# CS 405 Project Two Script

Ally Miller

Video: <https://youtu.be/LRuR_xnVcdw>

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hello, my name is Ally Miller, and this presentation will cover the Green Pace Security Policy. |
| **2** | The Green Pace Security Development Policy was created to address the risk associated with software development. As cyber threats evolve, maintaining a proactive security reaction throughout the software development lifecycle is essential. This policy supports the best practice of Defense in Depth which uses multiple layers of security controls. By using these layers of security Green Pace can ensure that if one layer fails the other controls will protect the system. |
| **3** | This matrix presents a threat analysis based on the likelihood of each risk occurring. In the likely category, avoiding using magic numbers is important because unnamed constants can introduce logic errors and make it harder to maintain. The use of prepared statements is important, as failing to implement them properly can leave applications vulnerable to an SQL injection attack. Accessing memory outside the lifetime of an object is another likely risk, which can result in crashes. Declaring static objects in block scope is recommended to reduce the risk of unintended data manipulation. And lastly, avoiding the use of namespace std; in header files helps prevent naming conflicts that can introduce hidden bugs. In the priority category, validating input data types is critical to protect against injection attacks. Ensuring that strings have enough space for a null terminator is also essential, as neglecting this can lead to buffer overflows. In the low priority category, the use of assert() for runtime error checking is discouraged, as assertions may be disabled in production environments. Finally, in the unlikely category, using exceptions for error handling presents a low risk when implemented correctly. |
| **4** | This slide highlights the 10 core security principles and how they are reinforced by the coding standards defined in the policy. The principle of validating input data is supported by standards STD-001 and STD-003, ensuring that all input is properly checked for type and format to prevent unsafe actions. Heeding compiler warnings is enforced through STD-006-CPP, which replaces the use of assert() with proper runtime checks. Keeping it simple is supported by STD-002 through the elimination of magic numbers, STD-007 and STD-008 by promoting clean error handling and scoped variable use, and STD-010 by preventing namespace conflicts. Sanitizing data sent to other systems is supported by STD-004, which uses prepared SQL statements to prevent injection attacks. Practicing defense in depth is implemented through STD-003, STD-005, and STD-009, which together enforce memory safety, proper buffer usage, and safe pointer initialization. Lastly, the principle of using effective quality assurance techniques is supported by STD-005 and STD-007, which promote safe memory management and exception handling. |
| **5** | These are the ten coding standards defined in the security policy. Each standard was assessed for severity, likelihood, remediation cost, and priority level. Severity is the potential impact of vulnerability if exploited. Likelihood is how common it may occur in a real-world scenario. Remediation cost reflected the effort and resources required to implement a fix. The priority level represented the overall risk. Standards STD-003 and STD-004 received the highest risk rating due to both high severity and high likelihood. STD-002 and STD-010 were also rated high because they affect code maintainability which can introduce bugs. On the other hand, STD-006 received a lower risk rating, as replacing assert() statements with proper error handling is a relatively simple and low-cost fix. |
| **6** | Encryption has three layers: at rest, in flight, and in use. Encryption at rest secures data stored on physical devices, like a hard drive, and uses strong encryption algorithms to protect sensitive information like passwords from unauthorized access. Encryption in flight protects data as it is transmitted across networks, ensuring confidentiality during communication. And lastly, Encryption in use secures data while it is actively being processed in memory. |
| **7** | The triple a framework is a framework that allows for identity verification via authentication, controlled access via authorization, and accountability via accounting. Authentication refers to the verification of the identity of a person trying to access a system. Using Role Based Access Controls we can prevent unauthorized access. Authorization refers to the actions a user’s is authorized to perform. A regular user should not have admin privileges, and any admin actions should be authorized or denied based on roles by the system. And accounting refers to tracking activities and logs of users. All sessions should be logged for review for auditing and compliance reasons. This way any changes to the system will be tracked. |
| **8** | Unit testing is an important process to ensure that the code is operating as intended. Let’s look at an example for one of our controls, input data validation. If we create a test that has the exception that only valid integer input is accepted, then we need to check for a few things. Frist does the program accept a valid integer, which it is supposed to so the expected result would be yes it does. Secondly, does the program reject a string input, which the expected result would be an error message because a string is not an accepted input for this test. Third, what happens when a user puts in a floating-point number? It should be rejected due to a mismatched type. Fourth, does the input validation handle no input, or an empty string? It should be rejected in this case. And lately, can the program re-prompt the user after the user uses an invalid input? The expected result would be that it does and accepts the next valid input. |
| **9** | This slide shows how automation supports security. In the pre-production phase, automation is applied to threat assessment, secure design, code builds, and vulnerability testing which helps catch issues early. During production, automation is used to configure security settings, run health checks, and continuously monitor systems using tools like SIEM. Integrating security into each phase through automation gives us continuous protection and reduces the risk of human error. |
| **10** | This security policy uses automated tools like CppCheck and SonarQube to identify vulnerabilities. CppCheck is an analysis tool that can detect common C++ issues like buffer overflows, uninitialized variables, and unsafe pointers. It helps with compliance by flagging any issues early in the development process. SonarQube is a deeper code analysis tool that allows us to remediate risks more effectively as well. With these tools we can reduce manual review and enhance the security of the code base. |
| **11** | By implementing this security policy there are a lot of benefits and a few risks to consider. The benefits include increased resilience, and less chance of attackers breaking into our system or bugs causing headaches that can lead to a reduction in incidents. By not implementing this security policy we can get exploited easier, which may lead to higher costs and a lack of government compliance. One risk I want to mention here is the risk of resistance to this change. Employees will need to be informed and trained on the new security policy heavily as some controls may change how their work is affected. |
| **12** | While this security policy helps provide good coverage for security controls there are a few things we can add to cover the gaps in the policy. This policy currently does not address third party dependency management. We should be scanning for vulnerabilities for third party applications. Additionally, we should include employee training and awareness for secure coding to reduce human error. |
| **13** | Additional standards should be considered such as secure dependency management practices. We should incorporate scanning third party libraries. Additionally expanding our controls to other programming languages would be something to consider in the future. |
| **14** | Thank you for your time! |