# CS 405 Project Two Script

PowerPoint is narrated and there is no Video link for this.

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hello My name is Sammy Shuck and I am a developer for Green Pace. This is a presentation of the Green Pace security policy and practice for providing guidance and recommendations for the future. |
| **2** | Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time. This policy defines the core security principles and a defense in depth approach; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. |
| **3** | This shows the threat matrix of each of the risk assessments done. There are varying threats with high and low likelihood and priorities for remediation. These threats can be identified through automation with tools such as CPPchecker and can alert the developers before code bases are released into production. |
| **4** | There are 10 security principles that are generally used when defining the coding standards. Each standard has been mapped to a security principle.  ValidateInput Data   * Validate input from all untrusted data sources. Proper input validation can eliminate the vast majority of software vulnerabilities. Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user controlled files.   Heed Compiler Warnings   * Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code. Use static and dynamic analysis tools to detect and eliminate additional security flaws.   Architect and Design for Security Policies   * Create a software architecture and design your software to implement and enforce security policies. For example, if your system requires different privileges at different times, consider dividing the system into distinct intercommunicating subsystems, each with an appropriate privilege set.   Keep It Simple   * Keep the design as simple and small as possible. Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use. Additionally, the effort required to achieve an appropriate level of assurance increases dramatically as security mechanisms become more complex.   Default Deny   * Base access decisions on permission rather than exclusion. This means that, by default, access is denied and the protection scheme identifies conditions under which access is permitted.   Adhere to the Principle of Least Privilege   * Every process should execute with the least set of privileges necessary to complete the job. Any elevated permission should only be accessed for the least amount of time required to complete the privileged task. This approach reduces the opportunities an attacker has to execute arbitrary code with elevated privileges.   Sanitize Data Sent to Other Systems   * Sanitize all data passed to complex subsystems such as command shells, relational databases, and commercial off-the-shelf (COTS) components. Attackers may be able to invoke unused functionality in these components through the use of SQL, command, or other injection attacks. This is not necessarily an input validation problem because the complex subsystem being invoked does not understand the context in which the call is made. Because the calling process understands the context, it is responsible for sanitizing the data before invoking the subsystem.   Practice Defense in Depth   * Manage risk with multiple defensive strategies, so that if one layer of defense turns out to be inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability and/or limit the consequences of a successful exploit. For example, combining secure programming techniques with secure runtime environments should reduce the likelihood that vulnerabilities remaining in the code at deployment time can be exploited in the operational environment.   Use Effective Quality Assurance Techniques   * Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Fuzz testing, penetration testing, and source code audits should all be incorporated as part of an effective quality assurance program. Independent security reviews can lead to more secure systems. External reviewers bring an independent perspective; for example, in identifying and correcting invalid assumptions.   Adopt a Secure Coding Standard   * Develop and/or apply a secure coding standard for your target development language and platform. |
| **5** | The following are the coding standards and the priority order for each of them. They are ranked based on highest severity of the issue coupled with the Likelihood and remediation costs of the risk. This order represents the most dangerous to least dangerous risks. We can see that SQL Injections and safe memory management are very high priority standards that have a high impact and are the most severe threats to address. |
| **6** | There are three encryption policies. Encryption at Rest, Encryption in Flight, and Encryption in use.  Encryption at rest   * Encryption at rest is the policy to encrypt sensitive data stored on persistent storage such as credit card numbers or social security numbers stored in a database.   Encryption in flight   * Encryption in flight is the process of encrypting data between systems. This would be HTTPS/TLS encryption between a client browser and the backend web server.   Encryption in use   * Encryption in use is securing data that is being handled in memory during transit. This would be something like hashing a password and validating the hash against a stored hash in a database over HTTPS. |
| **7** | The Triple-A policies are the Authentication, Authorization and Accounting, sometimes called Auditing policies.  Authentication   * Authentication is the process of validating a user is who they say they are. This is typically done with username and password coupled with multi-factor authentication such as an RSA token.   Authorization   * Authorization is the process of validating a user’s permission or privilege to access or process some data.   Accounting - Auditing   * Accounting is the process in logging and auditing all actions of a user so that a trail can be rebuilt on what, when, and how a user accessed or modified data. |
| **8** | The coding vulnerability used for unit testing was a memory management vulnerability by testing vector operations such as adding, removing, resizing, and indexing out of range.   * **TEST\_F(CollectionTest, CanAddToEmptyVector)** * This first test tests whether the collection is empty and if one entry can be added to the collection. This is a positive test and should pass with a true value if the vector size is equal to one. If it does not pass, then then there is likely an error with the add\_entries method. |
