# CS 405 Project Two Script Template

Anthony Baratti

April 20, 2025

Project Two: Security Policy Presentation

Presentation Video: https://youtu.be/hFVuLGIzZJw

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hello, my name is Anthony Baratti. Welcome to this presentation, which introduces a security policy plan for Green Pace. Today, we will review security principles and secure coding standards, which are recommendations and rules for secure coding in C and C++, and how they are applied to a security policy. We will also be reviewing defense in depth, Encryption, Triple A Framework, Unit Testing, Automation Tools, and the DevSecOps pipeline. |
| **2** | A security policy allows developers to address security concerns with confidence and authority. A security policy addresses the common security vulnerabilities, their causes, their risks, and the solutions to mitigating these vulnerabilities.  An adopted policy can help keep all developers working towards secure coding standards by implementing best practices and following the secure coding principles.  This helps lay the foundation of expectations for developers to maintain throughout projects, and allows a developer to take ownership of their work, reducing risks and creating a program that is secure.  Moreover, the policy reveals the risks and can be used as a checklist to prioritize vulnerabilities. Once enough of the checklist has been accomplished to reduce to the risks to a low enough standard, the project can be considered secure, depending on the needs and goals of the program.  The policy also helps us enhance our defense-in-depth strategy, which utilizes multiple processes and aspects of building and maintaining secure code to help reinforce the software and mitigate risks. Adding multiple layers as fail-safes can help create “security gates”. For example, proper logging of events can help a team discover when a vulnerability has occurred and allow developers to repair the vulnerability before further security risks occur. Another example is layering authentication with authorization, following the principle of least privilege to restrict access to unauthorized and unauthenticated users, such as an unauthenticated user only gaining access to lower-level components of the system because the actual authorized user was only granted access to those components. |
| **3** | A threat matrix represents the likelihood of a risk versus its impact. Likelihood suggests how often or how easy it is for the security issue to occur, where the risk involves the impact it will have on the company or the users involved. The risk could be financial loss, time loss, data loss, loss of life, or any other type of negative impact on the consumer, user, or company.  A combination of likelihood and risk is used to give priority to security concerns, and the recommendations and rules that are assessed can be categorized to help developers focus on mitigating the highest priority first. Often, there are automated tools that will help recognize and rectify common vulnerabilities, and should be utilized, especially when the risk is high. Static code review is a great way to start the process, having developers review each other's code for potential flaws and bugs. Automated tools can help with static code review as well. Dynamic testing can also help reveal vulnerabilities, testing the code under compilation, such as unit testing. |
| **4** | The 10 principles are: Validate Input, Heed Compiler Warnings, Design for Security Policies, Keep it simple, Default Deny, Principle of Least Privilege, Sanitize Data, Defense in depth, Effective Quality Assurance, and Adopting and Practicing Secure Coding Standards. The rules and recommendations for practicing secure coding in this policy can be mapped to these principles.  Recommendation DCL-007-C, Declaring function type can be mapped to Validate Input Data. Excepting appropriate data types to pass to functions is vital for the smooth and secure operation. Securely handling input parameters and return types ensures that unintended behavior is mitigated.  Recommendation INT-008-C, Verifying all integer values are in range, also applying to input validation. Ensuring there is enough buffer space for string or integer data can prevent memory leaks and overflows.  Recommendation STR-002-C, Sanitizing data passed to complex systems, is also a strategy for validating input data. SQL Injection in particular is a high priority risk, and ensuring that data is passed as it should be is a priority for maintaining a secure system. This includes datatypes, size, and range.  Recommendation MEM-030-C,Do not access freed memory helps prevent undefined behavior from occurring within a program. This can be mitigated when this recommendation is backed up by the use of effective quality assurance principle. Proper testing and code review can help detect attempts to access freed memory and help prevent unintended behavior.  The recommendation MSC-011-C, Incorporate diagnostic tests using assertions. Proper unit testing can help reveal bugs and defects in the code which can be corrected to help reduce the risk of vulnerabilities. Using assertions is a quick technique for fast testing of methods and exceptions. Static review and testing align with the effective quality assurance techniques outlined in the principles.  