# CS 405 Project Two Script

Taryn Brownfield

8/18/24

<https://www.youtube.com/watch?v=WCopFWfXj10>

| **Slide Number** | **Narrative** |
| --- | --- |
| **1**  **Title Slide** | Hi. Welcome to the Green Pace Security Policy Presentation. I'm Taryn. |
| **2**  **Overview: Defense in Depth** | First, let's discuss defense in depth. This is the practice of layering distinct security measures over top one another. The layers are often redundant or overlapping, but together provide a strong defense against attacks. The idea is that if an attacker manages to breach one layer through a certain vulnerability, the next layer will not have that vulnerability so the attacker will be stopped or slowed until the next layer can be breached. This hopefully gives the team time to detect and respond to a security breach of one layer before any data has actually been compromised. To accomplish this layered defense mechanism, it is important to have software security built-in throughout the code. Each coding standard and policy discussed in this presentation could be considered one layer of this defense. |
| **3**  **Threats Matrix** | Here we have the threats matrix. It compares the severity of a security vulnerability (either High or Low) to the likelihood of the vulnerability being encountered (either unlikely, likely, or probable). The highest priority vulnerabilities are those that have a high likelihood and high consequence.  Automation tools are excellent for detecting threats, and even assessing how severe the vulnerability is. For instance, Cppcheck is a static analysis tool that examines a C++ source file and reports on found vulnerabilities as well as the severity of the threat if left uncorrected. |
| **4**  **10 Principles** | The security policy defines 10 security principles, and supports many of them with relevant coding standards. The 10 principles are:   1. Validate input data 2. Heed compiler warnings 3. Architect and design for security policies 4. Keep it simple 5. Default deny 6. Adhere to the principle of least privilege 7. Sanitize data sent to other systems 8. Practice defense in depth 9. Use effective quality assurance techniques 10. And finally, adopt a secure coding standard. |
| **5**  **Coding Standards** | Here are the 10 coding standards, which deal with several common vulnerabilities in C++ code. Each of these coding standards is relevant to one or more of the 10 principles. The standards have been evaluated for the severity of the threat posed by the vulnerability, and ordered by highest priority threat to lowest based on the threats matrix presented earlier. |
| **6**  **Encryption Policies** | Here we have the encryption policy standards for data at rest, data in flight, and data in use. Data at rest refers to data currently in storage, such as saved files or data in a database. Data in flight refers to any data being transmitted, such as via email or HTTPS. Finally, data in use is data actively being used, whether it's being read, created, modified, or deleted.  Unencrypted data is vulnerable to attackers. Encrypting data ensures that, even if an attacker breaches the security of the system and gets ahold of the data, they will not be able to read it. Data at rest should be securely encrypted with a symmetric key (which should be kept secure and stored separately from the data). Data in flight should be encrypted by the sender using a public key and decrypted by the receiver using a private key. Hash encryption and security certificates can also be used to authenticate sender and receiver, as well as verify that data was not modified or corrupted in transit. Lastly, data in use can be encrypted and decrypted in real-time by using a trusted execution environment, which constantly encrypts and decrypts the data as it is used. An attacker outside the environment will only be able to see the encrypted data. |
| **7**  **Triple-A Policies** | Authentication, authorization, and accounting, or Triple-A policies, together form a security framework for managing computer access and resources.  Authentication deals with identifying an entity, such as an account with a username or user ID, and verifying that the entity is who they say they are, often with a password or PIN.  Authorization deals with what permissions that account has, or in other words, what data the account is allowed to access and what it is allowed to do to it. This can be easily accomplished using role-based access controls, which define a type of user such as a client or administrator, and then give the same permissions to everyone with that role. Role-based access controls work best when also employing the principle of least privilege, which limits the privileges that an entity has to only those necessary for its tasks. For instance, a client may be able to access only their own account data, and may be able to modify some data (such as a current address) but not other data (such as their user ID or birthday). Whereas an admin may be able to access any client data and modify any data relating to that client.  Finally, accounting deals with the recording of system activity. Logging things like who accessed a system, when they accessed it, and what resources they accessed allows the system administrators to monitor activity within a system and investigate if suspicious activity occurs. In the event of a software security breach, the system activity can be examined for how and when the breach occurred. |
| **8**  **Unit Testing** | Unit testing is a way to test and verify that code is working as expected. Here, I have written a few simple unit tests to demonstrate a few of the coding standards presented earlier. The screenshot here depicts the console output of my tests, and shows that they all passed successfully. |
| **9**  **Unit Testing**  **Static Assertions & ConstantExpressionAssertion** | First up, we have a somewhat unusual unit test that is meant to demonstrate the differences between using a test in a test suite vs. a static assertion. The relevant coding standard here is, "Use a static assertion to test the value of a constant expression." To demonstrate, I actually created a regular unit test as well as two static assertions, one that succeeds and one that fails, which you can see by the red underline. Right away, the benefits of static assertions are evident: you don't even have to run the code to determine whether a static assertion passes or fails. The IDE (Visual Studio, in this case) will tell you. |
