Malik Alnakhaleh

CS-405

Professor Joseph M. Conlan

Project Two

April 29, 2024

**Project Two Script**

Hello, everyone. Welcome to today's security policy presentation. I am a developer at Green Pace, my organization. My name is Malik Alnakhaleh. And today I am going to be going over the security policy presentation on behalf of my organization. For context, I did craft this security policy based on the information provided about our secure coding best practices at Green Pace. And we're just going to take some time going through the findings from said security policy. All right. So an overview of the policy will include a variety of different tactics as it relates to defense in depth. So defense in depth will be employed and supported throughout the security policy where the document ultimately is being used to establish a structured framework of coding standards, risk assessments, and automated detection methods that reinforce a multilayer defense strategy. Through the integration of these different elements, Green Pace is ensuring that security is woven into the fabric of software development and that our software development processes are supporting defense in depth best practices, which ultimately will enhance system resilience against potential security vulnerabilities as well as threats. And on the subject of threats, we have our threat matrix here that outlines each of the security vulnerability coding standards that we had determined worth investigating, including the potential issue with coding standards around strings through addressing things like buffer overflows, which are critical due to the potential to overwrite memory leading to crashes or code execution vulnerabilities. We also have null pointer vulnerabilities and these can lead to crashes or undefined behavior posing a significant risk to application stability and security. We have SQL injection vulnerabilities. So these are classic severe security risks that can often lead to unauthorized data access or manipulation. We additionally have memory allocation errors, which can include misused pointers that would lead to unpredictable behavior and potential security risks. We have invalid pointer to member access, which involves misusing pointers that can lead to unpredictable behavior and potential security risks, the enumeration range issues. So out of range values that can cause unpredictable behavior, increasing the risk of bugs and vulnerabilities, random number seeding. So improperly seeding number generators, which can compromise the unpredictability element of different security mechanisms associated with random number generation, exceptions before the main execution of a given program. So handling exceptions properly before the main method executes, which is ultimately crucial to prevent things like crashing during program startup, different types of copy mutations. So ensuring that copy operations don't alter the source for predicting behavior and the maintaining integrity of an application's different copied values, running sensitive data exposure. So information leakage, where we're ultimately trying to avoid transferring objects across different security contexts, which is essential for confidentiality. And ultimately that this matrix is prioritizing these different remediation efforts as they relate to their likelihood or unlikelihood, as well as their potential priority. So in addition to that, we also have engaged in 10 principles associated with our software development coding standards. So we want to make sure we have validated input data related to any data types, data values, string correctness, SQL injections, query protection, assertions, and other miscellaneous potential vulnerabilities or coding standard issues that can arise. In addition to heating compiler warnings with exceptions and object-oriented programming, shortened to OOP for short, architect, and design for security policies. We want to keep it ultimately simple, default deny. We want to adhere to the principle of least privilege with avoiding things like SQL injections and insertions, sanitizing data sent to other systems, such as through data type, data value, string correctness, SQL injection and assertions, practicing defense in depth with memory protections, assertions, and other miscellaneous standards, as well as using effective quality assurance techniques for memory protection exceptions and object-oriented programming and adopting a secure coding standard or data type string correctness exceptions and object-oriented programming. In addition, we have particular coding standards that were mentioned earlier that were ordered and prioritized based on severity, likelihood, and mediation costs, which in that order. So in the cases of ties within the same category, the next category is looked at with lower remediation costs being prioritized over high ones as well as higher severity and likelihood being prioritized over lower in both cases. So as you can see, we have the data type, the string correctness, SQL injections, memory protection, object-oriented programming, all having high likelihood being at the top of the prioritization order. In addition to other miscellaneous coding vulnerabilities that were discovered in some of our programs, in addition to data values, exceptions and some other OOP and assertion issues put at the bottom of that list. We have additional encryption policies based on the new security policy as it relates to encryption of data at rest in flight and in use. So in the case of encryption at rest, we want to adhere to the principle of least privilege to confirm that only authorized users and processes can access the encrypted data at rest. We want to practice, we want to be doing our best to practicing defense in depth by implementing multiple layers of encryption and access controls that enhance the overall security and data at rest. Adapting a secure coding standard that mandates the use of strong encryption algorithms and proper key management practices to help protect data at rest from unauthorized access breaches. So for encryption in flight, we also want to validate our data and ensure that the