Rule ERR-051-CPP states that all exceptions must be handled. Using proper handlers ensures that the stack is properly unwound and external resources are managed before termination, allowing for an application to fail safely. This follows the principles of adopting a secure coding standard, where exception handling is a vital aspect of all coding projects, and defense in depth, where failing safely can help mitigate other risks such as memory leaks.  Rule OOP-055-CPP states that the use of pointer-to-member operators to access nonexistent members is forbidden. Ensuring that pointers are set to null when a member is non-existent ensures secure coding that does not access data that it should not, as well as mitigates undefined or unexpected behavior. This also maps to adopting a secure coding standard, using best practices to help eliminate bugs and errors. Practicing the use of effective quality assurance techniques through static code analyzing and unit testing can help ensure that OOP-055 is not violated.  Rule FIO-030-C States that code should exclude user input from formatted strings. Passing a tainted value to a formatted string can result in viewing stack contents, memory contents, or writing to arbitrary locations. This directly relates to keeping it simple. Only use hard-coded variables and strings to deliver formatted data. Using effective quality assurance, such as unit testing and exception handling tests, can help mitigate security vulnerabilities with FIO-030.  Rule MSC-041-C States to never hard-code sensitive information such as passwords, keys, protected information, or personally identifiable information. Hard-coding can expose information to unintended parties, such as attackers or shared computer systems. This follows the principle of designing and architecting for security policies. Securing and separating sensitive data is a vital step in mitigating this type of security threat. It also aligns with adopting a secure coding policy by separating and securing sensitive information in any project. |
| **5** | The coding standards have been set in priority order from top to bottom. They are prioritized in this manner due to their risk level, which is a combination of likelihood and impact. Higher occurrences of the standard along with the higher costs of impact the top of the list as a higher priority when it comes to applying secure coding practices. Testing and review of standards at the top of the list should be a must, while the bottom of the list should be determined as needed by the design, implementation, and time constraints of a project. Bear in mind, it is not to say that the bottom of the priority list can be ignored. It just means that focus and thoroughness should be applied more to the top of this list. Memory, data protection, and input correctness are put at the top of the list, while testing methods and error handling are lower on the list, as they require less attention and present the lowest risks of vulnerabilities within the code that lead to threats. |
| **6** | Encryption is to be performed on any data that is legally protected or sensitive, such as usernames and passwords. This applies to all states of data. Whether at in storage long term or short term, in transit, or in use.  Data at rest is particularly vulnerable because it generally contains sensitive data such as passwords or personally identifying information. Keeping the contents secure through encryption protects the information by not revealing the data to unauthorized parties who attempt to or actually gain access to it. Encrypting data at rest should be part of a standard security policy and helps provide a layer of defense in depth.  Data in motion is data that is being transferred from one place to another, and it is most vulnerable because it is susceptible to interception. Encrypting data in motion must be done carefully, using only the most secure and trusted encryption methods. This is especially true for protected data such as banking transactions and medical records, including emails and messaging services. Encryption in flight should also be a part of a standard security policy and a defense-in-depth strategy.  Data in use is defined as data that is currently being processed by a system. This includes active input, reads, writes, updates, or accesses. Encrypting and decrypting data in use can help conceal sensitive data from attackers and is particularly important during external processes such as third-party information processing, cloud computing, or frequent exchanges of information between processing components. Because frequent encrypting and decrypting can be expensive computationally, it is important to determine what data needs to be encrypted and what data does not |
| **7** | Triple A framework provides another aspect of defense in depth. The Triple A policies include Authentication, Authorization, and Accounting.  Authentication is a process that is applied that determine that a user is who they say they are. A simple example is a user login process. Providing the correct login credentials can help authenticate a user that is supposed to be there, and help deter users who are attempting to gain access without proper credentials. This is particularly important for systems of high risk, such as banking applications or even development environment access, such as GitHub repositories. Applying multifactor authentication can increase the security of authentication and add more layers of defense-in-depth.  