| **10**  **Unit Testing**  **CanThrowAndCatchExceptions** | Next, the coding standard being examined is, "Handle all exceptions." Here I have created a class with two functions, one that simply throws an exception, and the other which calls the throwing function within a try-catch block, and catches the exception. I then assert that the throwing function does indeed throw the exception, and the catching function does indeed catch the exception (so no exception is thrown). |
| **11**  **Unit Testing**  **InitializeVariables** | Next, the coding standard being examined is, "Do not read uninitialized memory." C++ allows you to declare a variable without actually assigning that variable a value. It also allows you to read the value of that variable. This can cause undefined behavior, because that variable is essentially storing garbage data. Here, I have a variable x that is declared as an optional integer. When x has not yet been initialized, it will evaluate to false, and reading the value of this variable would result in garbage data. Once initialized, it evaluates to true and the value is now valid to read. |
| **12**  **Unit Testing**  **CanAccessValidContainerIndices** | Finally, the coding standard being examined is, "Guarantee that container indices and iterators are within the valid range." Here, a container is created and values within the container are initialized. But similarly to the previous example, C++ will let a user access invalid indices from the container, which results in garbage data and can cause memory leaks. The test verifies that the data is valid for all of the valid indices, but that data has not been initialized for indices beyond the size of the container. |
| **13**  **Automation Summary** | Automation is an important and useful addition to the DevSecOps pipeline, which is essentially the same as the DevOps pipeline but with security as an integral part of the whole process. In essence, DevOps is a combination of development and operations. A DevOps team is therefore involved in the design, development, release, and maintenance of a software system. However, this philosophy often treats security as a separate concern, often tacked-on toward the end of development and handled by a separate team. DevSecOps is a different philosophy that considers security to be integral to the process, and the responsibility of every developer. This proactive approach to security means that the software is designed and built with security concerns in mind right from the start, and that security is not left until the end of a project, which may cause huge delays in schedule and necessitate additional cost if the code needs significant overhauling to come in line with security.  Automation tools are therefore an important part of the DevSecOps developer's toolbox. Automation can be implemented virtually throughout the software development lifecycle, allowing security vulnerabilities to be addressed as code is written, rather than leaving security to the end of development. |
| **14**  **Tools** | There are lots of different automation tools, but a few key kinds are SAST and DAST tools, as well as IDES. Static application security testing tools, such as Cppcheck, examine the source code for vulnerabilities. These tools can be used right from the start of a project, as soon as any code is written. On the other hand, dynamic application security testing tools analyze running software. This means that a functional application is required for these tools to be used, so they won't be useful until later in development, but they can find run-time and environmental vulnerabilities that SAST tools will miss. Finally, even an IDE can be a useful tool for automatic security checking. IDEs provide compiler warnings, as well as other useful security features that can aid developers. |
| **15**  **Risks and Benefits** | When bringing existing code in line with security standards, it is important to consider the risks vs. the rewards, and when to act. In dealing with security vulnerabilities, mitigating them as soon as possible is always preferable. But there are scheduling and financial realities to consider. The security vulnerabilities with the highest priority (high likelihoods and high consequences) should be handled immediately, but it may be necessary to put off some of the lower priority items for the time being.  Going forward, it is important to adopt a culture that emphasizes security as an integral piece of the puzzle. Developers should each consider themselves responsible for writing secure code, rather than leaving security to the end or to someone else. Developers should be familiar with and adhere strictly to the security policy. The system should be kept up to date with the latest patches, and audits should be performed routinely to verify that the system is up to standards. |
| **16**  **Recommendations** | This security policy is a great start to bringing Green Pace in line with a stronger security posture, but there are further things to consider. The coding standards do not currently have much relevance to a couple of the key principles, namely default deny and adhere to the principle of least privilege. These principles have been touched on a bit by some of the other topics in the security policy – encryption is one method of by-default denying access, since you must have the decrypting key to access readable data, and the triple-A framework (especially role-based access controls) implements the principle of least privilege. But adopting coding standards relevant to these topics would flesh out the security policy.  Similarly, the policy has gaps regarding physical access security and cloud-based computing standards. These are important to consider, especially as a company grows. Where is the data physically being stored? Is that location secure? If using cloud storage, is that through a third party, and does that third party keep its servers in another country?  Finally, it's important to stay on top of security patches and stay aware of newly discovered vulnerabilities. |
| **17**  **Conclusions** | In conclusion, I hope this security policy helps Green Pace transition to a DevSecOps approach, where security is addressed proactively and built-in to the code base itself. The use of automation tools will help get this process started, and help keep software secure throughout the lifecycle of the software. It may be helpful to keep tabs on the current landscape of software security and recent security breaches, and to do that I have included a link here that will take you to a running list of recent incidents. |
| **18**  **References** | Here are the sources that were used to develop this security policy.  Thanks for watching, and have a great day! |