# CS 405 Project Two Script

Gagujas, Michael

CS405 – Secure Coding

June 24, 2024

Project Two: Security Policy Presentation

Video Link: https://youtu.be/LrXnA6Y5qhg

| **Slide Number** | **Narrative** |
| --- | --- |
| **1**  **(Intro)** | Hello everyone, thank you all for taking the time to listen and be here. My name is Michael Gagujas, and today, we will be going over a presentation for Green Pace’s security policy.  As technology continues to rapidly grow with cyber threats becoming more common and impactful, the necessity for a security policy becomes increasingly critical as organizations need to implement protective measures that can mitigate and prevent these threats. As a company, maintaining the integrity of our systems...protects our customers, builds their trust and confidence in us, while also protecting our own assets and reputation. This security policy will establish clear guidelines that follow regulatory compliance and allow us to respond to security incidents in a fast and effective manner. |
| **2**  **(DiD Illustration)** | When looking at our defense-in-depth illustration, we can observe the numerous layers that are available in our system, which the security policy will address as it defines and supports various defense-in-depth best practices described later in our presentation. Whether it is ranging from the outer, physical layer, where physical locations like rooms containing high value items, need to be securely locked.... to the inner, app security layer, where session management and proper error handling are implemented to improve security. Ultimately, our security policy will support defense-in-depth best practices by providing a framework for implementing layered security controls that promote a culture of continuous security awareness among all of our employees. |
| **3**  **(Threats Matrix)** | Looking at our Threats Matrix Table, which contains the Open Web Application Security Project’s Top 10 application security risks, we’ve categorized which vulnerabilities are a priority, likely, unlikely, and of low priority. Starting with the highest priority threats, we have security misconfiguration, missing function level access control, and components with known vulnerabilities. These threats are of a high priority because they are considered critical security risks that require immediate attention and remediation due to their potential for damage when exploited. Security misconfiguration can occur in various layers of the application stack and can involve improper admin rights, error messages containing sensitive information, and default accounts with unchanged passwords that can lead to unauthorized access and potential data breaches. Missing Function Level Access Control, or Broken Access Control, is when users gain unauthorized access to certain parts of a web application because of situations like unchecked user permissions. Components with Known Vulnerabilities are straightforward and are a priority risk because they contain software that already has known security flaws. These critical security risks must be thoroughly addressed to prevent disrupted service operations, stolen sensitive data, and unauthorized access.  The next category is “Likely”, which contains injection attacks, cross-site scripting, and sensitive data exposure, which are common threats that should also be prioritized because of the potential damage they may cause. Injection attacks, whether SQL, Operating System, or Lightweight Directory Access Protocol (LDAP), involve user-supplied input that injects commands because it wasn’t properly sanitized. Cross-site scripting attacks involve injecting malicious scripts into web pages viewed by others because of improper validation or escaping.(while) For Sensitive Data Exposure, it is categorized as “Likely” because of its potential impact and the frequency of unintentional exposure of private data like credit card numbers, Social Security numbers, and login credentials. Overall, each threat’s common occurrence warrants proper addressal with robust security measures and mitigation strategies to significantly reduce their risk... and protect sensitive information.  The final two categories in our threats matrix table are “Low Priority” and “Unlikely”.  The threats that are unlikely because they are less common and require specific conditions are “Unvalidated Redirects and Forwards”, where an application accepts untrusted input that can cause it to redirect the request to a malicious website, and “Insufficient Logging & Monitoring”, where inadequate tracking mechanisms or processes fail to identify suspicious activity or create an inability to respond effectively to incidents. The low priority threats, “Insecure Direct Object References”, where internal implementation objects are exposed without any access controls, and “Cross-Site Request Forgery”, where unauthorized commands are transmitted from a user that the web application trusts, are still serious threats, but they are ranked lower because they are less common and also require specific conditions to be exploited.  Ultimately, these top threats should always be considered, and a practical and effective approach to detecting these vulnerabilities, is through automated Static Application Security Testing (SAST) tools and Dynamic Application Security Testing (DAST) tools. Static Application Security Testing tools like CppCheck can analyze source code for patterns that indicate these potential vulnerabilities and are cost effective for early bug detection and improving overall code quality so that it also follows industry compliance. Dynamic Application Security Testing tools, on the other hand, add to a comprehensive strategy for protecting from vulnerabilities by testing the application in its running state and simulating real-world attacks that can identify security flaws. Other vulnerabilities automated tools can identify are buffer overflows, insecure communications, insecure deserialization, and uncontrolled resource consumption. |