Authorization is a separation of access. It is a way of allowing or restricting access to certain components of a system through the principle of least privilege. For example, an administrator or manager may have more access to a system to make changes or access secure data than an entry-level user. Designing the right of access appropriately can help mitigate risks, especially if the authentication of a user is compromised and a party gains access through unexpected means (such as account hacking). Utilizing authorization in combination with authentication can further enhance the defense-in-depth security strategy.  Accounting is a way of tracking the activity of a system. Proper process logging can help determine who, when, how, and where information and a system are accessed and used. This can help with early detection of vulnerabilities and can help contain and mitigate any damage during post-threat management. It can also help a team determine what went wrong and take steps to ensure that it does not happen again. With a complete Triple A policy, defense-in-depth is applied through multiple facets. |
| **8-12** | Unit Testing is a way to ensure that quality assurance is enhanced. Unit testing is a way of repeatedly testing software during development, upon completion, or after updates are applied. Assertions and expectations are applied to the code to determine its behavior, to ensure that vulnerabilities are addressed through the code’s lifecycle. It also shows that the code has been reviewed previously and offers a quick and easy way to review it as the lifecycle continues.  The first slide shows a simple unit test to show that the capacity of an array is greater than the number of entries. The test first shows a baseline check, using a “before each” build to create an empty array. Asserting that the baseline is in order is the first step to any test, otherwise, the test could be working with different starting parameters. The next step add an entry to the array, then ensure that the array is not empty. The test “expect greater than or equal to” takes the actual array and a comparison value, in this case, 1, which is the current size of the array. If the capacity is greater than or equal to 1, the test passes. Then 4 more entries are added, increasing the array to 5 values. The test is ran again to expect that the capacity is greater than or equal to the number of array values, and it also passes because it is true. The test is also applied against after adding 5 more entries, which increases the number of values to 10. Each of these expect GE tests return a Boolean value of true or false, and as long as all return true, the tests pass as shown at the bottom of the slide.  (NEXT SLIDE)  The second unit test shows the process of increasing and testing the array size. Against the test is set up with an empty array, which must be asserted for truth. Next, the array is resized to 10, and an assertion check would stop the test if the array was not changed from empty. Then it ensures that the array was actually increased to the size of 10 values, and it was due to the unit test passing.  (NEXT SLIDE)  The third test shows a successful decrease of an array to size zero. The test is set up with and tested for an empty array, and then 10 entries are added. Asserting that the array is not empty and that it holds 10 values ensures that it did not remain at zero before attempting to resize it back to zero. After resizing to zero, a test is run to expect that the values of the array are not zero, and it passes, as seen by the test result at the bottom of the slide.  (NEXT SLIDE)  The fourth slide shows an exception being thrown for attempting to access an index out of range. Once again, a base array with zero values is supplied to the unit test, and then 1 entry is added, which gives a position index of 0. When index 1 is attempted to be accessed using the at method, an out-of-range exception is thrown, as the test success shows.  (NEXT SLIDE)  The fifth slide shows throwing an exception for negative resizing. This test uses a mocked user input, where a user might enter a negative value when a positive value is expected, but not ensured. The base array is set up with zero entries, and the hard-coded mock user input of -1 is supplied as the collection resize value. The test expects a thrown exception when a negative value is supplied, and it does pass the exception test as shown at the bottom of the slide. |
| **13-14** | Security checking automation can occur throughout the entire DevSecOps pipeline. Assessment & planning can utilize threat modeling to analyze proposed designs and architecture as well as process the results from feedback fed through dashboards or alerts from a production environment when post-production security flaws are revealed. This input back into the cycle allows for developers to rebuild and refactor a more secure system.  (Next Slide)  During development, static analysis tools such as compiler warnings can be used to scan and mitigate vulnerabilities and bugs as code is written or committed. This is also a place where tools can be applied to ensure that proper architecture is used, such as single-instance architecture or model-view-controller separation of concerns.  During building and integration, dependency check tools such as OWASP can be deployed to scan open source and third party vulnerabilities. Automated policy enforcement tools can also be utilized to prevent builds or deployments should the code be in violation of the policies.  During testing, dynamic testing tools can be used during compilation for vulnerabilities and misconfigurations  Once in production, runtime analysis tools can be used to monitor, report, or even block threats as an application is running in its operating environment. Security monitoring software can help detect anomalies and provide early detection of security threats or breaches.  While maintaining software, security tools can provide results to feed back into development to help remove or reduce defects and vulnerabilities. Some security tools can even help block and report threats, allowing early detection and action, such as rolling back to a previous version if necessary. Utilizing the feedback from automation can be crucial for early threat detection and mitigation. |
