# 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 Mohamed Aziz Zaghdoudi, and this is my security policy presentation for Green Pace. In this presentation, I’ll Walk through our key standards, tools, and testing strategies to help keep our systems secure as our development team grows. |
| **2** | This security policy helps us turn our secure coding habits into clear standards. As we grow, it’s important that everyone follows the same practices. The policy supports defense-in-depth by building security into every step from writing code to deploying it. |
| **3** | This matrix shows the most critical risks. Data validation, unsafe strings, and memory handling are likely and high-risk, so they’re our top priorities. Safer logging and smart pointers are lower risk. Some rare issues, like misuse of assertions or exceptions, are less likely but still tracked. |
| **4** | These are the 10 core principles that guide our secure coding. Each one links directly to specific standards we’ve implemented. For example, validating input data is supported by our standards for data types, input values, and strings. We also apply principles like least privilege, fail-safe defaults, and defense in depth through memory protection, file handling, and error logging. This mapping keeps our development focused and consistent. |
| **5** | These are our ten coding standards ranked by risk. I focused on how likely each issue is to occur, how severe it could be, and how easily it could be exploited. Unsafe strings and SQL injection top the list, while things like logging and assertion misuse are lower. This priority system helps us focus on fixing the most dangerous issues first. |
| **6** | Our encryption policy protects data at every stage. In flight, we use TLS to secure all network communication. At rest, we encrypt databases and storage using strong ciphers like AES-256. While in use, we minimize the time decrypted data stays in memory and clear it when it’s no longer needed. This three-layer approach keeps sensitive information secure throughout its lifecycle. |
| **7** | The Triple-A model is our foundation for access control. First, authentication confirms user identity using secure passwords and multi-factor methods. Next, authorization ensures users only access what they’re allowed through role-based controls. Finally, accounting tracks all user activity logins, data access, and changes so we can audit and detect misuse. |
| **8** | These unit tests help us catch common security problems early. We test both valid and invalid input to make sure validation works. For strings, we check that buffers aren’t overrun. Memory use is verified using tools like Valgrind to catch leaks. All of this reinforces secure practices during development. |
| **9** | DevSecOps means building security into every step of development, not just at the end. |
| **10** | So, at Green pace We use tools like Cppcheck and Clang-Tidy during coding, GoogleTest and Valgrind for testing, and SonarQube for integration. Before release, we scan with Fortify or Coverity, then monitor activity with secure logs. This way, security is continuous and automated. |
| **11** | Waiting to fix issues increases the chance of attacks and system failures. Acting now prevents damage, saves cost, and improves stability. But our current strategy needs stronger exception handling, better logging, and access reviews. Automation helps, but we also need manual checks and better training to cover blind spots. |
| **12** | Our policy handles critical threats well, but there are still gaps. We need better exception handling, stronger rules for file access and logs, and more secure coding training. We also lack runtime monitoring to catch threats in real time. Closing these gaps will improve our overall defense. |
| **13** | To prevent future issues, we should adopt SEI CERT C++ and MISRA standards. They provide strict rules for writing safe, reliable code, especially in large or critical systems. Adding these to our policy will reduce long-term risks and help us maintain strong security as we grow. |
| **14** | These references include the secure coding standards and tools used in this project, such as SEI CERT, MISRA, OWASP, and analysis tools like Cppcheck, Clang-Tidy, and SonarQube. All sources are cited in APA format for accuracy and transparency. |