# CS 405 Project Two: Security Policy Presentation – Dylan Ngu

**YouTube Video Presentation:** [**https://youtu.be/HBNvLMWD8QU**](https://youtu.be/HBNvLMWD8QU)

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hi, my name is Dylan Ngu, and I will be presenting the new security policy I have developed for the development team at Green Pace. This presentation will include the policies, standards, principles, and best practices that help prevent the threat of potential security vulnerabilities in both code development and systems architecture. |
| **2** | This security policy is a standardized version of the implicit best practices that we follow as developers at Green Pace. This is used to establish a set of rules and guidelines, to ensure secure principles are consistently implemented when developing applications. By following the standards and policies outlined, we can support our security mission of defense in depth; where we protect applications using multiple layers defense, as shown in this illustration that highlights the different areas where our applications must be secured. |
| **3** | This slide depicts the threat matrix, which frames our policy by illustrating the threat level for each vulnerability that is covered in this security policy. The vulnerabilities are arranged vertically by likeliness, and horizontally based on severity. Therefore, items in the bottom left of the matrix, classified as Low Priority, pose a minimal threat because they are less likely to occur and aren’t severe. Whereas items in the top right, marked as Priority, can be viewed as significant security risks because they are both likely to happen and pose a severe risk if the vulnerability is exploited. The severity of a vulnerability outweighs its likelihood when determining its threat level and this is shown through the color coordination of the categories as well as their order on the left of the screen.  In order to detect these vulnerabilities, we will implement various automated security tools. For example, static code analyzers, like cppcheck, can be used to examine source code to identify potential errors and ensure adherence to coding standards. |
| **4** | This table lists the 10 core security principles and the coding standards that apply to each. Our coding standards are rules that we must follow in order to enforce our guiding security principles. As you can see, many of the coding standards align with multiple security principles and in order to become a coding standard, it must be supported by at least one of our principles. For example, principle 1, validate input data, states that all input data must be untrusted until properly verified. The 6 standards that align with principle 1, enforce the validation of all inputs, internal and external. |
| **5** | The next table depicts the 10 coding standards that were shown in the previous slide. These standards are listed in order, from top to bottom, based on their priority. Priority is determined by measuring both the likelihood and severity of the potential vulnerability, as shown previously in the threat matrix. Vulnerabilities that introduce greater security risks take higher priority, as highlighted by STD-004-CPP, which is our standard for preventing code and SQL injection. Improper compliance can result in compromising an entire system. |
| **6** | Our policies for encryption dictate how we will protect sensitive data within our systems. Encryption at rest refers to encrypting data that’s being stored, encryption in flight outlines how we will encrypt data that is being moved across a network, and our encryption in use policy will specify which data we will encrypt while it’s being actively used. Encryption polices add additional layers of security to protect sensitive data; even if an attacker were to gain access to our data stores or intercepts data in transit. Without the appropriate decryption key, encrypted data will be completely unreadable. |
| **7** | Our Tipple-A policies are used to support the processes of authentication, authorization, and accounting. MFA will be used to ensure access to our system is only granted to verified users. Strict authentication protocols help prevent unauthorized access. RBAC is used to determine what permissions and privileges each user has within the system and different levels of authorization will be assigned based on the role of the user. Authorization policies reduce the potential damage that can be done if an authorized account is compromised. Accounting is accomplished through the use of logging, real-time monitoring, and network segmentation to allow administrators to view who did what within the system. These Tripple-A policies combine to create a robust security framework that can withstand technical and human-centric threats. |
| **8** | The next four slides will illustrate how we can utilize unit testing with the Google Test Framework. This is a form of automated testing that validates the functionality of our program and identifies potential vulnerabilities. These examples test functions performed on a collection but can be adapted to test other programs and functions; they exemplify how to apply a unit testing framework.    The first unit test is used to verify adding a single value to a collection using the add entries function. Before we call the function we are testing, we must verify the preconditions of the test using EXPECT statements; in this case we are verifying that the collection is empty. After invoking the function, we use ASSERT statements to verify the results of the test because they will terminate processing upon failure whereas Expect will simply notify. If the assert statement fails, the result of the test will be a failure. |