| **4 (10 Principles)** | In our next slide, we have 10 coding principles that are considered best practices for secure coding. For each principle, I will quickly go over its definition, purpose, and use cases. Under each principle, there are listed coding standards that align with the matched principle, and we will be going over each coding standard in the next slide.  \*The first principle is “Validate Input Data”, and it is the process of thoroughly checking data provided by external systems or users to ensure that the data satisfies expected formats and ranges before it is handled by the system. This principle should be applied whenever data is received from an external source like from file uploads, API calls, and user inputs, and it should be applied at the point of data entry into the system whether it's from the user interface level, API endpoint level, or database level. In order to prevent processing errors, system crashes, and security vulnerabilities like SQL injection attacks, various methods, such as format checks, range checks, and presence checks should be implemented to ensure data integrity and system security.  \*The second principle is Heed Compiler Warnings, which emphasizes the importance of reviewing warnings from the compiler because they can indicate potential issues that may lead to bugs or unexpected behaviors. Although a program can still run since the warnings are not errors, it is important to carefully inspect and understand compiler warnings during the compilation stage of every program, so that issues like possible loss of data due to type conversions, deprecation warnings, and mismatched types can be addressed and corrected. Other warnings like unused variables and unreachable code should also be attended to as it indicates a piece of code is not functioning as expected. Overall, any programming environment that provides compiler warnings should be inspected to prevent potential runtime errors, improve the reliability of the code, and enhance the overall quality of software.  \*The third principle is Architect and Design for Security Policies, which involves incorporating security considerations throughout the entire lifecycle of a system so that security becomes an integral part of a system rather than an addition that is applied at the end. By emphasizing security and identifying potential security risks during the initial stages of a system to its design, implementation, and maintenance stages, the likelihood of vulnerabilities can be significantly reduced while also making it easier to manage and maintain security. This can be done by implementing a combination of security best practices and industry standards throughout architectural design that address how user data is stored, how a system will handle authentication and authorization, and how a system will be updated. Examples of its application can involve selecting secure frameworks and components during the planning phase, conducting threat modeling exercises during the design phase, and regular security audits combined with vulnerability assessments during the maintenance phase.  \*The fourth principle, Keep it Simple, stresses the importance of simplicity in design because complex systems are harder to understand, harder to manage, and more likely to contain errors or vulnerabilities. By keeping systems and security measures simple throughout the entire lifecycle of a policy and in areas like the architecture, code, user interface, and security measures of a system, the overall efficiency and effectiveness of a system can be significantly improved. Overall, it is important to implement clear and understandable security protocols that utilize user-friendly tools and provide comprehensive training to its users so that everyone that interacts with the system is well-equipped to maintain its security.  \*The fifth principle is Default Deny, which refers to blocking all access by default and only allowing actions that are specifically stated based on a defined security policy. This principle is the opposite of Default Allow and considers all actions as potentially harmful, so an action is restricted unless explicitly allowed. This principle hinders usability by being more restrictive, but it helps to minimize the attack surface and reduce the risk of security breaches. It can be applied in network security where a firewall only allows specific types of traffic from trusted sources, system security where access controls grant specific privileges necessary for intended functions, and application security where specific inputs must match a defined pattern in order to be accepted.  \*The sixth principle, Adhering to the Principle of Least Privilege, means providing users with the minimum level of access necessary for them to complete their job functions. By restricting access and preventing users from accessing sensitive information or resources that they do not need for their duties, potential damages from security breaches are minimized, monitoring user activities is simplified, and the risk of unauthorized access by malicious attackers is reduced. Whether it is database management, network security, or system administration, access should be restricted to only what is necessary for a user to perform their specific role or task. For example, a graphic designer does not need the same level of system access as a network administrator, which can protect the system from potential threats if the designer’s account becomes compromised.  \*The seventh principle is Sanitizing Data Sent to Other Systems, which is the process of cleaning and preparing data before it is transferred to another system in order to protect the security and integrity of the receiving system. By scrubbing data and ensuring data does not contain malicious code, sensitive information, or any threat that can compromise a system whenever data is being transferred from one system to another, the data and system it interacts with can be protected along with preventing the disruption of the functionality of the system or the utility of the data. It can be performed during routine data backups, data migration, or when sharing data with third-party vendors, and it can be applied at points of data transfer like during file transfers at the database level or when moving data from one application to another. Common examples of data sanitization are data masking, data anonymization, pseudonymization, data deletion, and data scrambling.  \*The eight principle, Practicing Defense in Depth, involves using a layered approach to security where multiple security measures are implemented throughout various levels of a system so that if one security measure fails, others are in place to prevent the breach or mitigate its damages. This principle should be practiced at all times with any system that requires protection, and it can be applied at all levels of a system, such as physical, network, host, application, and data areas. For example, a company can encrypt sensitive data, require two-factor authentication for system access, implement a firewall to protect its network, and physically lock their server rooms to prevent unauthorized access to sensitive data.  \*The ninth principle is Using Effective Quality Assurance Techniques, which centralizes around applying techniques and various practices like integration testing, performance testing, and security testing throughout the software development lifecycle to improve the quality of a system and prevent defects that increase development costs. Quality Assurance techniques should be applied at every stage of the software development process and can include methods like code reviews where peers review code for errors or improvements, unit tests ensure code acts as expected, and system testing verifies an entire system meets the necessary requirements. For security, static and dynamic application security testing, risk assessment, audits, and penetration testing should be used to maintain the integrity, confidentiality, and availability of data within a system along with ensuring that the system is secure and free from any vulnerabilities that can be exploited by malicious attackers.  \*Finally, the tenth principle is Adopting a Secure Coding Standard, which is the practice of adhering to a specific set of rules and guidelines during the development of software to guarantee that code is free from vulnerabilities that can lead to security breaches. It should be applied throughout the software development lifecycle from the initial design phases to the final testing and deployment stages, and secure coding standards should be applied in the development environment along with the testing environment. Various companies like Microsoft, Google, and Apple provide guides for developers on secure coding standards that involve practices like implementing proper error handling, validating input, and using secure communication protocols. However, secure coding standards and practices should be used depending on the specific security requirements of a project and the nature of the application that is being developed.  Each principle has a coding standard that it can be applied to it, and we’ll go over each coding standard in the next slide. |
| **5 (10 Coding Standards)** | From this slide, we have a risk assessment table, where I’ve organized each coding standard in priority order with the highest priority on top.  The first coding standard that I also believe has the highest priority, is to Implement Parameterized queries because they ensure that malicious attackers cannot change the intent of a query or gain unauthorized access to our system that can result in data corruption and data loss. Inputs from users should always be treated strictly as data rather than SQL code, and this standard is ranked highest between the other coding standards because of how common SQL injection attacks are and the potential for damage that they can cause to the integrity and security of a system.  The second coding standard is to not access freed memory because it can lead to undefined behavior since the memory could have been reallocated and modified. Accessing and modifying freed memory can cause the program to crash, which can lead to data corruption or segmentation faults. It is ranked highly as a priority because it directly impacts the stability and reliability of software.  The third coding standard is to guarantee that storage for strings have sufficient space to hold character data and the null terminator so that errors like buffer overflows do not occur. Failure to ensure sufficient space for strings and their null terminators can cause system crashes or unpredictable behavior that malicious attackers can exploit to inject malicious code.  The fourth coding standard that is the last of the “high” priority coding standards is to always initialize variables, especially pointers, because segmentation faults can occur when pointers aren’t initialized. In general, each variable has a type that the compiler uses to determine the size and layout of a variable’s memory, and it’s important to initialize variables to ensure they start with a value that is appropriate for their data type. By initializing variables, it prevents the variable from being in an indeterminate state that contains a garbage value and can cause undefined behavior. It also improves code readability and helps avoid logical errors from unexpected or incorrect values.  