# CS 405 Project Two Script Template

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

[**https://youtu.be/ZLVYrSyPGao**](https://youtu.be/ZLVYrSyPGao)

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | My name is Chandler Duhamel, and today I will be presenting my security policy proposal for Green Pace.  In this presentation I'm going to take you through all the security standards, testing methods and policies that will keep our development team secure as we grow. |
| **2** | To begin, Green Pace has developed a security policy based on the principles of defence-in-depth. This means that we will not place all our security assurance in one place, but will use multiple layers of security throughout the development process. The intention is to ensure that our systems are secure, even if one of the security measures fails. |
| **3** | Using likelihood and severity matrices we categorized risks and ranked the top safety threats in order of likelihood and severity. With SonarQube and Clang-Tidy we can identify dangers early to help eliminate errors before they are exploited in production. With automated detection tools we can "stop it in the code" long before it is ever deployed. |
| **4** | Here are ten fundamental security principles upon which our policy rests. Each principle maps accordingly to certain coding standards. For example, validating input includes standards such as EXP34-CPP and FIO50-CPP. This provides us with a level of assurance that our standards do indeed promote sound security practice. |
| **5** | This slide provides the specific ten coding standards we are using; they cover a lot of ground from SQL injection to memory leaks. I ranked them in importance, starting with SQL injection and unsafe file paths, which are the greatest risk to us. |
| **6** | Encryption is an essential element of our security strategy. We use AES-256 to protect data at rest, TLS 1.2 or greater to protect data in flight, and strong memory handling practices to protect data in use. Using encryption enables us to ensure sensitive data is always protected regardless of where that data resides. |
| **7** | Triple-A means Authentication, Authorization, and Accounting. We require users to login with unique credentials. Permissions are granted based on least privilege, so users only see what they need. And we log every user action, which can help us track and audit what's going on. |
| **8** | The first unit test checks to see if the vector contains nothing upon initialization. This is an important check in case there's uninitialized data in the vector, which can lead to unexpected behavior. This is a simple, but crucial, test to show that our initialization is sound. |
| **9** | Next, I tested what happens when you attempt to access an index outside of a vector's bounds. Not only is this test important due to the fact that it shows our code raises exceptions instead of crashing, but it also relates directly to EXP34-CPP because it is a complete example of validating input. |
| **10** | I also tested the functionality of the resize operation of a vector working both up and down, and even to zero. These tests ensure that our program manages memory safely while avoiding leaks and errors. |
| **11** | Next I tested how reserve behaved. Reserve should increase capacity without changing the size, and these tests help verify that the system can't shrink capacity in a way that defeats the logic behind reserve. This is a subtle test, but one that is important for memory safety. |
| **12** | Another test I executed verified that calling reserve with a smaller number doesn’t change the capacity. This prevents undesirable behaviors and upholds safe memory management in our applications. |
| **13** | All of these tests were written using GoogleTest, which automatically runs as part of our DevSecOps processes. Each of our tests improves the likelihood that we will find problems sooner, mitigate the effect of crashes, and that our software behaves expectantly. |

|  |  |
| --- | --- |
| **14** | At Green Pace, we implemented a DevSecOps pipeline to identify security issues during development and not just at the end of the process. Developers utilize SonarQube, Clang-Tidy, and Cppcheck active during coding. During the build and deployments we scan with Coverity and execute GoogleTests. After deployment we also look at things like the firewall logs and other system activity. These multiple layers of automation allow us to respond quickly and keep our systems secure. |

|  |  |
| --- | --- |
| **15** | One of the most significant risks we face is the existence of legacy code that is not following our up-to-date security standards. For example, code that utilizes strcpy library or code that inputs SQL without validation exposes us to risk. If we clean these things up now and replace them with safer alternatives, we will eliminate vulnerabilities and reduce the chances of a breach. Cleaning it up now builds comfort in our software and reduces future technical debt. The longer we wait, the greater the risk. |

|  |  |
| --- | --- |
| **16** | Our existing policy is strong and we are doing some good things, but there are gaps. We don't have real-time threat detection or much training for developers. Adding ThreatMapper or SecureFlag, and incorporating mobile and cloud security to our day-to-day security standards, will help us mitigate threats and reinforce the security of our apps. |
| **17** | In conclusion, Green Pace should incorporate more secure coding standards such as OWASP Top Ten and continuously track the SEI CERT C++ rules. We should also periodically update our automation tools and foster security as a shared responsibility by the entire team. This will allow us to continuously prepare for what's next and continue delivering secure and reliable software. |