# CS 405 Project Two Script

**Diego Bez Zambiazzi**

**https://youtu.be/nJh0QVR0RK0**

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hello. My name is Diego, and I will discuss the details regarding the security policy created for Green Pace. |
| **2** | Cyber-attacks can have severe consequences to organizations, including financial loss and damage to public image. It is crucial to implement a defense in depth strategy addressing layers of physical, technical, and administrative vulnerabilities to prevent attacks from being successful. Using tools like encryption, firewalls, VPNs, and sanitizing data shared between systems can reduce risks of common attacks such as SQL injection, Denial of Service, and man-in-the-middle attacks. Investing in physical security, like biometrics and other forms of authentication, authorization, and accounting, can also reduce the risk of attacks performed by internal actors. Following coding standards, creating policies, and implementing a DevSecOps process is needed to enforce these practices and strengthen the organization’s security. |
| **3** | The coding standards, which will be discussed in depth in the following slides, can be organized in terms of likelihood and priority. As the word implies, the likelihood of a standard refers to how probable it is to find that issue in your code. The priority also takes into consideration its impact and remediation costs. |
| **4** | There are ten core security principles listed in this policy. It is also a best practice to list the coding standards related to them, to ensure that the standards are aligned with the principles.  The first principle is to Validate Input Data. Invalid input data can lead to unexpected behavior and program failure. For instance, if a program is expecting an integer as an input value, entering a letter would represent an error. Therefore, input data must be validated to ensure that programs behave correctly.  The second principle is to Heed Compiler Warnings. Developers must not ignore compiler warnings. Although code can be compiled and executed in the absence of errors, compiler warnings are often a sign of bugs that can lead to security flaws.  The third principle is to Architect and Design for Security Policies. An application’s architecture and design should be aligned with the security policies required. One example of architecture and design based on security policy could be compartmentalizing the software according to different levels of access privileges.  The fourth principle is to Keep It Simple. The complexity of security mechanisms is directly correlated to the complexity of the application’s design. Therefore, keeping the design simple will mitigate the possibility of errors and security flaws.  The fifth principle is to Default Deny. Controlling user access to a system based on exclusion, meaning allowing access by default and excluding specific users, increases the system’s vulnerability. Instead, users should be denied access by default unless they meet specific criteria for permission.  The sixth principle is to Adhere to the Principle of Least Privilege. Programs should always execute processes with the least privilege possible for the operation. This approach prevents scenarios where attackers need elevated privileges to exploit a vulnerability.  The seventh principle is to Sanitize Data Sent to Other Systems. Exploiting vulnerabilities in the communication between systems, such as SQL injection, is a common type of attack. Therefore, sanitizing the data sent to other systems is crucial to secure coding. A program should be able to identify and escape injection characters and commands before sending data to other systems.  The eighth principle is to Practice Defense in Depth. Since there is no security mechanism that addresses all vulnerabilities and is completely secure, it is necessary to practice defense in depth. Combining multiple strategies and layers of defense mechanisms blocks and delays attackers in different levels. Examples of these layers are policy, configuration, authentication, and authorization management.  The nineth principle is to Use Effective Quality Assurance Techniques. To verify that the mechanisms used to protect a system from attacks are working, it is necessary to use effective QA techniques. For example, tests can be performed to verify that defense mechanisms are effective against injection and overflow attacks, as well as unauthorized access to sensitive data.  Finally, the tenth principle is to Adopt a Secure Coding Standard. Adopting secure coding standards is essential to developing secure programs. These standards help to mitigate errors and vulnerabilities, and to ensure that security concerns are taken into account during the development of the code. |