The fifth coding standards transitions into the “Medium” priority standards and emphasizes that code should prevent integer overflow and underflow. Because unsigned integers have a fixed size, operations should not cause the value of an unsigned integer to go beyond its maximum limit. It’s important to use appropriate checks and bounds with the correct data types to prevent malicious attackers from exploiting integer wrapping and executing arbitrary code that can lead to unexpected results.  The sixth coding standard is to use range checking for element access. It is necessary to ensure that a string or other data structure is not accessed with an out-of-bounds index to prevent runtime errors and potential security vulnerabilities. A good practice is to use string methods to throw an out\_of\_range exception when an index is not within the range of a string. This prevents undefined behavior and potential system crashes.  The seventh and final coding standard for the medium priority standards...is to not modify the standard namespace. Modifying the standard namespace can cause undefined behavior, name clashes, portability issues, and make code harder to read and maintain. A best practice is to define your own namespaces and leave the standard namespace as it is so that standard library functionalities are not altered.  The eighth coding standard and start of the low priority standards... is to catch and handle all exceptions. All exceptions should be caught and handled to ensure program stability and maintainability. However, not all exceptions should be handled the same way, and they should be used appropriately based on their type and catch. Overall, though, unhandled exceptions can mask bugs and cause a program to terminate unexpectedly, which can lead to potential data loss and poor user experience.  The ninth coding standard, use constants over magic numbers, makes code easier to understand and reduces the chance of misinterpretation. Magic numbers can be misleading and can be prone to errors when needed to be modified. This coding standard is ranked low because it doesn’t directly create a vulnerability, and instead adds clarity so that code is easier to read, maintain, and debug. Overall, it’s better to use descriptive naming conventions that provide context for what numbers represent rather than magic numbers that appear without any explanation of their meaning.  The final and least priority coding standard among our 10 standards is to use assertions for programming errors and not runtime errors. Because assertions are designed for debugging and development purposes, they are ranked as a low priority since they don’t directly contribute to the robustness or security of a program in a production environment. They are typically disabled or ignored in production code, which makes it not as critical as other coding standards that can impact a program’s functionality or security. They are, however, still an excellent tool for catching bugs and logical errors in development, but they should not be used for error handling in a production environment.  Ultimately, my system of prioritization weighs heavily on the overall impact on the program’s functionality and performance, but it also considers factors like potential for damage, likelihood, and remediation cost. |
| **6 (Encryption)** | \*For this slide, we’ll be talking about encryption policies, where the three forms of encryption are \*at rest, \*at flight, and \*in use.  \*Encryption at rest refers to the process of encrypting data that is not being accessed or transferred into an unreadable format unless decrypted with a decryption key. The data is stored on a physical or virtual disk, and the data is encoded by using \*strong encryption algorithms that are generally categorized as either \*symmetric or asymmetric. Symmetric encryption is fastest and most efficient, which makes it suitable for encrypting large amounts of data, but asymmetric encryption is more secure as it has a public key that is used for encryption and can be known to everyone while also having a separate private key that can only be used by an individual that is decrypting the data. Overall, encryption at rest should be used to protect confidential information in case a storage medium is lost, stolen, or accessed by unauthorized individuals. It also makes it more difficult for attackers to access readable information, adheres to compliance requirements involving personal information, and builds trust between a company and its customers.  Encryption in flight, or encryption in transit, \*involves the process of encrypting data while it is being transferred from one location to another. Whether the data is transported across a network or between systems, encryption in flight’s purpose is to protect sensitive information from being intercepted or captured by a third party during the data’s transmission. It can be applied in practice through methods like \*Hypertext Transfer Protocol Secure (HTTPS), Secure Sockets Layer/Transport Layer Security (SSL/TLS), and a Virtual Private Network (VPN). Overall, these methods and secure file transfer protocols allow organizations to enhance their security when transporting data from one place to another.  