**CS 405 Project Two Script**

Justin Starr

Department of STEM

CS 405 – Secure Coding

Professor Mike Alesso

August 18, 2024

# CS 405 Project Two Script

**Link to YouTube video:** [**https://www.youtube.com/watch?v=jPcKybpzs2U**](https://www.youtube.com/watch?v=jPcKybpzs2U)

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Title page:  Welcome today to this Security Policy Presentation developed by Justin Starr for Green Pace. |
| **2** | Overview: Defense in Depth  In preparation for Green Pace’s transition from a DevOps methodology to a DevSecOps methodology to become more secure, Green Pace wants to be well prepared for its security audit.  Developers have already been employing best practices, however, as the team grows, it is critical that everyone remains in sync with the principles and best practices, which is a significant aim of this security policy.  This security policy will be used to support the defense-in-depth best practice by integrating various security tools and principles at multiple layers in the development process. Defense-in-depth is a practice of incorporating security at multiple layers so that if one security measure fails, another layer of security is in place to help reduce or mitigate potential security flaws or vulnerabilities (Fruhlinger, 2022).  During this presentation, various layers of security will be discussed, such as following coding standards and principles for secure coding. Additionally, the implementation of automation, including the use of static testing, encryption policies, the Triple-A framework, and the use of unit testing to improve code reliability and security will be discussed.  Furthermore, is important to Green Pace that automation is maximized to ensure compliance and keep costs down. It will be later discussed where in the DevSecOps process, automation can be best utilized to help Green Pace achieve this goal. |
| **3** | Threats Matrix:  As shown in the threat matrix, we have a variety of threats that are classified based on the type of threat, the likelihood of each threat producing or introducing vulnerabilities during development, and the threat’s priority.  Each of these threats is specific to our area of work and has been identified as coding standards that should be followed, and therefore are included in this security policy.  Each coding standard begins with three letters indicating which coding standard it is, followed by a two-digit padded number for the standard, and a one to three-letter acronym representing the coding language it applies to.  We can see in the matrix, threats related to characters and strings (STR), Input validation and Data Sanitization (IDS), and Memory Management (MEM) have the highest likelihood of being introduced if they go unnoticed or are not handled correctly and prevented.  Security threats associated with Expressions (EXP), Exceptions, and Error handling (ERR), and one associated with Declarations and initialization (DCL) have a probable likelihood of introducing vulnerabilities if they are not handled appropriately.  Lastly, we have one threat related to Declarations and Initializations (DCL) that has a low likelihood of introducing vulnerabilities if they occur within our code. However, it is essential that while it is unlikely to occur, and has a low priority, this does not necessarily mean that the potential vulnerability should be overlooked. Because this security policy does include this standard, it must be addressed if it occurs during software development. |
| **4** | 10 Principles:  Here we see a list of 10 security principles that are applied to this security policy and which of the coding standards apply to each of them.  **Validate Input** means to validate input from all untrusted data sources.  This principle can be directly mapped to three of the coding standards in this policy related to Declarations and Initializations (DCL), Characters and Strings (STR), and Input Validation and Data Sanitization (IDS).  **Head Compiler Warnings** means always using the highest warning level available when compiling code. Only eliminate warnings by modifying code meaning don’t suppress warnings unless necessary. To help detect and eliminate vulnerabilities, use static and dynamic analysis tools.  All of the coding standards listed in this security policy can be applied to this principle.  **Architect and Design for Security Policies** means to essentially create a software architecture and design software to implement and enforce security policies.  Nearly all of our coding standards directly map to this principle because when we design our software it should be designed according to our established security policy which, enforces our coding standards.  **Keep it Simple –** As the name implies, keeping designs simple and not complex reduces the likelihood of introducing vulnerabilities.  Coding standards related to characters and string (STR), Memory Management (MEM), and Exceptions and Error handling (ERR) directly map to this principle because oftentimes, it can be seen with these types of issues that had the software been designed in a more simplistic approach, vulnerabilities related to these standards could have been prevented beforehand.  **Default Deny -** Essentially, this means that access should be denied by default and the protection scheme then identifies under what conditions access should be permitted.  The coding standard that directly maps to this principle is IDS-00-J, Prevent SQL injection.  **Adhere to the Principle of Least Privilege** – this means that processes should execute with the least amount of privileges that are needed to complete their job. If and when elevated permissions are accessed, this should be done in the shortest amount of time to accomplish the task.  