| **5** | In this slide, the policy’s Coding Standards are organized by importance based on their likelihood and priority.  The most important standard on this list is STD-004-CPP, which is to Sanitize query strings to prevent SQL injection using “’ or 1=1” approach; a common type of SQL injection attack. If this attack is not prevented, the query can return sensitive information. It is important to also note that not only 1=1 can be used, but any value that makes the statement true.  The next standard, STD-003-CPP is to Guarantee that storage for strings has sufficient space for character data and the null terminator. Not guaranteeing that a variable has enough space for characters can result in a buffer overflow.  STD-005-CPP states Do not access freed memory. Reading a pointer to memory that has been deallocated, also known as a dangling pointer, results in undefined behavior. Therefore, freed memory should not be accessed.  STD-009-CPP is commonly warned by compilers and states Do not read uninitialized memory. Variables must be initialized before the memory can be read. Otherwise, reading uninitialized memory results in undefined behavior.  STD-008-CPP states that Value-returning functions must return a value from all exit paths. An exit path that does not return a value in a value-returning function results in undefined behavior. Therefore, all exit paths must return a value in these types of functions.  The next standards have medium to low priority due to their likelihood and severity.  STD-010-CPP is to Detect errors when converting a string to a number. Converting strings to numbers can result in errors; therefore, it is important to throw and handle exceptions when these issues happen. This prevents unexpected and undefined behavior.  STD-002-CPP states Do not cast to an out-of-range enumeration value. Enumerators are used to group and define a range to a specific set of integer values. Casting to an out-of-range enumeration value will result in unspecified value and lead to unspecified behavior.  STD-007-CPP is simple, Handle all exceptions. When an exception is not handled, the program will terminate abruptly. Attacks such as Denial of Service commonly exploit abnormal termination of applications. Therefore, it is necessary to handle all exceptions thrown and terminate the program in a controlled manner.  The next standard, STD-006-CPP, regards to assertions, and states that Expressions used in assertions must not produce side effects. Assertions are great mechanisms for ensuring that the program behaves as expected. However, since assertions are normally removed from the final code for release builds, expressions that have side effects will also be removed, which can lead to a bug.  Lastly, STD-001-CPP Never qualify a reference type with const or volatile. Only non-reference types can be qualified as const or volatile. Otherwise, trying to qualify reference type with const or volatile will result in undefined behavior. |
| **6** | A few policies are recommended regarding different uses of encryption.  Encryption-at-rest is the encryption of stored data, or at rest, and is essential to cyber security. This way, even if an attack captures the data, it will be unreadable unless the attacker also obtained the encryption key. Recommended policies are to encrypt user passwords and to store encryption keys in safe locations.  Encryption in flight is the encryption of data in transit between systems and networks. It avoids interception attacks from being able to read sensitive data. Recommended policies are to use secure protocols for communication across computer networks and to use an encryption tunnel to connect to the company’s network. This ensures that the data in motion is protected.  The third type of encryption, encryption in use, protects data while it is being processed and read from memory. This requires new forms of technology, such as confidential computing, which functions as a black box that only allows authorized programs and users to read the encrypted data being processed. Encryption in use is ideal to prevent malicious attacks, internal threats, and business partners to access sensitive information in memory. A recommended policy is to use confidential computing for running applications that process sensitive information. |
| **7** | Next, we must also define policies for authentication, authorization, and accounting.  Authentication can be described as the process of identifying users and is an elementary step to ensure that systems are secure. Recommended authentication policies are to enforce unique usernames and strong passwords, as well as requiring passwords to be changed often.  Once authenticated, users should be authorized to only access predefined levels of information, such as authorized databases. Furthermore, users should only be allowed to perform specific actions. A recommended authorization policy is to follow the Default Deny principle and limit user access based on their permission levels.  To avoid access violations and attacks, Accounting should be used to monitor user activity. Generating log reports and monitoring activity help mitigate threats by identifying the resources being accessed, the users accessing them, the time of access, and actions being performed. |
| **8** | Unit Testing is also an essential part of creating a secure application. For this example, I will show how we can use Google Test to find a bug related to the standard that says that Value-returning functions must return a value from all exit paths. |
| **9** | First, if we were to create a unit test to verify that passing a negative number as the argument to the function returns its positive counterpart, we would not see any error and the unit test would pass for the noncompliant code. That is because the if statement returns the correct value. |
| **10** | However, when we create a unit test for the positive argument, the if statement condition is not met and no value is returned. We see that, because this causes undefined behavior, the value stored in the variable returned\_value is 9435792 instead of 1, which causes the test to fail. |
| **11** | Because this example is very simple, this bug can easily be fixed by adding a return statement outside of the if statement to return the argument value when it is not negative. |