Encryption in use, or runtime encryption, \*consists of encrypting data that is currently being accessed, read, updated, and processed in a computer’s memory. Because this is when data is immediately available and decrypted so the system can understand it, it is when data is most vulnerable. In practice, data should only be decrypted when necessary and should be re-encrypted as soon as possible to reduce the amount of time that the data is exposed. Secure computing environments like \*Trusted Execution Environments (TEEs) can be used to decrypt and process data securely as they reduce the risk of exposure by being isolated from the rest of a system. Another approach, especially for cloud computing environments where data might be processed on shared or potentially insecure systems, use of Secure Encrypted Virtualization (SEV) should be used to isolate the memory of each virtual machine from others running on the same physical server. Overall, these are common methods for achieving encryption in use, but a comprehensive approach that includes regular auditing, robust key management, and other industry standards are necessary to maintain the data’s security and protect it from malicious attackers. |
| **7 (Triple A)** | Our next slide is about Triple-A policies where... \*Authentication is the first step of the Triple-A framework, \*and it is the process of verifying the identity of a user, device, or system so that it can gain access to a network. The policy applies because authentication is the first line of defense against unauthorized access and assists with ensuring that the entity requesting access is who or what it claims to be. With regards to user logins, \*credentials like login information, biometric data, or digital certificates are verified against a database to ensure that only authorized users or systems can access the network. By establishing the identity of the entity requesting access, the system can enforce access controls and track activity accurately. Especially if a user needs to make changes to the database like update their user credentials, authentication must be performed to ensure that the user making changes is the rightful owner of the account. Just like with the addition of new users, \*unique credentials must be assigned to them and authenticated when they log into the system, so that the system knows what they are authorized to perform.  After a user has been authenticated, \*authorization follows, \*which involves the process of denying or granting the user access to specific resources or services. Ultimately, the process determines what the user, device, or system has the permission to do, and it is usually based on access control lists, role-based access controls, or other policy-based security mechanisms. Depending on the user’s role and responsibilities, \*authorization decides the user level of access an account may have to a system’s resources and what changes they are able to make to a database. Other methods of how authorization is applied is through Discretionary Access Control (DAC), Attribute-Based Access Control (ABAC), and Mandatory Access Control (MAC). Overall, the purpose of authorization is to enforce that each account can only do what they are supposed to do and nothing more, which emphasizes effective security management and aligns with the principle of least privilege.  Once users are authenticated and authorized, the process of \*accounting occurs, \*where the user is tracked and recorded based on their activities within a system. Data is collected about user activities \*like when they logged on, what files were accessed by users, the number of resources they’ve used, and what changes they may have made. After being collected, the data is securely stored and analyzed to identify patterns, detect potential security threats, and ensure compliance with policies. Reports can then be generated for purposes like auditing, planning, and billing. \*In the end, proper accounting improves security by detecting unusual behavior, resource management by tracking consumption, and overall planning by providing valuable insight on system details. When combined together with authentication and authorization, they form a comprehensive security framework that utilizes defense in depth best practices and makes it easier to maintain the integrity and security of a system. |
| **8 (Unit Test)** | Unit testing is an important process in the software development lifecycle. It helps with early bug detection, improves design, and can serve as documentation. More importantly, it ensures the functionality and performance of code, so that it leads to more robust, reliable, and efficient software. For our unit tests, I used the Google test framework for their rich set of assertions, among other features. To apply the unit testing framework, download and install Google Test, include its necessary headers in your code, and use the TEST macro to define test cases. Appropriate structure can be seen in the following slides, where each test involves compliant code situations that prevent an attacker from causing undefined behavior and crashes that allow them to ultimately perform unauthorized actions. |
| **9 (Test 1)** | For our first test, we have a function, get\_element, that has the potential to access an array out of bounds. To address the issue, we have the function throw an out-of-range exception whenever a passed index is less than zero....or equal to or greater than the array size. To test this, we create a vector consisting of the integers 1, 2, and 3, and verify that index 3 and index –1 throw an out of range error. Similarly, we create the same vector and verify that it is capable of returning valid indexes, with the correct values. To take this test a step further, we can test empty arrays and how the function behaves with large arrays and indices. |
