# CS 405 Project Two Script Template

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hello, and welcome to my presentation on the new security policy here at Green Pace. |
| **2** | This security policy will outline 10 core principles for security best practices and 10 coding standards that we will use to keep everyone in sync with the best practices for coding securely as Green Pace continues to grow.  This policy is needed so that we don't lose sight of how to code securely throughout the software lifecycle and never saving it until the end.  This policy uses Defense in Depth, which is the idea that in order to have a secure architecture, you need to have multiple, independent, layers of security that all work to provide security. Having the multiple layers be independent means that if one fails to stop an attack, another will stop it. |
| **3** | There are 10 security threats that are outlined in this policy. They are each tied to a coding standard that will help mitigate the threat that they pose. For each standard there is a likelihood that the threat will occur and a priority for that threat to be resolved. Keep in mind that just because a threat is likely does not give it high priority, and just because something is low priority does not mean that that threat is unlikely. These are independent factors.  STD-001-CPP relates to selecting the right kind of data type that can handle all of the valid inputs so the system will operate as intended. While this is unlikely, it has a medium priority because if the mishandled data makes it to release there is a likelihood that bugs will occur.  STD-002-CPP is a policy that addresses a major threat in over and underflows, and it deals with ensuring that there is no loss or misrepresentation of data when converting from one data type to another. This is a probable error with medium priority since the logic can be difficult to identify, but must be addressed before release.  STD-003-CPP deals with string correctness, meaning that when you are converting from a string, verifying the data in the string is vital to avoiding errors. This is an unlikely and low priority threat, but that does not mean it can be ignored. It is important that inputs are validated to prevent events like crashes.  STD-004-CPP is a policy that relates to one of the most common forms of attacks, SQL injection. Always sanitize string data that is passed from a user. This is a likely threat with high priority and ignoring this will leave the program wide open to an attack.  STD-005-CPP relates to memory protection and it is about making sure that you initialize all variables before they are called. This is a probable threat with high priority since uninitialized memory being accessed will cause a crash of the system.  STD-006-CPP is about assertions, while these are invaluable for helping to find vulnerabilities, they need to be both limited and used only for identifying incorrect assumptions. This is an unlikely threat with low priority, keep this policy in mind as you write assertions and problems should be prevented before they even exist.  STD-007-CPP deals with exception handling. Make sure that all exceptions are properly handled, this prevents the default abort function from being called, which improperly handles resource deallocation and is a common foothold for Dos attacks. This is a probable threat with low priority, make sure that exceptions are handled as you write and double check that all possible ones are handled, including a catchall.  STD-008-CPP is about how commenting your code consistently and properly will ensure that the source code is maintainable both to developers that did not write the code and future you. This is an unlikely and low priority threat, but that does not make it any less valuable. Nothing is worse when maintaining code than trying to decipher a poorly commented source code.  STD-009-CPP relates to naming variables. Make sure that the same name is not used twice in the same scope, if you find yourself with two variables that have the same name, there is a good chance that the names are too generic. This too is a unlikely and low priority threat since proper naming conventions should prevent this, but failing to do so will result in undefined behavior.  STD-010-CPP is about logical completeness, when you are writing the code for a program, make sure that you are considering all possible paths and all possible states of data. This is a probable threat with medium priority. Make sure that when you are planning the logic for a program that you consider all possibilities.  While there are a lot of things to consider, and I just threw a lot of information out, there are automated tools that can help to identify vulnerabilities and aid you in eliminating threats. |
| **4** | These are the 10 core security principles that this policy was built around.   * ValidateInput Data   Validating input data is a vital part of secure coding. By validating the data, you will help protect yourself from buffer overflow/underflow which can allow an attacker to gain unauthorized access to private areas of the program or gain privileged information.   * Heed Compiler Warnings   Compilers come with a lot of warnings that help identify possible vulnerabilities, it is important to not ignore them. While the compiler may not catch everything, and it is important to code securely from the start, the warning list is a useful starting point for ensuring the code is secure.   * Architect and Design for Security Policies   One of the best ways to ensure a program is coded securely is to keep security policies and best practices in mind during the entire software development lifecycle. Do not be reactionary with secure coding. By using secure coding practices from inception through release, the company will save time, money, and public relations.   * Keep It Simple   One of the biggest coding principles is to keep the program as simple as possible. Avoid over complicated programs, ensure that names of variables and functions are descriptive, and be sure to use comments throughout the code to illustrate intention and explanations. By doing this you can reduce errors both in coding and in use.   * Default Deny   When creating access to secure areas of the program, deny permissions by default. The only way that access should be given is when specific conditions are met. By denying access by default, another layer of protection is added that can prevent outsider access to secure areas of the program.   * Adhere to the Principle of Least Privilege   When designing levels of privilege, users should be given as little access as possible. By only giving the access that is necessary, you can help protect yourself from unauthorized access to privilaged areas of the software.   * Sanitize Data Sent to Other Systems   By sanitizing the data, you can help protect yourself from things like SQL attacks and invalid input and output data.   * Practice Defense in Depth   When designing your cyber security, use a series of independent layers of security so that even if a single layer is made vulnerable, the attack will not be successful.   * Use Effective Quality Assurance Techniques   It is important to test the security you have developed in your program. Auditing, penetration testing, and bound testing are vital to ensure that the program is as secure as possible. It is not enough to assume you have accounted for all possible vulnerabilities.   * Adopt a Secure Coding Standard   Apply the best secure coding practices to your code from the beginning and in every program that you work on. |
