# CS 405 Project Two Script Template

Devin Wheeler

6/29/2015

Project Two: Security Policy Presentation

<https://www.youtube.com/watch?v=XstFS2i26ew>

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hello everyone, my name is Devin Wheeler. Today, I’ll be presenting the Green Pace Secure Development Policy. This presentation covers our secure coding standards, core security principles, encryption and Triple-A policies, and suggestions for risk mitigation with automation. |
| **2** | As Green Pace continues to grow, it is very important to make sure everyone, new and old, follows a consistent set of secure coding practices. These policies define a very clear framework in which we can protect our systems and data by adding security into every step of the Software development lifecycle. |
| **3** | This threat matrix produces a great way to view the likely hood and Priority to implement each of the policies. Standards that try and fix vulnerabilities like buffer overflows, SQL injections, and memory issues should be of the highest priority because they expose the most risk. An example would be STD-004-CPP using Prepared statements as this is a great way to help protect against SQL Injections. Standards like Don’t modify Const should be lower priority as they are very unlikely to happen, however they should still be implemented in case they ever do. |
| **4** | Our 10 core security principles should drive all of our decisions. From validating inputs to adopting a secure coding standard. Each principal helps to prevent security risks and helps make sure there is consistency throughout the codebase.   1. We should treat any and all input data as untrusted. We should check its length, type, and format to prevent attacks. 2. We should not ignore Compiler warnings. We need to enable the strictest warnings in order to try and catch issues early on. 3. Make sure security is thought about in all steps of the Security policies and not just when needed. 4. Keep the code clear and maintainable. Keeping it simple makes it easier to test, audit, and understand. 5. Design systems to deny access by default and only grant permission to the users that need it. 6. Ensure users and processes run with the minimum required access to accomplish what they need. This lowers the risk of using access to something they shouldn’t. 7. Clean and validate all data leaving the system to ensure no injections occurred. 8. Layer multiple security controls and systems on top of each other so that if one fails the second has a chance to stop it. 9. Include security focused testing, code reviews, and automation scans to detect and fix vulnerabilities early on. 10. Follow an established coding standard like SEI Cert C++. |
| **5** | On this slide I have arranged our 10 coding standards using 4 main criteria: Severity, Likelihood, Remediation cost, and Priority level. The highest ranked standards like using prepared statements, proper string termination, and correct memory management, directly prevent the most critical and likely exploits to occur like SQL injection and buffer overflows. Standards lower on the list, like using nullptr or avoiding const modification, are still important to prevent overall security and maintainability but have less severs problems if not followed and are a lot less likely to occur. This ranking can help us focus on what our priority should be for reducing the most risks. |
| **6** | All data is encrypted when in rest using AES-256 encryption. When data is being transmitted it uses TLS (Transport Layer Security). When data is in use the expose in memory needs to minimized and the buffers should be cleared as soon as possible. |
| **7** | Our Triple-A framework ensures users are verified, can do what they need, and all their activities are logged. To help with this we require Strong password and MFA for Authentication. We follow a least privilege model for Authorization. And for accounting we track logins, what was accessed, and changes made by a user. |
| **8** | 8.1 This negative test checks that accessing an out of range index throws and std::out\_of\_range error instead of causing unexpected behavior or memory corruption. The reason this is important is to help prevent buffer overflows and enforce safe boundary checking.  8.2 This is a positive test that checks to make sure that after calling clear(), the collection is empty and has a size of 0. The reason we check is to ensure memory is getting cleaned up properly, to prevent leaks or stale data.  8.3 This is another positive test that makes sure a new collection starts initialized, empty, and has a size of 0. We do this to make sure no uninitialized memory is being used on its creation.  8.4 This negative test checks to make sure that accessing the collection with a negative number will again through a std::out\_of\_range error. We do this because it validates safe error handling on invalid indices, helping prevent crashes or corruption to the data. |
| **12** | This Diagram shows how security automation is embedded in every stage of our DevSecOps workflow.  In preproduction we start by assessing and planning, this is where we select static analysis tools like Cppcheck and Clang-Tidy to identify coding vulnerabilities early on. During the design phase, we apply secure design patterns and enforce linting rules based on a standard.  When we build our compilers, we will be set to treat all warnings as errors, which should prevent unsafe code from moving on. We can also scan any dependencies we use to catch known vulnerabilities if using open-source code.  In the verify and test step, unit tests run in a CI pipeline to check we are following secure coding standards. We may also run scanning tools to check for any misconfiguration or know issues before release.  Before moving onto production, validation and penetration testing are performed to make sure security is working and meets our policies.  One in production our SIEM system and intrusion detection tools will continuously monitor for unusual or suspicious activity. If something is detected automated warning or workflows can help to stop attacks, turn of services, or roll back to a secure state.  This layered automation strategy strengthens our defenses , helps reduce human errors, and makes sure security is though about in every step of the process. |
| **13** | During pre-production, we can use Cppcheck and Clang-tidy to check for security issues in our code, then use Google test to run automated unit tests.  As we build, we will have the compiler set to treat warning as errors, stopping any unsafe coding practices from moving on. We also run dependency checks to make sure libraries are secure  In verification and testing, we use the same static analysis tools as before, Cppcheck and Clang-tidy, to confirm we are following our coding standards and vulnerability scanners like Nessus to help find know issues.  Lastly in production, SIEM systems and intrusion detection tools, like Splunk and Snort, will watch for attacks in real time and logs are collected for audits and alerting. |
| **14** | Implementing a secure coding practice offers clear benefits: It reduces the chances of security incidents, improves the overall code quality, and helps build trust with users.  However, there are a few risks. Vulnerabilities might not get fully addressed; there is an upfront cost in both time and money. Additionally, adding security measures make the code more complex and requires more testing.  However, in most cases it is far better to implement the secure coding practices as early as possible. |
| **15** | On this slide I will be going over some of the current security policy gaps that we were able to recognize.  First, we don’t have strong controls around 3rd party libraries or dependencies, which can have known vulnerabilities. Not all of our developers have the same level or consistency when it comes to secure code training, and our incident response plan hasn’t been tested in practice.  Some of our security checks still heavily rely on manual checks rather than automated tools, which can lead to human error. We also do not have a way to enforce or determine if these policies are being followed as they should. |
| **16** | To address the gaps from the previous slide I recommend adding a dependency scanning tool, requiring secure code training, scheduling incident response exercises a few times a year, and updating our policies regularly.  A great example of why these improvements is necessary would be the Pegasus hack. Attackers were able to take over devices by exploiting unpatched systems due to weak security controls and not well tested updates. This shows why policies need to be both up to date and enforced effectively. |
| **17** | [Insert text.] |