| **10 (Test 2)** | Our second test checks that we can handle null pointer dereference properly. In this test, we can see that the class MyClass has a method called process that takes in a pointer. By having it check if the pointer is null, it can then throw an invalid\_argument exception to prevent the null pointer dereference from occurring. Our unit tests then check if the process function throws the exception when the pointer is null, and if it returns a valid pointer for a valid value like 10. To take this test a step further, we can experiment with dangling pointers, uninitialized pointers, and extreme input values to ensure they are handled correctly as well. |
| **11 (Test 3)** | For our third test, we created a copyString function that copies a string from a source pointer to a destination pointer. However, before it copies the string, it checks if the destination has enough space to hold the source. If not, it prints a buffer overflow error and returns without copying. In our first test case, we can see that a character array space is declared for 10 characters and that a source string, which says “This is a very long string” is much longer than 10 characters. After trying to copy the source string to the smaller character array, we expect an error output that notifies a buffer overflow was detected. In contrast, the next test case has a larger destination with a source string capable of being stored in it. We then verify that no error is outputted and that the destination string and source string are equal to each other. To take the test a step further, we can perform empty string tests and verify that the source string was unaltered. |
| **12 (Test 4)** | For our fourth test, we have a function that is used for division that takes two double’s for numbers. To prevent undefined behavior and vulnerabilities from dividing by zero, we have the function throw an invalid argument exception whenever the denominator is zero. For the first unit test, we check that it throws an error when the denominator is zero, and for the second unit test, we check that it performs the proper division for valid inputs. To take the test a step further, we can use extreme input values that can uncover potential issues that aren’t apparent in normal input values. Overall, unit tests are essential for verifying that expected outputs are the actual outputs, and that code is properly written to handle vulnerabilities that can be exploited. |
| **13 (DevSecOps)** | For our next topic, we have the DevSecOps pipeline, and the DevSecOps pipeline is an approach to software development and operations that integrates security at every stage of the software development process. It emphasizes a security by design culture, and rather than waiting at the end of a project to implement security, it considers it from the beginning. By implementing security best practices and fostering a culture that prioritizes security, DevSecOps seeks to proactively reduce the risk of security issues and make continuous security more efficient and effective. Looking at our DevSecOps diagram starting in pre-production, the first stage of assessment and planning already emphasizes DevSecOps’s dedication to security by defining threat landscapes and considering new threats. By identifying potential threats, analyzing attacker motives, and researching possible vulnerabilities through threat landscapes, organizations begin the process of proactively developing robust security measures that will be integrated throughout the entire development lifecycle. By acquiring a comprehensive understanding of the vulnerabilities in our system, development teams can better plan and implement effective mitigation strategies in the design phase. From the design phase, architectural patterns that fulfill the project requirements along with tests that satisfy previously defined security requirements are conducted until the code is compiled during the build phase. Once code from different developers is integrated and compiled into an executable program, conflicts are resolved, and initial checks are performed to prepare the system for thorough testing. From there, the verify and test stage can begin, where numerous tests like integration, system, and security tests are conducted. At this time, static application security testing (SAST) and dynamic application security testing (DAST) tools are used to identify security vulnerabilities and potential areas for security incidents.  After pre-production is complete with a deployable project to be released, the transition and health check stage begins. With the help of automated tools, teams identify and fix any issues before the software is moved from the development environment to the production environment. Security settings are configured, and penetration testing is conducted to ensure the system is secure against potential attacks. The next stage is monitor and detect, where continuous monitoring is performed to ensure the software is functioning as expected. Activities can include vulnerability scanning, intrusion detection, and anomaly detection. The respond stage follows, which involves taking appropriate actions to address any identified security issues and developing incident response plans to enhance the team’s security posture and resilience against future attacks. After, the maintain and stabilize stage begins where teams ensure the continuous and secure operation of the system by making necessary adjustments that maintain its stability and security. Activities can range from updates and patches, to balancing system resources, to adjusting security controls. |