Again, the coding standard that directly maps to this principle is IDS-00-J, Prevent SQL injection.  **Sanitize Data Sent to Other Systems**  - All data that is passed to complex subsystems should be sanitized (destroyed).  The coding standard that directly maps to this principle is IDS-00-J, Prevent SQL injection.  **Practice Defense-in-Depth –** Multiple defense strategies should be utilized to manage risk. This is the basis for this security policy and directly maps to all of the coding standards.  **Use Effective Quality Assurance Techniques –** When identifying and eliminating vulnerabilities, good quality assurance techniques are effective.  This secure coding principle directly maps to all of the coding standards because penetration testing and source code audits can help identify where coding standards have not been followed.  **Adopt a Secure Coding Standard** – means to develop or apply a secure coding standard for the target development platform and language.  This principle maps directly to all of the coding standards because each of the coding standards will be incorporated into the security policy as required coding standards.  Principles sourced from CERT Secure Coding (Seacord, 2023). |
| **5** | Coding Standards:  Here we see what each of the coding standards are that are identified in this security policy.  They have been prioritized by their likelihood to introduce vulnerabilities in code, and based on how high of a priority they are when considering their associated risk or severity.  First, with a likelihood of likely, and a high priority, we have:   * STR-50-CPP – Guarantee that storage for strings has sufficient space for character data and the null terminator * IDS-00-J – Prevent SQL injection * MEM-50-CPP – Do not access freed memory * STR-51-CPP – Do not attempt to create a std::string from a null pointer * MEM-52-CPP – Detect and handle memory allocation errors   Second, with a probable likelihood, and high priority, we have:   * DCL-50-CPP – Do not define a C-style variadic function * EXP-53-CPP – Do not read uninitialized memory   Next, with a probable likelihood, and low priority, we have:   * ERR-51-CPP – Handle all exceptions * ERR-50-CPP – Do not abruptly terminate the program   Lastly, with a likelihood of unlikely and a low priority we have:   * DCL-03-C – Use a static assertion to test the value of a constant expression   Again, the system of prioritization here is how likely they are to introduce vulnerabilities, and the priority is based on their associated risk or severity. |
| **6** | Encryption Policies:  Encryption is the process of securing data by using encryption algorithms that scramble plain text in a way that can’t be read by anyone who does not have the key to decrypt the data (Stouffer, 2023).  When do we encrypt? There are three states that data can be in which the need to encrypt the data exists: in rest, in flight or motion, and in use.  Encryption at rest refers to the encryption of data that is not being used and is not in motion (in flight) across networks and between devices. It is the practice of encrypting data stored on a device by encoding it using encryption algorithms (Clinton, 2023). Encryption at rest applies to this security policy because one layer of defense is to encrypt data that is not being used to protect it from potential attacks and to reduce the likelihood of the encrypted data being mishandled. Even if an attacker gains access to the encrypted data, the likelihood of the attacker being able to unencrypt the data is highly unlikely without the appropriate key, especially with the higher-bit encryption that exists today.  Encrypting data in flight or in motion is the process of encrypting data that is moving from one device to another over a network (OpsCompass, 2015). This is important and applies to this security policy because when data is in motion across a network or networks, it is then while in flight that it is at most risk for attack. If data in motion is not encrypted, an attacker that intercepts this data would be able to read all of the data. Another layer of defense for this security policy is to ensure that all data in flight is encrypted, making the likelihood of an attacker who could potentially intercept this data and read it, far more unlikely than if the data were not encrypted while in motion.  Encryption in use refers to encrypting data while it is being used (phoenixNAP, 2023). Typically, when data is being used (accessed or used by an application), the data is decrypted and converted to plaintext so that it can be more easily read and understood (phoenixNAP, 2023).  This poses a significant threat where if an attacker gained access to a system while data that was unencrypted was being used, the attacker could also be able to read the unencrypted data. Encrypting data in use is a method for using the encrypted data without ever having to decrypt the data, maintaining its security. Encryption in use is another defensive strategy that is implemented in this policy to help ensure that if an attacker did gain access to a system while data was being used, they would still not be able to read the data without the appropriate decryption key. |