| **9** | The next unit test shows how we can use a parameterized test function to run a test using multiple values. Here we are testing to see if the max size of a collection is greater than or equal to the size for 0, 1, 5, and 10 entries. Like the previous test, this is also a positive test case because we are verifying a successful result when calling a function. This is an example of how we can test for proper memory allocation; ensuring compliance to STD-010-CPP by verifying sufficient memory allocation. |
| **10** | This unit test is an example of a negative test, where we prove our program will behave appropriately when an error or exception occurs. In this specific test, we are verifying that an out-of-range exception is thrown when attempting to access an element that is out of bounds. STD-002-CPP dictates we must guarantee container indices are within a valid range. Using automated test cases like this one, we can catch this type of vulnerability during development. |
| **11** | In the last example, we are verifying that a length error exception is thrown when trying to allocate memory that is too large for a collection. This test case ensures we are complying with STD-005-CPP, by handling all memory allocation errors. We don’t expect this type of error to occur, but utilizing negative tests ensures errors don’t result in undefined behavior that could compromise our system. |
| **12** | Our security policy also dictates how we will implement a DevSecOps approach that utilizes automation to build secure applications. This diagram highlights how we can include security in all stages of the software development lifecycle and transform a traditional DevOps approach into DevSecOps. |
| **13** | Based on the diagram of the DevSecOps pipeline in the previous slide, this table illustrates the automated security tools that can be used at different stages. During the build phase, compilers, static code analyzers, and unit testing will be used to ensure potential errors and vulnerabilities are caught earlier during development. Before software is released, static application security testing can be done to scan source code to identify vulnerabilities. Dynamic application security testing and integration test will be used to test applications when they are running.  During production, automated penetration testing tools can assess the system for any exploitable weaknesses and other automated tools can collect and analyze log data to detect any anomalies.  By integrating automation into CI/CD pipelines, we can automate security controls and tests, which promotes consistency and can limit the introduction of vulnerable code earlier in the software development lifecycle. |
| **14** | While our policy is to build security into every stage of SDLC, it helps to understand the potential risks and benefits of this approach and one where we wait until the end.  Waiting to implement security can result in missing critical flaws and also be difficult to implement, especially in larger code bases. This requires applications to go through extensive testing to ensure an appropriate level of security. However, the chance of overlooking potential vulnerabilities is still high.  Choosing to act now and incorporate security throughout development aims to resolve these issues. While this approach can be complex and time-consuming, the enhanced security that can be achieved and the consistency of applications proves why our DevSecOps approach is superior. This requires developers to be properly trained to use security tools and adopt a mindset that prioritizes security. |
| **15** | Moving forward, we realize there are some gaps in the security policy that still need to be addressed. Incident response plans must be developed to outline the steps we will take in the event of a breach in security. Along with that, a recovery plan must be explicitly defined to ensure systems are restored to their normal state after a security breach. Our policy must also dictate how our developers will comply with the legal and regulatory requirements that govern different industries, such as GDPR or HIPAA. Finally, we must also be proactive with monitoring potential vulnerabilities and employing regular software patches to effectively mitigate any risks that may arise. |
| **16** | The MGM Resorts Cyberattack in 2023 serves as good reminder of the importance of a well thought out and effective security policy. In this attack, hackers were able to obtain administrator privileges by impersonating an employee to gain their log in credentials from MGM’s IT help desk. After the attackers gained access, they deployed ransomware and disrupted operations across all of their hotels and casinos. This resulted in financial loss of $100 million and unmeasurable damage to their reputation.  If MGM had established the appropriate security controls and trained their employees to enforce these controls, this breach could have been prevented. |
| **17** | In conclusion, we as a company must adopt a mindset that emphasizes the importance of security throughout an applications lifecycle. Employees must be trained on application security best practices and must comply with our security policy. Our security policy must also be periodically reviewed and maintained because as we know, security is a continuous endeavor and not a one and done thing. This also means we must conduct regular security audits to identify and address any potential issues. Security threats and vulnerabilities are constantly evolving; therefore, we must adapt as developers and ensure the delivery of consistently secure code. As employees of Green Pace, we must understand and follow all aspects of our security policy. |