| **15** | The benefits of applying a security policy include secure coding practices that produce efficient, working, and protected software. It also helps reduce bugs through automated security tools and provides a comprehensive and prioritized checklist for mitigating the assessed risks. Starting early with security can help save time and money in the long run by applying a coding standard from the beginning of a project, proactively managing the risks rather than reactively attempting to fix them as they inevitably happen.  The risks involved in waiting to implement security are software that has flaws and vulnerabilities, no warning systems or program logging, violations of coding security standards, and an increased cost and time to redeploy faulty software. Furthermore, if the impact is great, the reputation of the developers could be significantly damaged, negatively impacting future development opportunities and consumer retention. |
| **16** | Security is an ever-changing and always present aspect of any active software development. OWASP consistently reports new vulnerabilities daily, and new exploits are revealed as technology advances, especially with artificial intelligence, where phishing, data-mining/research, and denial-of-service can happen more effectively. Choosing the appropriate security tools for testing and monitoring is an important process in secure development. Although types of tools were addressed, researching the appropriate fit was not. Using reputable and reliable third-party software, such as encryption, is a vital decision when it comes to securing applications.  Planning the architecture and design with security needs and goals in mind is also a concern for any project. Consideration should be given to the platforms it will deploy across, the environments it will be maintained and operate within, and the third-party libraries that will be used and maintained as the lifecycle of the software moves forward. Researching the appropriate technology is critical in maintaining a secure and efficient program and helps add just one more layer of defense to the system.  And last, although the list of standards is strong, it is not entirely comprehensive. As the planning and development of the software moves forward, more standards will need to be considered as more elements and components are added to the system, increasing the risk of flaws and vulnerabilities (such as using databases and schemas or cross-site interactions). |
| **17** | While the coding standards, recommendations, and rules are crucial to securing code and mitigating vulnerabilities, the secure coding principles are a cornerstone for practicing these standards now and in the future. It starts with adopting and utilizing secure coding practices. That means designing and implementing policies that include the C and C plus plus recommendations and rules as software is developed, ensuring to compare the likelihood and the impact to asses and prioritize risks.  Using effective quality assurance techniques such as static code review, unit testing, and automated security tools can help ensure that flaws and vulnerabilities are considered at every phase in the software development lifecycle and the DevSecOps pipeline.  Using the highest level of compiler warnings can reduce the bugs that sneak their way into a program, and with the combination of security tools, teams can remain alert to the threats that are presented by the vulnerabilities.  Practicing defense-in-depth and security from the start is one of the best ways to mitigate and reduce risks, especially common ones. Securing data, using the Triple A framework, and applying coding standards and security policies are all ways to apply multiple layers of defense to address security concerns.  Applying the principle of least privilege can help reduce the impact of many types of security threats and breaches. Separating concerns can also help mitigate potential damage after an incident has occurred.  Sanitizing data, default denying, and validating input are all straightforward ways to reduce the likelihood of flaws or bugs making their way into software. Many coding standards apply to these three principles, and proper planning and design with these 3 principles in mind can save a lot of time and money if considered throughout the project.  And last, keep it simple. The complexity of architecture and code increases the risk of flaws and vulnerabilities. Keep the code concise, separated, and as simple as possible. More complex code means more security measures have to be in place to maintain, track, and alert within a secure system. |
|  | [Insert text.] |
|  | [Insert text.] |