| **7** | Triple-A Policies:  Another key component of this security policy and defense-in-depth security strategy is the incorporation of the Triple-A framework.  Triple-A, Authentication, Authorization, and Accounting (AAA), is a security framework that controls the access to system’s resources, enforces policies, and audits all usage (Fortinet, 2024).  Authentication, the first step in the Triple-A security process, is a network or application's way of identifying who a user is and that that user is who they say they are (Mylonas, 2018). When a user sends a request to be authenticated (a log-in request), the user’s credentials are compared to existing credentials within a database. If the user credentials are found and match the credentials in the database, the user is authenticated (Mylonas, 2018). Authentication applies to this policy because it is an essential component for keeping a network or application secure against potential attackers.  Authorization in Triple-A refers to enforcing policies, such as determining what services, resources, and the quality of activities a user is allowed to use (Mylonas, 2018). This could include what changes a user is allowed to make to a database, whether they are allowed to enroll or create new users, and what level of access the user has. Once a user is authenticated, the user is then assigned a specific authorization level that defines what level of access to a network and its resources they can have (Mylonas, 2018). This is another critical component of Triple-A defense that limits a user to only what is necessary for them to accomplish the tasks they need to accomplish.  Accounting in Triple-A defense refers to monitoring and keeping track of the resources that are accessed by users, when, by whom, and what if any commands are issued (Mylonas, 2018). This includes keeping track of what users log in, who makes changes to databases, and what files are accessed.  All three components of the Triple-A framework play a significant role in maintaining the security of systems and it is a critical component to this security policy and defense-in-depth strategy. |
| **8** | Unit Testing 1:  The following four slides introduce a variety of Unit tests that have been conducted based on the various types of vulnerabilities identified throughout this security policy.  It is essential that developers perform their own unit testing as a part of the development process to ensure they are not introducing vulnerabilities.  Unit testing is a critical component of this security policy.  First, we have the following test:  Create A Test to Verify Reserve Increases the Capacity but Not the Size of the Collection  This slide shows the use of a Google Test to test a potential vulnerability where improper memory management could lead to undefined behavior. This is a positive test to verify the function reserve increases the capacity but not the size of the collection. If for some reason it did increase the size of the collection as well, this could lead to data corruption and is unexpected/undefined behavior.  In this particular test, we begin by expecting that the collection is empty and that if it is empty then the size must be zero. We declare a variable of type int called startCapacity and initialize it to collection’s current capacity. We then declare and initialize another variable called increase Reserve that will be used to increase the capacity of collection. We then verify with an assert that the capacity of collection is now greater than the starting capacity of collection. We finalize the test by asserting that the size of collection has not changed and therefore should be zero.  Therefore, this test proves that reserve increases the capacity but not the size of collection and therefore, we will not be vulnerable to improper memory management in this case.  We can see from the test results that the test passes. |
| **9** | Unit Testing 2:  Here we have the test:  Create a Test to Verify the std::out\_of\_range Exception is Thrown When Calling at() with an Index out of Bounds  Here we are testing a potential vulnerability of accessing an element that is out of bounds. Doing so could lead to undefined behavior and data corruption. This is a negative test because we are expecting an exception to be thrown when using at() with an index out of bounds.  For this test, we begin by expecting that the collection is empty and if it is empty then the size must also be zero. We then assert that an exception should be thrown when calling at() with an index out of bounds. This is done by calling at with the current size of the collection + 1. We are expecting the std::out\_of\_range exception to be thrown and we can see in the test results that the test passes, which indicates that the exception was thrown. |
| **10** | Unit Testing 3:  The next test is:  Create A Test to Verify that Calling shrink\_to\_fit() Shrinks the Capacity to the Size of Collection  Here we are testing another potential vulnerability concerning improper memory management that could lead to undefined behavior. This is a positive test that verifies that calling shrink\_to\_fit() shrinks the capacity of collection to the size of collection. If it did not, this could lead to unexpected/undefined behavior as well as the corruption of data.  The test begins by expecting that the collection is empty, the size is zero, and that the capacity of collection is also 0. We then add five entries to collection and then expect that the size of collection is now 5. We increase the capacity of collection by 1000 and then expect that capacity of collection is now 1000. We then shrink the capacity to fit the size of the collection, and then assert that collection’s capacity and size are the same.  If the capacity and size were not the same at this point, this would mean that unexpected behavior occurred which could lead to undefined behavior. We see from the results, however, that this is not the case and the test passes because collection’s size and capacity are equal to each other. |