| **14 (Tools)** | Throughout the DevSecOps process, automated tools play a crucial role in streamlining security checks and maintaining the security of an application. They not only reduce the risk of human error but also allows for faster detection and response to security threats. Ultimately, the goal of DevSecOps is to integrate security into the development lifecycle at the same speed and scale as decisions and actions from development and operations. Examples of these tools starting from \*pre-production and the assess and plan stage are:  \*Microsoft Threat Modeling Tool and OWASP Threat Dragon, where companies can evaluate the current state of their system and provide risk analysis for potential security threats. In the \*design phase, policy as code frameworks like Open Policy Agent (OPA) can be used to automatically enforce design policies related to access control, authorization, and other service-level policies. For the \*build phase, automated tools like Nexus Repository OSS can be used to manage trusted repositories and contribute to a secure build environment. \*CPPCheck and Coverity can then be applied as static application security testing (SAST) tools to scan the codebase for security flaws. Once the Verify and Test Phase begins, dynamic application security testing (DAST) tools like OWASP ZAP and Burp Suite can identify runtime vulnerabilities in the code.  For the \*Production phase, penetration testing can be performed, and configuration management tools like \*Chef can be used to automate the deployment of C++ applications with secure configurations. \*For the monitor and detect phase, intrusion detection systems (IDS) and Security Information and Event Management (SIEM) tools can monitor and analyze logs for unusual activities that could indicate a security breach. \*Automated response tools like IBM Resilient or any other Security Orchestration, Automation, and Response (SOAR) tools can then be used to help respond to incidents and minimize their potential damages. \*Finally, in the maintain and stabilize phase, backup and recovery tools like Rubrik along with change management and monitoring tools like GitLab should be used to keep the production environment stable and secure. |
| **15 (Risks)** | When deciding on whether to act now or to wait to implement a security policy, each approach has its own problems, risks, benefits, and solutions.  The potential \*problems for acting now on implementing a security policy can include resource allocation, disruption of operations, and resistance from employees. Implementing and enforcing a security policy is a complex process that requires resources like time, money, and personnel. Acting immediately can strain these resources, which can lead to \*risks like inefficiencies and potential mistakes. However, \*the benefits are a proactive approach to preventing threats, which makes the overall system more secure. \*Solutions to these problems are to properly and efficiently plan for their implementation, clearly communicate the reasons for the new security policy, and to provide regular training to employees so that they are up to date and comply with new policies.  The \*potential problems for waiting to implement a security policy, are that the organization is much more vulnerable to threats and the risks associated with them. \*Other risks involve non-compliance with regulations and the possible additional costs for implementing the policy later. However, \*waiting includes benefits like having more time to develop a comprehensive strategy and aligning it with business growth and changes. To \*solve these problems, it’s important to develop a strong security policy as soon as possible. If time is a major constraint, a basic policy that addresses immediate threats will be better than no policy. |
| **16 (Recommendations)** | After reviewing the current state of our security policy as part of conducting a gap analysis, I recommend that we create a more comprehensive policy that goes \*more in depth and adds to the 10 principles and 10 coding standards that were previously stated. Areas that we can improve on are implementing security measures for other layers in our defense in depth illustration, detailing processes to update outdated practices, and establishing more rigorous incident response plans. A real-world example would be how companies now need to adjust to remote work as more positions for it open up, which would require addressing new gaps like handling the increased demand for remote access. Necessary improvements would be to invest in cloud-based solutions and stronger cybersecurity measures. \*By implementing a more robust system for monitoring security events, conducting more \*thorough security audits, and improving \*incident response plans, we can close the gaps in our current security infrastructure, enhance our ability to detect and respond to threats promptly, and ultimately strengthen the overall security posture of our organization. |
| **17 (Conclusion)** | In conclusion, changes and adjustments are always going to be necessary, which is why it is important to be guided by well-known and established standards. Standards that should be adopted to prevent future problems should come from institutions like the International Organization for Standardization (ISO) and the National Institute of Standards and Technology (NIST). They provide examples that exemplify the desired state of our security policy through policies like ISO 27001 and NIST SP 800-53. However, although they serve as excellent models for developing a robust and effective security policy, our policy will still be tailored more towards our organization’s specific needs. Overall, the key to a successful security policy lies in the balance between adopting established standards and customizing them to fit the unique requirements of our organization. This ensures that our security measures are comprehensive, relevant, and makes us equipped to handle any security challenges that come our way. |