| **12** | Furthermore, unit testing is helpful when we need to better understand other behaviors that the function may have. For example, we can create unit tests to verify how the function would behave in case of implicit type conversions. |
| **13** | To discuss the automation tools available, we must first define the process being implemented. DevOps is a great process that focuses on collaboration through development and deployment stages that allows for frequent and efficient product deliveries. DevSecOps extends DevOps by also addressing security concerns throughout the process instead of waiting until the end of the software development lifecycle. In pre-production, security factors are evaluated during planning, design, build, and testing stages. This involves analyzing the software’s threat landscape, following development best practices and processes such as test-driven development, performing secure build, using trusted repositories, as well as performing vulnerability scanning and security testing. In production, the application is deployed, and vulnerabilities are monitored, which is done through penetration testing, log collection event alerting, and intrusion detection. In case an attack is detected, a response is needed to block the attack and maintain and stabilize the application, restarting the process. |
| **14** | Automation is an important part of DevSecOps, allowing the process to be efficient and developers to focus on higher-level aspects of security. Each step of the process can be improved with the use of automation tools. In pre-production, we can use threat modeling, unit testing, mocking, compile build checks, and static application security testing when planning, designing, building, and verifying and testing our application respectively. In production, we can use tools to help us to transition and health check the application, as well as monitor, detect, and respond to threats, and stabilizing the application. These tools allow us to run deployment health checks, runtime application self-protection, automated incident response, and assess and return to a stable state. |
| **15** | Now that we have a comprehensive understanding of these policies, we must define when and how to implement them and the benefits and risks of this strategy. Improving security can be a gradual process, but we must act now. We should focus on high priority standards, encryption, triple A, and automation tools, as these are the most important aspects of this policy. The benefits of this strategy are improving the level of protection against the most common types of attacks and mitigating their consequences, which could be severe. Even though this strategy is efficient, there is still the risk of having low priority or unknown vulnerabilities exploited. |
| **16** | Additionally, my recommendations would be to further invest in Defense-in-Depth and to study the motives behind attacks. First, as mentioned previously, having multiple layers of physical, technical, and administrative protection increases security considerably. Including practices such as code reviews, which also reduces the risk of insider threats. Secondly, considering the motives behind attacks is important to defining how to respond to different incidents. For example, Colonial Pipeline, one of the largest pipeline operators in the United States, suffered a ransomware attack in May 2021. The motive was determined to be money, and the company’s response was to pay the ransom for the decryption key to their files. They acknowledged that their response could have been different if the motive had been determined to be more complex, such as hacktivism or political. |
| **17** | In conclusion, there are many ways in which this policy can be improved. As part of the DevSecOps process, continuous assessments should be made to proactively update the threat landscape and automation tools. Lastly, third-party verification is becoming increasingly important as libraries and frameworks account for 80% of most applications’ code. A real-world example of why this is important is the attack suffered by ParkMobile in March 2021. The company stated that the vulnerability that resulted in user data to be leaked came from a third-party software used in their application. |
| **18** | Thank you, and these are my references in case you are interested in reading more about automation tools and the Colonial Pipeline and ParkMobile cyber-attacks.  References  Bird, J. (n.d.). Chapter 4. Security as Code: Security Tools and Practices in Continuous Delivery. *O’Reilly*. Retrieved on August 14, 2021, from https://www.oreilly.com/library/view/devopssec/9781491971413/ch04.html  Gast, K. (2020, October 8). Top 5 Reasons to Invest in an Automated Incident Response System. *LogRhythm*. Retrieved on August 14, 2021, from https://logrhythm.com/blog/top-5-reasons-to-invest-in-automated-incident-response/  Kisielius, J. (2020, March 17). Automated Incident Response Explained. *AT&T*. Retrieved on August 14, 2021, from https://cybersecurity.att.com/blogs/security-essentials/automated-incident-response-in-action-7-killer-use-cases  Osborne, C. (2021, May 13). Colonial Pipeline attack: Everything you need to know. ZDNet. Retrieved from https://www.zdnet.com/article/colonial-pipeline-ransomware-attack-everything-you-need-to-know/  Update: Security Notification - March 2021. (2021, Mar 26). ParkMobile. Retrieved from https://support.parkmobile.io/hc/en-us/articles/360058639032-Update-Security-Notification-March-2021 |