| **11** | Unit Testing 4:  Our last test is:  Create A Test to Verify that an Exception is Thrown When Trying to Reserve More Than Collection’s max\_size  Again, here we are testing another potential out-of-bounds vulnerability. This is a negative test because we are expecting an exception to be thrown when attempting to increase the capacity larger than the max\_size of collection.  In the test, we begin by expecting that the collection is empty and that if it is empty we can also expect that the size is zero. We then assert that an exception will be thrown (std::length\_error) when trying to reserve a size that is larger than max\_size() for the collection. This is done by reserving max\_size() + 1, and we can see from the results, that the test passes, meaning the std::length\_error exception is thrown because we are attempting to reserve a size that is out of bounds for collection because it is too large. |
| **12** | Automation Summary:  By looking at the DevSecOps pipeline shown in the diagram we see that by transitioning from a DevOps methodology to a DevSecOps methodology, security becomes a critical component in all phases of the Software Development Life Cycle (SDLC). Security is introduced from the very beginning of a project (pre-production) and implemented throughout each phase through the end of production. This process is an iterative process, enabling developers to continuously monitor their software and deliver frequent updates, and new features.  During the planning phase, security is considered by first analyzing the current threat landscape, researching and incorporating regulatory changes, updating the impact analysis, prioritizing the project backlog, and responding to new threats.  From there, the project moves into the design phase where security is implemented through the use of test-driven design, and application best practices, such as utilizing OWASP resources.  The project continues by moving into the build phase where the project is built in a secure environment, using trusted repositories, and secure open-source usage.  The last stage in pre-production is to verify and test the software using vulnerability scanning, ensuring sources are trusted (digitally signed), that the software is functional, is in compliance, and is when security testing is performed.  The project then transitions to the first phase of production, where the product is properly configured and deployed, including the use of security settings, and penetration testing.  After deployment, the product is monitored through the use of log collections, SIEM analytics, event alerting, and intrusion detection.  Responding includes blocking attacks, turning off services when necessary, and rolling the product back to prior states that were more secure.  Lastly, the product is maintained and stabilized by assessing against the security baseline, returning to the baseline or a stable state after an attack or compromise.  This process is again, an iterative process, and repeats from the beginning to assess and make changes, whether it be regarding new features or improvements and bug fixes that need to be made. |
| **13** | Tools:  The use of external tools such as Unit testing is implemented during the build phase of development. This is where developers are incorporating testing into their software applications to help ensure code runs correctly and to help determine where bugs may occur within their code. Components or individual units of the software are tested in isolation, which helps ensure that applications work as intended before being deployed.  Examples of Unit Testing frameworks include Google Tests, Catch Unit Tests, and built-in Microsoft Unit Tests for Visual Studio.  Compilers are another way of finding errors in code or potential errors that could result in unexpected behavior. Compilers are used during the build phase of development, and all compiler warnings and errors should be addressed appropriately.  Static analysis tools are an excellent method for incorporating automation into the development process to help identify potential vulnerabilities within software. Several tools exist that can be utilized across a large number of platforms and languages. Static analysis tools are implemented during the verification and testing phases of the development cycle. Another benefit to using static analysis tools is that they can help identify potential vulnerabilities in code that may not be found when compiling code.  Examples of static analysis tools are Cppcheck, Parasoft C/C++ test, and CodeSonar.  Links have been provided for reference. |
| **14** | Risks and Benefits:  A dilemma often faced in software development is whether or not to act now or wait. At what point is a problem a problem that must be addressed?  A variety of problems can arise if you decide to wait on security rather than acting now, such as the potential for vulnerabilities to be exploited that exist within software. This can lead to data breaches or theft, substantial financial losses, reputational damage, legal and regulatory consequences, and a project being terminated or scrapped altogether (Dang, 2023).  Benefits of acting now include the ability to prevent attacks from happening, cost savings associated when dealing with data breaches, improved product quality, meaning it's more secure and therefore more reliable, and the reputation of the company and the trust with customers is maintained.  Areas where the strategy is lacking is that sometimes it can be expensive to implement new security especially if an organization is large. Also, the amount of time it takes for larger entities to incorporate security or new security features can be time-consuming. This can lead to additional financial drawbacks if the deployment of a product is waiting on additional security implementation. |
