# CS 405 Project Two Script Template

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

**Sheel Patel**

**August 18, 2024**

**Project Two: Security Policy Presentation**

**https://youtu.be/psFlF3u4ixY**

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hello, everyone. My name is Sheel Patel, and today I will be presenting the Green Pace Security Policy. I am going to be covering how we can ensure that our development practices remain secure and consistent as our team grows. |
| **2** | Our security policy at Green Pace is essential for maintaining a consistent approach to secure development across all our projects. The primary goal of this policy is to support our defense-in-depth strategy, which is a layered security approach that helps protect our systems and data from various threats. By applying secure principles at every stage of the development lifecycle, we can significantly reduce the risk of vulnerabilities and ensure that our applications remain robust and secure. |
| **3** | Displayed here is the Threats Matrix and a brief description of the categories. The Threats Matrix categorizes the different vulnerabilities we've identified within our coding practices. The matrix helps us prioritize which vulnerabilities to address based on their likelihood, severity, and the cost of eliminating or mitigating them. For example, SQL injection vulnerabilities are high-priority because they are likely to be exploited and can cause significant damage. On the other hand, some lower-priority issues, while still important, may be less likely to be exploited or easier to fix. |
| **4** | Our security policy is built on ten core principles that guide our coding practices. These principles include validating input data, adhering to the principle of least privilege, and practicing defense in depth. Each principle is directly linked to specific coding standards to ensure that our code remains secure and resilient against potential threats. By aligning our coding standards with these principles, we create a solid foundation for secure development. |
| **5** | Here, I've listed the ten coding standards we've prioritized based on their relevance to our security principles and the potential risks they address. For instance, our highest priority standards include preventing SQL injection and ensuring proper memory management. These standards are critical because they address vulnerabilities that could have severe consequences if left unaddressed. By following these standards, we can ensure that our code is both secure and maintainable. |
| **6** | Encryption is a key component of our security strategy, and this slide outlines our policies for encryption at rest, in flight, and in use. Encryption at rest ensures that data stored on devices or databases remains protected even if the storage medium is compromised. Encryption in flight protects data during transmission over networks, making it unreadable to unauthorized parties. Finally, encryption in use safeguards data while it is being processed or in memory, preventing unauthorized access even during active operations. |
| **7** | The Triple-A framework—Authentication, Authorization, and Accounting—is crucial for controlling access to our systems and data. Authentication ensures that users are who they claim to be, typically through Multi-Factor Authentication (MFA). Authorization verifies that authenticated users have the appropriate permissions to access specific resources. Accounting logs all user actions, providing an audit trail that can be analyzed for suspicious activity. Together, these three elements help us maintain tight control over our systems and ensure that only authorized actions are performed. Utilizing all three of the Triple-A policy is crucial. The recent MGM resorts hack was a combination of social engineering and poor user-based access controls. A malicious actor was able to convince MGM’s help desk that they were an employee, and they needed help resetting their password or MFA. The malicious actor was granted access to the account of a super admin with advanced privileges across all of MGM’s systems. In this situation, MGM should have had tougher verification questions to confirm the person's identity and prevent the social engineering attempt. However, their IT team should have set up better access controls, as well, so that a user didn’t have so much control over all their systems. This was a failure on multiple fronts that could have been prevented if they had adopted a better Triple-A policy. |
| **8** | In the next slides, I'll go over our approach to unit testing, which is used to ensure that segments of our code functions as expected. We use unit tests that utilize assertions and exception handling to validate code behavior. For example, we test to ensure that our code properly handles out-of-bounds errors and invalid input, which are critical for maintaining secure and robust applications. |
| **9** | In this example we cover verifying that an attempt to access an index which is out of bounds will throw an exception. The test first adds 10 entries to the collection and then asserts that checking if the collection is empty will return a false statement. It then asserts that the collection size will be equal to 10 and finally it asserts that an exception will be thrown when an attempt is made to access the 11th index of collection. This type of test is called a negative test because it tests that we appropriately handle the attempt to access out of bounds memory. Accessing out of bounds memory can cause unexpected behavior so it's important that we properly catch and handle it. Additionally, we can see from our results that our test was successful, and an exception was thrown. It's important to verify that exceptions are being thrown so that they can be properly handled. |
| **10** | This is another example of a negative test. In this unit test, we verified that an invalid input for resizing the collection will throw an exception. The test begins by verifying the collection is empty. It then asserts that the collection capacity must also be 0 if the collection size is 0. Finally, the unit test attempts to resize the collection to -1. We assert this will result in an exception being thrown. The results indicate the unit test was completed with no errors. Therefore, all our assertions were correct. Where the previous test attempted to verify that an exception is thrown when attempting to access out of bounds memory, this test attempts to verify that an invalid input throws an exception. |
