlooked but important layer.  By encrypting data at every stage — rest, flight, and use — we ensure full lifecycle protection. |
| **7** | The Triple-A framework — Authentication, Authorization, and Accounting — is essential for secure operations.  **Authentication** verifies that users are who they claim to be, using strong credentials and multi-factor authentication.  **Authorization** makes sure users only access what they need, reinforcing least privilege principles.  **Accounting** tracks all user actions — logins, file access, database changes — for compliance, auditing, and investigation.  Together, these build a complete model for detecting, preventing, and responding to unauthorized activities. |
| **8** | For unit testing, we built 16 tests based on secure coding practices. Positive tests confirmed that behaviors like collection initialization, adding entries, resizing, and clearing collections worked correctly.  Negative tests included checking that exceptions are thrown for invalid access, and we even had an AlwaysFail test to make sure failures were detected.  Out of 16 tests, 15 passed successfully — showing that our code meets expectations and that vulnerabilities are caught early. |
| **9** | Automation is built into every stage of our DevSecOps pipeline.  In Assess and Plan, we model threats and train teams on security. In Design, we build security into development through practices like OWASP guidelines. During Build, we automate static code analysis tools like Cppcheck during the compile phase. In Verify and Test, we run vulnerability scans and security tests. Transition includes penetration testing and secure deployment practices. In Production, we Monitor and Detect using logs and SIEM systems, setting up automated intrusion alerts. Respond automation blocks threats immediately and restores systems if breaches happen. Finally, Maintain and Stabilize ensures systems are returned to a secure baseline to prevent recurring issues. |
| **10** | Automation is built into every stage of our DevSecOps pipeline.  In Assess and Plan, we model threats and train teams on security. In Design, we build security into development through practices like OWASP guidelines. During Build, we automate static code analysis tools like Cppcheck during the compile phase. In Verify and Test, we run vulnerability scans and security tests. Transition includes penetration testing and secure deployment practices. In Production, we Monitor and Detect using logs and SIEM systems, setting up automated intrusion alerts. Respond automation blocks threats immediately and restores systems if breaches happen. Finally, Maintain and Stabilize ensures systems are returned to a secure baseline to prevent recurring issues. |
| **11** | If Green Pace waits to act, we risk major security breaches. Legacy code remains vulnerable, input validation remains inconsistent, and security testing happens too late.  By adopting secure coding standards now, automating scans, and enforcing code reviews early, we build stronger systems, enhance user trust, and avoid higher breach costs down the road. |
| **12** | Despite strong policies, Green Pace still has critical gaps. Code reviews aren’t mandatory for every pull request. We don’t yet formally protect data in use. And many developers haven’t practiced breach response simulations.  We recommend integrating static analysis tools into every CI/CD pipeline, adding encryption for sensitive data in memory, and running regular security drills to close these gaps. |
| **13** | Looking ahead, Green Pace should adopt standards like the OWASP Top 10 to address common vulnerabilities.  Implementing Zero Trust Architecture will help remove implicit trust inside the system. We should move toward continuous security testing rather than relying on periodic reviews. Mandatory static and dynamic analysis tools before every release will raise our code quality dramatically.  And regular audits will ensure we stay ahead of evolving threats — just like companies who avoided major breaches by adapting early. |
| **14** | Here are the references I used to prepare this presentation, including OWASP, NIST guidelines on Zero Trust, Peerlyst’s Defense-in-Depth diagram, and SonarSource documentation for static code analysis.  Thank you for your time and attention. |