# CS 405 Project Two Script Template

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hi, my name is Ryan. Today, I will be presenting our comprehensive security policy with a focus on implementing a defense-in-depth strategy. This presentation will cover the core security principles we adhere to, our coding standards, encryption policies, Triple-A policies, unit testing, the DevSecOps pipeline, and the tools we use to maintain security throughout the development lifecycle. Let's get started by discussing the importance of a defense-in-depth approach to security. |
| **2** | Defense in depth is a strategy where multiple layers of security controls are placed throughout a system or application. This approach mitigates risks associated with single points of failure and enhances overall system, ensuring that if one security measure fails, others are in place to maintain security. |
| **3** | Next, we have our threats matrix. This matrix categorizes potential threats based on their likelihood and severity. Likely threats, such as ERR-051-CPP and FIO-008-CPP, have a higher chance of occurring but lower severity. Priority threats, including STR-002-CPP and MEM-056-CPP, have a good chance of happening with high severity. Low priority threats, like DCL-004-CPP, are less likely and less severe. Unlikely threats, such as ERR-006-CPP, are rare but could have a high impact. |
| **4** | Let's discuss the ten core security principles that guide our security policy. These principles include validating input data, heeding compiler warnings, architecting and designing for security policies, keeping it simple, default deny, adhering to the principle of least privilege, sanitizing data sent to other systems, practicing defense in depth, using effective quality assurance techniques, and adopting a secure coding standard. Each principle is supported by specific coding standards to ensure comprehensive security. |
| **5** | Here are our coding standards aligned with the core security principles. For example, STR-002-CPP focuses on sanitizing data passed to complex subsystems like SQL, ensuring input validation and defense in depth. MEM-056-CPP advises against storing already-owned pointer values in unrelated smart pointers to prevent memory leaks, supporting the principles of keeping it simple and heeding compiler warnings. |
| **6** | Our encryption policies cover three key areas: encryption at rest, encryption in flight, and encryption in use. Encryption at rest protects data stored on physical media from unauthorized access. Encryption in flight secures data during transmission to prevent interception. Encryption in use ensures data remains secure even when being processed or used in memory. |
| **7** | The Triple-A policies consist of Authentication, Authorization, and Accounting. Authentication verifies identities before granting access. Authorization defines what authenticated users can do within the system. Accounting tracks and records user activities for monitoring and compliance, ensuring that all actions are logged and reviewed. |
| **8** | Here are some of the unit tests that ensure compliance with our coding standards and security policies. This shows a test to see if an empty vector can have items added to it. The test ran and passed. |
| **9** | This test looked at if a vector could be increased after being set. It also ran and passed its test. |
| **10** | Here is a test to see if an exception would be thrown if a request is out of range. This test also ran and passed. |
| **11** | Lastly, this test looked to see if the program would handle accessing an element at a specific index. This both ran and passed the test. |
| **12** | Automation in the DevSecOps pipeline helps maintain security standards by using security testing tools, automating vulnerability scans, and ensuring consistent enforcement of security policies throughout the development lifecycle. |
| **13** | Tools like SonarQube, PVS-Studio, and Clang-Tidy play crucial roles in our DevSecOps pipeline. SonarQube provides continuous inspection and static analysis, identifying code quality issues and vulnerabilities. PVS-Studio focuses on detecting bugs and security vulnerabilities during coding and testing. Clang-Tidy offers static analysis and quality checks during coding, build, and testing stages, ensuring adherence to standards. |
| **14** | Implementing security now versus waiting has significant implications. Acting now enhances our security, saves costs in the long run, and ensures compliance. However, it requires upfront investment and a learning curve. Waiting increases the risk of vulnerabilities, higher costs, and potential compliance issues. Immediate action is crucial to prevent data breaches and maintain robust security. |
| **15** | I recommend reviewing and updating security policies regularly, providing continuous training and awareness programs, and fostering a culture of security. Additionally, conducting regular audits and compliance checks ensures adherence to security policies. A security policy is only effective if every developer follows it, ensuring consistency and robust security practices. |
| **16** | In conclusion, security is just as important as the finished application and must be considered from the start. Sticking to a security policy holds all developers to the same requirements, ensuring consistency. Regular updates, training, and a unified approach to security foster collaboration and accountability within the team, leading to a robust and resilient application. |