# CS 405 Project Two Script

**Project Two: Security Policy Presentation**

Eric Breznen

10/15/2024

Link: https://youtu.be/ZBGV4p2tEfo

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Welcome everyone, my name is Eric, and I am pleased to be presenting the latest update to the Green Pace security policy. |
| **2** | It is important to remember that as technology evolves so quickly, so do means of attack, so it is our responsibility as developers to continue to evolve our security practices and keep most up to date on our defensive strategy. For this discussion, we will go through various aspects of the new policy and address the updated security principles and coding standards that apply to each as well as automation tools and unit testing that can assist with verifying that our work is secure. We will also go over aspects of the Triple-A framework and why it is a standard practice. Every aspect of this policy is part of an excellent Defense-In-Depth practice by covering as many holes in existing processes as possible with multi-layered protection. |
| **3** | Here we present a threat matrix that goes over the various standards we will be discussing. On the left-hand side of the matrix we have the more likely threats, and on the right side the unlikely. Both sides are listed in order of priority with the high priority on the top |
| **4** | These next two slides walk through the 10 security principles and give some examples from the coding standards outlined in the security principles document that is provided to you. The principles are:  Validate Input data – Validating input data is essential in ensuring that information being taken into a program from users is correct, will not cause any unexpected behavior such as buffer underflows or overflows which can compromise the integrity of the program and possibly allow unauthorized access to the system to include the retrieval of sensitive data.  Heed Compiler Warnings – Compiler warnings are meant to inform the developers when there is a potential issue that can cascade into multiple failures, causing unexpected behavior or data breaches. Proper investigation into any warnings that appear is essential to catching and resolving issues early on in development so that the issues do not occur when an application is live.  Architect and Design for Security Policies – Security should be at the forefront of any software design, not an afterthought. Security features should be built into the design at every level to mitigate the risk of attacks. Code like an attack will happen, not like it might happen.  Keep it Simple – Errors in overly complicated code are more difficult to find and fix and roundabout ways of accomplishing a task waste valuable memory resources. Keep the path from initialization to termination as straightforward as possible and utilize standard libraries to reduce the number of functions that are hand coded, limiting risk of developer error even more.  Default Deny - The only way a user should be able to access a system is through proper authentications. If they are not authenticated, they should be denied. This goes for any user-driven functions as well, if they do not have permissions, it should be an automatic denial with no way around it. |
| **5** | Adhere to the Principle of Least Privilege – Users should only have the bare minimum permissions required to perform the task they are meant to. A customer does not need the ability to delete a product from the database. This ensures nobody has access beyond what they are trusted and required to do.  Sanitize Data Sent to Other Systems – Any time data is output to a client device or another server, it must be checked to ensure that there is no sensitive data that is being transferred that should not be. Ensuring that there is no code that leaks data to unauthorized users is paramount in maintaining integrity and the security of the company’s and customers’ data.  Practice Defense in Depth – No single method or tool can protect against every type of attack. Defense in Depth is meant to maximize coverage and protection for the system and the data contained within.  Use Effective Quality Assurance Techniques – Test as extensively as possible and manually review code to check for errors that could lead to failure. Continue to periodically audit code after launch to ensure that the code continues to meet the standard for security.  Adopt a Secure Coding Standard - People’s definitions of what is “secure” may differ. Create guidelines that everyone must follow to make sure that the company’s standard for security is upheld. This also assists in the event that the original developer leaves the company, anyone else can pick up where they left off. |
| **6** | With the next two slides we will be going over a few coding standards that are expected throughout the company. We saw in the previous slides how these relate to our security principles, so let’s take a look at the standards actually are. Examples of compliant and non-compliant code are documented in the policy document for your review.   * Data Type - [STD-001-CPP] - Validate user input for proper data types. * Data Value - [STD-002-CPP] - Use data types that dynamically allocate memory. * String Correctness - [STD-003-CPP] - Format Strings with user input when entering into pre-written sentences. * SQL Injection - [STD-004-CPP] - Check for SQL code in user input and reject if found. * Memory Protection - [STD-005-CPP] - Clear std::cin buffers when rejecting input. |
| **7** | * Assertions - [STD-006-CPP] - Use assertions to assist with validating data during development, best not to use assertions post release. * Exceptions - [STD-007-CPP] - Do not abruptly terminate program. * Object Oriented Programing - [STD-008-CPP] - Write constructor member initializers in declaration order. * Containers - [STD-009-CPP] - Use valid ranges for iterating through containers. * Expressions - [STD-010-CPP] - Do not access an object before it is initialized or after it is destroyed. |
| **8** | Changing pace and moving over to encryption policies. Data in all three phases, storage, transfer, and processing or at rest, in flight, and in use must be protected. Lets talk about how:  Encryption at rest - User and company data should be encrypted while being stored on hard drives or databases to mitigate the risk of security breaches compromising confidential data. User passwords should be stored using hashing or by another one-way operation.  Encryption in flight Data that is being transmitted between devices should be encrypted using an RSA encryption algorithm. This will mitigate the risk of an eavesdropper gaining access to confidential information.  Encryption in use Data that is actively being processed should be raw to limit the chances of miscalculations or manipulations. Because of this other safeguards must be put in place to secure the data being processed. The company’s single sign-on and multi-factor authentication are in place in order to protect this data. |
