Afshin E. Ahvazi

Professor Mike Prasad

CS-405-T5552

June 21, 2023

CS 405 Project Two Script Template

Link to video presentation: <https://youtu.be/UccqbbiYquw>

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Good everyone. Welcome to our Green Pace security policy. My name is Afshin, and I will be walking you through this presentation |
| **2** | First, let’s discuss the defense in depth strategy.  Defense in depth focuses on implementing independent security layers that complement each other, so that if one of these layers fails in case of an attack, the others can still withstand and mitigate damages. This is because there are no perfect security methods that can protect our applications against all sorts of vulnerabilities. |
| **3** | Some of the top threats addressed in the security policy are exceptions, injection attacks, buffer overflows, and numeric overflows. We categorize them as likely, unlikely, high priority, and low priority.  Exceptions are likely to occur because they are an important part of any program. Many methods will throw exceptions based on a variety of reasons. So these need to be handled properly.  Injection attacks take high priority because of how dangerous they are. Attackers can inject their code and take over the application or extract sensitive information which can lead to catastrophic results.  Buffer overflows take lower priority because of how easily they can be prevented, simply by using safe classes like strings and vectors instead of arrays and literals. But that doesn’t mean they should be overlooked.  Lastly, depending on the application, numeric overflows are unlikely because new technology allows us to store and represent a wider range of numbers.  Using automation techniques such as static code analysis tools will help identify potential security vulnerabilities by offering warnings and suggestions so the developers could fix. |
| **4** | Next, we’re going to discuss a few security principles and explain how they connect to the coding standards, upcoming next.  We need to ValidateInput Data. This is one the first layers of defense a system can use to protect itself against many external threats, some as dangerous as SQL injection attacks. Validating data allows us to make safe assumptions when we store and use that data, therefore preventing crashes, errors, and unexpected system behavior.  As a coding standard, we ensure that division and remainder operations do not result in divide-by-zero errors. One way to prevent this, is validating input data.  Often times, the compiler gives us warnings can reveal important information about potential errors and vulnerabilities in the code. These aren’t fatal, but if ignored, they can make for some very hard to debug errors like overflows or accessing null pointers.  As another example, we should not modify string literals. In C/C++ (as well as other static languages) string literals are immutable, and attempting to change them will result in errors and unexpected system behavior, which is not always caught by exceptions  Concurrent QA during development allows us to detect bugs and errors as early as possible, therefore improving system security, reducing costs overall, and yielding defect-free products that satisfy the user experience. As a result, it is good practice to incorporate diagnostic tests using assertions. Assertions are ordinarily used for debugging during development and are turned off for release. |
| **5** | Here, we have a list of the coding standards from our security policy, some of which we discussed in the previous slide.  They’re organized based on how fatal they would be in the program. While some cause crashes, exceptions, or otherwise terminate the program, others are best coding practices we should follow for a safe program. |
| **6** | Here, we’re going to discuss different encryption stages and relevant policies  Encryption in rest is about encrypting data when it’s stored in databases or in files, so that even if the storage is compromised, the data remains secure. So, as a policy, all confidential system data should be encrypted before storage.  Encryption at flight involves encrypting data while it is being transmitted over networks to help against threats of eavesdropping or otherwise intercepting the connection. Thus, it is policy to transport data using secure internet protocols like TLS (Transport Layer Security) or (SSL) Secure Sockets Layer. We should also use strong encryption algorithms, like SHA-256.  Encryption in use involves data while it is being processed or otherwise used by the system and its components. It’s about providing protection against unauthorized access to sensitive data during its processing. And for this, we need to design a secure system architecture that uses this type of encryption. |
| **7** | In this slide, we’re going to discuss the Triple-A framework strategies, authentication, authorization, and accounting, and how they relate to our security policies.  Authentication is verifying the identity of a user before granting them access to the system. So a policy will require password-based authentication, where users have to use a unique username and password to validate their identity.  Once a user is authenticated, authorization will determine what permissions or privileges that user has in the system. Per policy, the users are assigned roles that define their abilities, like reading from or writing to specific resources, and executing certain functions.  Lastly, accounting is auditing the system. It involves tracking and recording how the users interact with the system. So the system logs have to be analyzed daily to ensure compliance, and detect unauthorized system activity. |
| **8** | In the following slides, we’re going to show how unit testing can help us identify vulnerabilities in a program. In this case, we’re working with a simple calculator that finds a person’s body mass index, but the program is susceptible to numeric overflows as well as the division by zero error. |
| **9** | This first unit test simply checks whether the function returns the expected results. So a person who weights 150 lb and is 2 ft tall should have a body mass index of approximately 183.1. |
| **10** | This unit test checks that when the function is given 0 for height, the result is infinite (undefined). |
| **11** | This unit test shows us that when the arguments for the BMI function are in range, the function does not cause an overflow error. |
| **12** | This unit test shows us that when the arguments for the BMI function are too large in value, a numeric overflow can in fact occur. |
| **13** | Here we have the results from the previous unit tests. As expected, all the tests have passed. |
| **14** | There are a number of ways that we can use automation.  For example, we can integrate a static code analysis tool that automatically scans the code for potential security vulnerabilities. An example is incorporating Cppcheck into the development pipeline. This would be included in multiple stages, like the Design and Build from the figure.  We should implement auditing tools that perform regular security scans on the system, so that potential vulnerabilities or policy violations may be identified and fixed. This would be added in the Monitor and Detect section.  In this section, we’re going to discuss some of the problems, solutions, and the risks/benefits involved if we act immediately or decide to wait when it comes to addressing vulnerabilities.  Immediate action means we’d have to divert resources like attention and time away from developing new features. But this could pay off if the team is successful in averting dangerous vulnerabilities in the system. However, if changes are rushed without proper testing, then we risk introducing new bugs.  On the other hand, the longer we wait before addressing an issue, the longer our system remains open to potential exploits. But, this will give us time for a more thorough analysis of the risks, making sure that the most effective solutions are implemented. |
| **15** | In conclusion, security is of utmost importance to our organization, which is why we’ve come up with this security policy to protect our systems from vulnerabilities and threats. |