| **15** | Recommendations:  Potential gaps that exist in the current security policy is that it is susceptible to human error. One example would be when setting up initial configurations, such as user permissions, it is possible to mistakenly grant access to users who should not have access to certain resources. Additionally, when updating settings, it is possible to make mistakes when configuring security components, such as misconfiguring a firewall.  Another gap in the security policy is that it focuses primarily on the pre-production phases of the development cycle and has incorporated limited strategies for post-production. The policy should require routine reviews and changes should be made accordingly and where necessary, especially considering the threat landscape is always evolving.  Because of susceptibility to human error, the policy should include specific criteria for how security configurations are handled, by whom, and how often. Additionally, the security policy should have well-defined security audits, meaning it should describe in detail what is required to be audited, and the methods by which these audits should occur.  Also, the security policy should have a well-defined strategy for mitigating intrusions, and what actions should be carried out if an intrusion were to occur.  As it concerns future potential gaps and improvements, one of the biggest gap is that there are so many unknowns. This means that we don’t know what we don’t know, and even with all of the correct tools and processes in place, it is still possible for threats to go unnoticed. This can be largely attributed to the fact that technology is continuously evolving and with that, so are the risks associated with these new technologies (Nayyar, 2020).  Another major potential future gap is preventing unauthorized third-party access. This is a problem that can easily go unrecognized because attackers can potentially compromise the credentials of outside vendors who have access to the organization’s network.  Last, privileged access is another future potential gap that is often challenging for organizations to manage, and because of this, they are often abused. Oftentimes, when creating new user credentials, employees responsible for this job become complacent and grant new users the same privileges as existing users, leading to excessive permission granting when not necessary. Also, the same can be said when reverting permissions or even removing credentials that are no longer needed. In some cases, employees who are no longer a part of an organization are still able to gain access to the organization’s network simply because their credentials were never revoked.  One real-world example of a data breach that occurred due to a security gap was the Capital One Data Breach that occurred on July 19, 2019 (Neto et al., 2024). In this data breach, the main reason why this occurred was because Capital One was using Amazon Web Services (AWS) and its cloud computing infrastructure to store sensitive data related to consumers. The attacker, Paige Thompson was able to gain unauthorized access to AWS servers because the firewalls were misconfigured, enabling her to execute commands remotely and retrieve the sensitive data (Neto et al., 2024).  A link to the case study has been provided, which also describes in greater detail the steps that could have been taken to prevent this intrusion.  It is largely attributed to the gaps in the organization's security policy that ultimately led to multiple vulnerabilities being exploited. |
| **16** | Conclusions:  Standards that should be adopted to prevent future problems are as follows:  A secure coding standard should be developed and implemented, which should be strictly adhered to by all team members now, and in the future. If team members are not following this policy, potential vulnerabilities will likely go unnoticed, exposing the organization to great risk.  During the development phase of the software development cycle, developers should incorporate unit testing and static analysis tools to help identify and mitigate potential vulnerabilities.  Encryption policies at rest, in-flight, and in use should be enforced to reduce the likelihood of data being exposed to attackers if they are able to gain access to the encrypted data. Without the private key, it is highly unlikely that even if an attacker retrieved sensitive data, that they would be able to do anything with it if the data is encrypted.  The Triple-A framework should be implemented to ensure users are who they say they are and only assigned the permissions that are necessary to complete the tasks they require. Accounting measures should be in place to monitor resource consumption and detect unusual or suspicious activity.  Automation should be implemented where possible to help reduce costs, however, automation policies should be regularly reviewed and updated accordingly.  Conducting routine security audits will help identify potential vulnerabilities that may have gone undetected.  Adhering to the DevSecOps methodology and incorporating security at every stage of the software development lifecycle will help address security issues sooner rather than later, reducing the associated effects of leaving security until the end.  Lastly, It is critical to remember that security is not only the responsibility of the organization but also that of the developer.  Thank you. |
| **17** | References:  Provided on presentation. |