| **9** | * **TEST\_F(CollectionTest, CanAddFiveValuesToVector)** * This test tests whether the collection is empty and if five entries can be added to the collection. This is a positive test and should pass with a true value if the vector size is equal to five. If it does not pass then then there is likely an error with the add\_entries method. |
| **10** | * **TEST\_F(CollectionTest, AlwaysFailIndexOutOfRange)** * This test tests whether accessing the collection index with an out of bounds index value will throw an exception. This is a negative test and should be expected to fail if, after attempting to access an invalid index within a try/catch statement, the exception is caught for out of range. If it does not throw an exception the fail is called and there is likely an error with the collections object. |
| **11** | * **TEST\_F(CollectionTest, AlwaysFailResizeToNegativeSize)** * This test tests the resizing of the collection to a negative value. This is a negative test and should be expected to fail if, after attempting to resize the collection, within a try/catch block, to a negative one value, an exception is thrown. If no exception is thrown then there is likely to be an issue with the collections object. |
| **12** | * **TEST\_F(CollectionTest, CanResizeIncrease)** * This test tests whether the collection is empty and if the vector can be resized to an increased value. This is a positive test and should pass with a true value if, after validating the collection is empty, the resize method is used to resize the vector by plus one to produce a vector size of one. If it does not pass, then then there is likely an error with the resize method. |
| **13** | * This is a standard automation diagram for DevOps. DevOps is a methodology aiming at establishing closer collaboration between programmers and system administrators in the software development process. A DevOps engineer's primary objective is to take the predictability, efficiency, and security of software development to the highest level possible. DevSecOps is a further development of the DevOps concept that, besides automation, addresses the issues of code quality and reliability assurance. |
| **14** | * DevSecOps is the approach to applying security to product development in the DevOps process. This allows for a more secure approach via various automation techniques to incorporate security within the DevOps model. This is the idea of implementing security as code into the development lifecycle and to maintain monitoring and analytics of the lifecycle. This is a simple concept but has important aspects incorporated into it. The idea that at all levels and integrations points security should be implemented to ensure a safe product to market. * Automated testing within a development lifecycle is the goal and target of DevSecOps. Implementing security CI/CD pipelines that scans code upon merging or unit testing when a merge is performed. Writing effective tests to broadly cover the code base to ensure continuous security compliance. |
| **15** | * The phrase don’t leave security to the end means that security needs to be addressed at all times. Many times developers will not think about security while developing a project. This is not a great idea as security is sometimes challenging to remediate. There are varying levels of remediation and threat levels as well. If we leave security to the end, it is a difficult task to remediate but will also introduce the risk of missing a security threat. It is best to create a set of standards that address security concerns. This allows for checks and testing to help identify the risks earlier. * There are many things a developer can do to prevent threats in their secure coding progression. The first and easiest way is to write unit tests that test for various security threats. Unit testing should often be performed and early to address these concerns quickly. Additionally, there are several software solutions out there that can be leveraged to scan and identify threats. One such software is CPPChecker. This application will scan the code base and will report varying levels of threats or issues. This will provide the developer a quick way to identify threats and remediate the threats. The developer can also implement various checks in a CI/CD pipeline to check for security threats upon a merge request. This could include running the unit tests. Also, there are a number of different capabilities with a CI/CD pipeline that can be implemented to validate code and reduce the number of threats. If threats are found, then the merge fails, and the code is not changed on the master branch. This can alert others as well and a more thorough code review can be performed to ensure all security threats are addressed. * Attackers have many motives that vary. They can be to steal money, obtain and sell data, revenge, disrupt services, and the list continues. Understanding these motivations will help the developer understand where the risks for their project are at and allow them to address any threats that may exist. |
| **16** | * When considering security, additional considerations have to be evaluated—those such as costs, time constraints, and operational capabilities. Costs are associated with the products or services that are purchased and the costs of hiring people to manage and maintain these products and services. Costs related to breaches also need to be in consideration as well. What are the costs to the business if a specific risk wasn't mitigated? These costs to risk analysis play a key role in determining the appropriate level of defense required. If a security breach happens, then there are reputational costs associated with not implementing security. This can lead to decreased sales and can have a downward spiral into the destruction of the business. * Operational considerations must also be evaluated. What level of effort to monitor and manage the various security layers is required needs to be answered. There needs to be a defined operational monitoring system and alerting capability to provide a real-time reaction to security vulnerabilities. The ability to patch vulnerabilities quickly and hopefully automated is required to promptly address security needs before bad actors have an opportunity to exploit the business. * Defense in depth is unique in certain aspects. There are physical, technical and administrative, or policy-driven defenses that make it unique. One area may be of higher focus or need over another. Maybe physical security is a more significant concern over a technical one. This may require more physical defenses, and the use of technical defenses will be reduced. Each situation and business will drive the depth in which defenses are deployed. |