| **5** | These are the 10 coding standards that are outlined in this policy.  STD-001-CPP, Select the Right Data Type, Selecting a data type that suits the program's needs and can accommodate all likely valid inputs allows for the secure handling of the data and proper coding and use of the program. This is a level 4 standard because it has a low severity, it is unlikely to occur, it has medium cost to remedy, and it has medium priority.  STD-002-CPP, Ensure Integer Conversions Do Not Result in Loss of Data, When converting from one type of integer to another, it is vital that there is no loss of data, and that the data is not misinterpreted. This is especially important when dealing with the character data type. This is a level 2 standard because it has a high severity, it is probable to occur, it has high cost to remedy, and it has medium priority.  STD-003-CPP, Detect Errors when Converting a String to a Number, When converting from a string to a number there can be several errors. The string numbers could be out of range of the number, the string could contain non number symbols, etc. It is important to make sure you verify the string before the conversion occurs. This is a level 1 standard because it has a medium severity, it is unlikely to occur, it has medium cost to remedy, and it has low priority.  STD-004-CPP, Detect Errors when Converting a String to a Number, String data that is passed to a complex subsystem may contain special characters that can trigger commands or actions. This can result in unpredicted behavior at best and unauthorized access to sensitive data at worst. It is necessary to always sanitize the data before passing it to ensure protection from things like SQL attacks. This is a level 5 standard because it has a high severity, it is likely to occur, it has medium cost to remedy, and it has high priority.  STD-005-CPP, Do Not Read Uninitialized Memory, Ensure that all variables are initialized before being called. Just declaring them leaves the values as indeterminate and unpredictable. Also, some memory allocation functions do not initialize the memory allocated for that variable. This is a level 5 standard because it has a high severity, it is probable to occur, it has medium cost to remedy, and it has high priority.  STD-006-CPP, Use a Static Assertion to Test the Value of a Constant Expression, Assertions are a valuable tool in helping to find vulnerabilities in code. However, because of the runtime overhead and because it calls abort(), it should only be used for identifying incorrect assumptions and not for runtime error checking. This is a level 1 standard because it has a low severity, it is unlikely to occur, it has high cost to remedy, and it has low priority.  STD-007-CPP, Handle All Exceptions, When an exception is thrown, if there is no handler found within the handler's try block, the search for a handler will search other try blocks within the same thread. If there is no handler, then the default will be used. The default std::terminate() calls std::abort() which terminated the process abnormally. When abort() is called the stack may not be properly unwound, destructors may not be called, and resources may be left in an indeterminate state. This improper handling of resources is a common foothold for a DoS attack. This is a level 2 standard because it has a low severity, it is probable to occur, it has medium cost to remedy, and it has low priority.  STD-008-CPP, Use Comments Consistently and in a Readable Fashion, When you are writing code, it is vital that comments are used throughout the code to ensure that it is maintainable. The comments must be consistent, relevant, readable, and understandable. If comments are omitted programs can quickly become unmaintainable as the code intentions become more complicated and less obvious to someone who did not write the source code. This is a level 2 standard because it has a medium severity, it is unlikely to occur, it has medium cost to remedy, and it has low priority.  STD-009-CPP, Do Not Reuse Variable Names in Subscopes, Do not use the same variable name in two scopes where one scope is within another. Reusing names will lead to programmer confusion, reusing often means that the names of the variables are too generic and need to be adjusted, and could cause problems with the program not altering the correct variable. This is a level 1 standard because it has a low severity, it is unlikely to occur, it has medium cost to remedy, and it has low priority.  STD-010-CPP, Strive for Logical Completeness, Ensure that when you are writing your program that you consider all possible paths and all possible states of data. Failure to do so will result in vulnerabilities. This is a level 3 standard because it has a medium severity, it is probable to occur, it has medium cost to remedy, and it has medium priority. |
| **6** | * Encryption at rest   + Encrypt all files when being stored and not in use. This means that all data being stored in a physical medium (hard drive, thumb drive, etc.) and virtual medium (database) is encrypted in the event of a breech or theft. * Encryption in flight   + This is when data is most likely to be misused, so encrypting data as it leaves, enters, and moves around the network is vital. * Encryption in use   + When data has been sent from a database or storage and has either arrived at the user or is being used by the system, it is important that the data remain encrypted. |
| **7** | * Authentication   + This is the process of the system verifying that the user is who the user says they are when the user logs into the system. This ensures that users cannot imitate other users to gain access to areas of the system they should not have access to. * Authorization   + This is the process of granting authority to a user depending on their level of clearance in the system. This ensures that users are given access only to the areas of the system that they need to perform their tasks in the system and changes made to the system can only be made by those that should be able to make those changes. For example, adding new users and altering the access levels of users should be reserved for individuals like the owner of the system. * Accounting   + This is the process of tracking and documenting use of the system, particularly the unusual activity in a system. This ensures that in the event of an attack, the source can be identified and rectified as well as knowledge to what was accessed in the attack. |