| **11** | In this unit test we verified that our function for resizing the collection is able to reduce the size all the way to 0. It begins by first setting the size of the collection to 10. It then asserts that the collection size will be equal to 10. The code then calls the resize function on collection to resize the collection to 0. The code next asserts that checking the collection is empty will return a true statement. Finally, the code asserts that the collection size is equal to 0. The outcome of the results show that our tests were successful and thus our assertions were correct. When developing unit tests, it's important to create additional assertions as you're going through the test scenario. There could be a scenario in which the add\_entries function isn't working appropriately; thus, the collection size is never increased to 10. In such a scenario. The collection would stay at zero. Then when assert that the size of the collection is 0 it will return a true and we would think our unit test is successful. However, this might not be the case because the add entries function never updated the collection size. It could be that our resize function is also not working. |
| **12** | In this final unit testing example, we verify the collection is cleared when the clear function is call on a collection. The test begins by setting the collection size to 10 using the add\_entries function. The code then asserts that the collection size is equal to 10. The code next calls the clear function on the collection. It verifies the collection is empty by asserting the empty function will return true when used on the collection. As a second check, the code asserts that the collection size is equal to 0. This is another example which highlights the use of multiple assertions to be thorough in testing. In this case, the code asserts that the collection is empty when we call the empty function, but if the empty function isn't working correctly, it may return a false positive. The code would catch this error when it makes a second assertion on the collection size to verify it's equal to 0. If the test failed, we could investigate further and avoid pushing out bad code which can contain vulnerabilities. |
| **13** | DevSecOps integrates security into our DevOps process, ensuring that security is prioritized at every stage of development. Automation is key to this approach, as it allows us to enforce and verify compliance with our security standards consistently. By embedding security checks into our CI/CD pipeline, we can detect and address vulnerabilities early, reducing the risk of security breaches. |
| **14** | The pipeline shown in the previous slide illustrated the development lifecycle process when security is incorporated. Preproduction included designing and testing the software and production consisted of the deployment, monitoring, and maintenance of the software. There are several automation tools which we will employ in each phase of the development lifecycle. SonarQube and CodeSonar each continuously scan the codebase to verify secure coding standards are being adhered to. OWASP ZAP provides automated security testing, including penetration testing. These automations allow us to identify and address any security flaws before the code is deployed, ensuring that the software is secure from the outset. As we move into the production phase, deploy Ansible and Terraform to enforce our security configurations such as our encryption policy. We will utilize Security Information and Event Management tools to continuously monitor logs from the system, firewall, and applications to identify any anomalies. In the event a threat is detected, Splunk and ELK Stack will provide real-time alerts and enable automated responses to the detected threat. |
| **15** | Security should not be an afterthought; it must be implemented from the beginning. By integrating automation tools, like those mentioned in the previous slide, and unit tests into our CI/CD pipeline, we can identify and address vulnerabilities in our code early. Prioritizing threats based on their severity and likelihood allows us to apply multiple layers of defense to mitigate them effectively. The benefits include early threat detection and increased code reliability, but there are risks, such as the potential high cost and the need for widespread buy-in across the organization. However, delaying these actions could lead to greater costs because the codebase will be larger and take more time and resource to bring in line with the security policy. There is also a possibility that a malicious actor will exploit an existing vulnerability that we have not identified. A breach will cause our users to lose trust in us which will affect our company and team negatively. |
| **16** | While our current policy provides a solid foundation, there are gaps that need to be addressed. For instance, we should use a third-party white-hat security firm to perform annual security tests and identify vulnerabilities. This provides a fresh set of eyes on our codebase and removes any possible bias. Additionally, we need to enhance our pre-production process with more automation tools to detect and eliminate vulnerabilities earlier. The Equifax breach which occurred in 2017 was caused by Equifax not patching a vulnerability in the Apache Struts framework they used. The vulnerability had been identified months prior but Equifax just hadn’t patched it. The vulnerability was exploited by Chinese military hackers who expose the information of 147 million people. A breach of this nature could have been avoided if Equifax had embedded automation tools which would identify and patch vulnerabilities into its CI/CD pipeline. Implementing AI and machine learning technologies could further improve our security. DeepMind, a Google owned company, recently developed an AI which taught itself to play GO. The AI has taught itself so well that it's now defeating the best players in the world. AlphaGo Zero initially learned by playing against itself, which led to it developing new, unconventional strategies that had not been seen before. The self-learning algorithm used by AlphaZero Go and the pattern recognition used by AI, in general, can be utilized to identify security vulnerabilities and develop new defenses. We should also consider adopting a standardized secure coding framework, such as CERT C/C++, and developing an annual training program similar to Google's, to ensure all developers are up to date with our current security policies and practices. |
| **17** | In conclusion, immediate action is necessary to protect Green Pace from potential security threats. While this policy is a strong starting point, the security landscape is constantly evolving, and we must continuously adapt by adding new standards and adopting best practices. Integrating automation tools and unit tests into our CI/CD pipeline will help us reduce overhead and prevent breaches. Delaying adoption only leaves us vulnerable to attacks, and the costs associated with securing our systems will only increase over time. |
| **18** | Today's presentation utilized information from these sources. Please visit them if you would like more information on their respective topics. |