# CS 405 Project Two Script Template

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Good morning, everyone. I'm Anthony McCormack, and I'm here today to present the **Green Pace Security Policy Guide**. This document marks a critical organizational shift. Our goal is to take the implicit secure coding practices that our senior team members use every day and standardize them into a formal, repeatable policy for everyone. This ensures consistency as we grow and prepares us for our required security audit. |
| **2** | This entire policy is built on the philosophy of **Defense-in-Depth**. As you see in the illustration, we cannot rely on just one security layer. The policy implements controls from the low-level code up to the system architecture. Our **Coding Standards** act as the innermost defense, mitigating flaws at the source, while our **Triple-A Framework** and **Encryption** policies act as the outer barriers, preventing and containing breaches. We are now truly a DevSecOps organization. |
| **3** | To frame our work, we used this **Threats Matrix** to prioritize our identified vulnerabilities. The matrix combines **Likelihood** with **Severity** to assign a level. Our critical risks—Level 1—are grouped in the *Likely* and *Priority* quadrants. This includes **SQL Injection** and **Memory Protection** flaws. These are our highest concern because they lead directly to catastrophic failures like data theft or server compromise. |
| **4** | Our technical standards aren't arbitrary; they are rooted in 10 established security principles. For example, the principle of **Validate All Input** directly underpins three of our standards, including **SQL Injection** and **Data Value Validation**. This alignment shows that our technical rules are all supported by industry-accepted security theory, providing a robust philosophical foundation for the whole team. |
| **5** | Here is the complete list of our 10 coding standards. We prioritize them using the **Level** system from the Threat Matrix. Our **Level 1 Critical Risks**—which require immediate action—are **SQL Injection**, **Data Value Validation**, and **Memory Protection**. The lower-priority standards, like Exceptions and Assertions, are still important for secure failure, but they present a Medium (Level 3) risk and can be addressed after the critical flaws. |
| **6** | On the architectural side, we mandate encryption across three states of data. **Encryption At Rest** uses AES-256 to protect our backups and databases from physical theft. **Encryption In Flight** mandates the use of TLS 1.2 or higher for all communication, protecting our data from network eavesdropping. Finally, **Encryption In Use** requires us to mask or tokenize highly sensitive data when it's actively loaded into system memory to defend against advanced scraping attacks. |
| **7** | The **Triple-A Framework** forms a critical layer of access control. **Authentication** requires mandatory MFA for administrative access. **Authorization** uses **Role-Based Access Control (RBAC)** to enforce the principle of least privilege, ensuring developers and users can only access the files and features strictly necessary for their job. Lastly, **Accounting** mandates immutable audit logging for all critical events, including user logins and database changes, providing us with non-repudiation in an audit. |
| **8** | This is where we verify compliance. We use **Unit Testing** in the **Verify and test** phase of the pipeline as a security gate. Our tests are written to check two things: that the code works correctly (*positive tests*) and that the code *fails securely* when given bad data (*negative tests*). This screenshot shows the structure of our Google Test framework and the assertions we use.  This console output proves our system works. The test **AtThrowsExceptionForOutOfBounds** is a **negative test** enforcing the **Data Value** standard. Because it passes ([ OK ]), we know our code correctly throws a std::out\_of\_range exception when given an invalid index, preventing the application from crashing or exposing uninitialized memory. The PopBack test verifies that our removal logic works perfectly under normal conditions. |
| **9** | This diagram shows how we automate enforcement. In the **Build** phase, **Static Analysis** (SAST) tools like SonarQube scan every line of code against our 10 standards. If a Level 1 or 2 vulnerability is found, the **SEC** gear triggers a **failure gate**, preventing deployment. In the **Verify and test** phase, **Runtime Analysis** tools like ASAN and OWASP ZAP actively check the compiled application for memory and injection flaws, ensuring no vulnerability slips through. |
| **10** | We must address the problems now, as there are significant **Risks of Waiting**. If we wait, we maintain Level 1 risks that can lead to millions in financial losses, as seen in breaches like Yahoo! By **acting now** and automating enforcement, the cost of fixing these flaws is dramatically reduced, as it's caught at the cheapest point: compilation. This immediate action provides us with a high return on our security investment. |
| **11** | Looking ahead, we must address **current policy gaps**. Our policy currently lacks standards for secure **Configuration Management** and **Third-Party Dependency Scanning**. For example, we don't have mandatory technical rules to ensure our cloud resources are configured with the strongest possible security settings. |
| **12** | To prevent future problems, I recommend three key standards. We should adopt **IaC Security**, scanning our infrastructure code with tools like Ansible to enforce secure server configurations. We must also mandate **Supply Chain Standards** and use tools like Snyk to automatically check open-source packages for vulnerabilities, eliminating future threats like the Log4j incident. |
| **13** | This Green Pace Security Policy represents a complete commitment to DevSecOps. By acting on our Level 1 priorities and enforcing these standards automatically, we ensure compliance, protect customer data, and achieve a robust defense-in-depth posture, moving ahead of the threat landscape. |
| **14** | Thank you |