| **8** | One of the many tools at a developers fingertips is unit testing frameworks. While there are countless out there, each with varying degrees of robustness and each with their own sets of pros and cons, for this example Google's Unit Testing Framework was used.  At the start of the first test there are comments giving some background information on other areas of the program that are called, then the definition of TEST\_F is called with the parameters of the test fixture, or group of tests, and the test name. Next there are assertions to check the collection. The first is to check that the collection is created, and the next checks that the collection's smart pointer does not point to null.  The second example of a test follows the same definition with the test fixture followed by the test name and then a check to see if the collection is empty, and then verifying that the empty collection has the proper size of 0. |
| **9** | This test checks if the empty collection can be added to |
| **10** | This test checks if multiple entries can be added to the collection |
| **11** | This test checks that the max size adjusts accordingly with entries being added to the collection |
| **12** | This test is similar to the max size, but this checks the capacity of the collection |
| **13** | The first test in these examples ensures that the size decreases as the entries in the collection decreases and the second checks that the collection empties when the clear function is called |
| **14** | Similar to the clear test from the last slide, this first test makes sure that the collection is cleared from the beginning to the end of the list. The second test ensures that reserve changes the capacity and not the size. |
| **15** | This next test is an example of a negative test, it ensures that when an entry that is out of range is accessed that the proper exception is thrown. The test after that ensures that when an entry is pushed back that it is added to the back of the collection, and the final test in this example checks to make sure that pop\_back removes that last element in the collection. |
| **16** | The examples on the previous slides were a part of a larger file with a lot more examples of the kinds of tests you can run using unit testing. When you run the tests, these are the results that you can get. From the collection of tests called CollectionTest 15 tests were ran.  You can check if statements are true, false, equal, not equal, if an error is thrown, if the program is exited, and so many other checks. One of the biggest advantages is that these tests can all be run again in the future if changes are made to the program, and these tests can be altered just as easily as changing anything else with the source code. |
| **17** | Most of the standards defined in this document do not require a special time and place to be addressed, in fact doing so would negate the usefulness of these standards. Rather, these standards should be considered at every stage of the development process. Selecting the right data type is essential when designing and building the program but should not be forgotten about during the later stages like maintenance. Ensuring that data is not lost or misrepresented is vital when verifying and testing the program but is still important when monitoring and detecting possible attacks. Detecting errors when converting a string to a number is important during building and testing, but string manipulation is a common foothold for attacks as well and should be routinely tested during health checks of the system. Sanitizing data is important when testing a program for security risks, but security is a constant battle so monitoring and detecting possible SQL attacks is just as important. Not reading uninitialized memory is important to keep in mind when building a program but it is also important to respond to instances when this does happen as memory deallocation is unpredictable at best when the abort is called. Using the right assertions at the right time and ensuring all exceptions are handled are important both in the testing phase and in the subsequent health checks. Using proper comments is important when building the program, but also important when maintaining it. Finally, striving for logical completeness is important for the system from the planning stage, through the end of the software's lifespan. |
| **18** | \*\*Said after reading the slide  All of the tools and information on those tools can be found on the Security Policy Document and associated links. They can be found underneath each of the coding standards. |
| **19** | There is always a cost/benefit to consider whenever you are doing anything. The development world is no different. You can wait and hope that you are not targeted for an attack, risking your companies time, money, and relationship with its users or you can act now, prevent problems before they happen, prevent the project from being killed by an attacker, and keep its users safe and only risk that you miss something or that no one with malicious intent tests the strength of your security. |
| **20** | Some of the gaps that I noticed still exist in this policy are a lack of policy regarding interconnected devices that utilize networks, databases, and other internet of things as well as a lack of policy regarding preventing things like non-human users and data scraping. I linked below some standards that could be adopted to help prevent these problems. |
| **21** | In conclusion, there are a lot of areas of security that need to be kept in mind when going through every single stage of the development lifecycle. The standards that have been adopted into the Security Policy is not a complete and exhaustive list of every vulnerability that may present itself, nor is it exhaustive on all of the remedy methods. That being said, with this policy we will continue to strive for creating safe code that has the right data types used, does not lose or misrepresent data during conversions, validates string to number conversions, sanitizes data, does not read uninitialized memory, uses assertions properly, handles all possible exceptions, uses comments throughout the program that are consistent and understandable, uses proper naming conventions, and considers all valid paths of logic and states of data. |
| **22** | Thank you. |