| **9** | Looking now at the Triple-A framework, we want to make sure of a couple of things. Please note that these all apply to both virtual locations on the network and physical access to confidential rooms or floors in the building.   * Authentication- Each employee will have a unique password and have single sign-on permissions associated with their account. They will also be required to used unique physical security keys with a security pin for multi-factor authentication to be able to access the system to ensure that anyone who is able to access the network, should be accessing the network. * Authorization - No user will have any more access than is absolutely necessary to do their job. Temporary exceptions can be made with manager approval for unique circumstances with business justification. Each user will be added to LDAP groups based on their organization tree and role to manage permissions. This especially includes access to confidential files and ability to modify company databases. * Accounting - User activity will be logged in a private database for accountability and investigation purposes. This log will not be accessible to anyone who is not a manager and managers will only be able to see the activities of those below them in the organization tree to ensure that private data is kept private. |
| **10** | Now let’s look at some Unit Testing. Using google test packages, we want to thoroughly check test our code for functionality and security. However, we have to come to terms with the fact that it is both impractical and near impossible to test for every possible case. So how do we choose?  We want to get results on tests that test median inputs, those that are what any normal case might input, but we also want to test for edge cases just in case, with a good mix of tests, this should cover most inputs and be able to predict whether or not the cases in between median and edge cases will be able to be handled by the program.  We also need to test for both positive and negative results, and in this, test not only for user input, but to make sure that our own processes perform as expected.  The following tests are actions on a collection that we are taking including adding values, resizing, and two different ways of verifying that calls cannot be made to elements outside of the range of our collection. |
| **11** | For this example, we test whether or not an entry can be added to the collection without causing any errors. It starts with verifying that the collection is empty so that we know for a fact, if an entry exists later, it is new. Then we perform our function and subsequently use assertions to verify that the value was added to the collection. |
| **12** | As a continuation of the previous test, we now want to make sure that the system can handle adding more than one value to a collection at a time. Again checking to make sure it is empty to start, performing the function, and asserting that the new size of the function is equal to the number of added entries. |
| **13** | Here we are testing a function to resize our collection, specifically to make sure that resizing decreases the collection size.  We start by verifying that the collection is empty, and add an arbitrary number of values before calling the function to resize the collection.  We then verify that the new size of the collection is the value that we resized it to. |
| **14** | Now, I said earlier that we want a mix of positive and negative tests, but what is a negative test? What defines a negative test is that the tester purposefully writes the test in a way that the code should fail. The point of these tests is to verify that the error is handled properly. In this case, we attempt to access element 10 of an empty collection and test that it does, in fact, throw an exception. |
| **15** | This is another negative test for a thrown exception, this time, testing whether an iterator, a for loop, will throw an exception when the loop attempts to access an out-of-range element. |
| **16** | In addition to manually written unit testing, using automation tools can help check for errors and vulnerabilities that compilers missed. Automation tools will used the most during the build and the verify and test phase, but can also be included in the maintain and stabilize phase when adding new code. In the next slide we will introduce some of these tools that we can use. |
| **17** | There are many tools out there, and there are a lot listed down in the SEI CERT webpages for multiple languages, on the slide you can see a few common ones that are listed for the specific coding standards that have been listed so far, there’s cppCheck version 24.9.0, CodeSonar version 8.1p0, Parasoft version 2023.1, and Polyspace Bug Finder version R2024a to list a few.  Using these automation tools will catch issues that your compiler doesn’t since the compiler simply checks to see if the program will run, not that it is necessarily secure. |
| **18** | The purpose of adopting a security policy is to make sure that the systems we build are designed for security, and for success. Waiting until the end of a build to implement these requirements will only cost time, money, and manpower. Instead, implementing these the first time will limit the amount of rework that will be required as it can become quite expensive to fix these vulnerabilities. |
| **19** | Unfortunately, no security policy is complete. As technology evolves, so will methods of exploiting it for personal gain. Often software technology advances to address vulnerabilities or weakness that are found in previous versions such as all of the deprecated libraries that exist and that have been replaced with updated, more secure versions. Entire data types have been created such as using String to replace character arrays. It is our job, as secure coders to evolve defensive strategies to match and defend against the new threats. As a result, this security policy, principles, and coding standards should be reviewed and updated annually to combat the threat. |
| **20** | In addition to these coding standards and design practices, security awareness training should be included for employees, anyone can make mistakes no matter how well practiced they are and reminders in the form of trainings are good to refresh employees on not only virtual practices, but also physical ones, such as being aware of who is present around you and not leaving confidential information open to be seen and keeping track of company equipment that may have confidential information on it so that it does not go missing and become a threat. |