data being encrypted and transmitted is free from malicious content or unauthorized modifications. Sanitizing data sent to other systems during that flight process by applying appropriate encryption techniques, which ultimately helps protect confidentiality, integrity of the data in transit, as well as practicing defense in depth by using secure communication protocols such as TLS or SSL and implementing additional security measures like message authentication codes, which ultimately enhances the overall security of the data in flight. And furthermore, when we are dealing with data encryption at use or in use, we should be adhering to the principle of least privilege to ensure that only authorized processes and users can access and manipulate the unencrypted data in use. Practicing defense in depth by implementing secure memory management techniques such as secure enclaves or hardware-based encryption, which ultimately helps protect the data in use from unauthorized access or tampering, as well as adopting a secure coding standard that emphasizes secure memory handling, side channel attack mitigation, and proper key management practices, which helps to safeguard the data in use and maintains confidentiality and integrity. We also follow particular Triple-A policies as it relates to authentication, authorization, and accounting. In the case of authentication, we want to be validating input data during the authentication process, such as through user logins, which would help the prevention of unauthorized access and would ensure the integrity of the authentication mechanism of our given devices and systems, as well as practicing defense in depth by implementing multiple factors of authentication such as password and biometric data, which would enhance the overall security of authentication process. Using effective quality assurance techniques like penetration testing and code reviews to help in identifying and addressing vulnerabilities in the authentication system, including the addition of new users. Furthermore, as it relates to authorization, we want to be adhering to the principle of least privilege to ensure that users are granted only the necessary permissions to perform their tasks, limiting the potential impact of unauthorized access or misuse. Validating input data such as user-supplied information for accessing files or modifying the database, which ultimately helps prevent unauthorized changes and maintaining the integrity of the system. Sanitizing data sent to other systems like databases or file servers, which ensures that only authorized and safe operations are performed. Mitigating the risk of data corruption and unauthorized modifications, as well as accounting as it relates to practicing defense in depth by implementing multiple layers of logging and monitoring, such as application level logging and system level auditing, which enhances the security of the overall application or system, as well as the overall visibility and accountability of user activities, as well as using effective quality assurance techniques like log analysis and anomaly detection, which helps to identify suspicious or unauthorized activities such as unusual file access patterns or unauthorized changes to the database, as well as adopting a secure coding standard that emphasizes proper logging, error handling, and secure data management practices, and ensures that the accounting mechanism captures relevant events and maintains the integrity of the audit trail facilitating incident response and forensic analysis. Additional unit testing was done in the form of SQL injection testing. This is a simple example from one of our Java applications that was applied. Normally, I would be hashing this password, but we wanted to focus on compliant code as it relates to using parameterized queries versus direct concatenation, which could lead to a SQL injection. So in the case of the unit test provided, we did a negative direct concatenation test, which passed. We also did a positive input validation, which also passed. A negative escaped input, which passed as well. And the only test that didn't pass was the positive parameterized query, but that was due to a separate database issue of the table not being properly created. So technically that did fail and that will be something we would need to address on our end, but that is something that we are currently in the process of addressing as it relates to that particular example of compliant versus non-compliant code. Additionally, we're currently working on transitioning from a DevOps pipeline to a DevSecOps pipeline. What that ultimately looks like is the integration of security practices within the entire DevOps process using the following tools that we have, such as Jenkins, SonarQube and Contrast and how they all fit in within the DevSecOps pipeline. So within that pipeline, we have the assess and plan step where Jenkins can be used to automate tasks for regulatory change and checking on the backlog prioritization, utilizing plugins to scan for new threats. SonarQube integrates in the design process with the design phase to ensure secure coding practices and adherence to the OWASP guidelines that provides continuous inspection of code quality. We also have in the build phase, Jenkins automations of the build processes, which ultimately ensures secure reproducible builds from trusted repositories. We have the verification and test phase where SonarQube is used again to assess code quality and security before deployment, where Contrast additionally provides interactive application security testing or IAST testing by working within the application, identifying vulnerabilities as they interact with data and control flows, transitioning data health check or transitioning in health check phase. That in that phase, we have Jenkins, which would automate the deployment integration with tools for configuration and security settings while Contrast conducts security testing and pre-production. We would also have monitoring and detecting phase where SonarQube may contribute to static analysis during runtime, which we do currently have it doing that. In addition to Jenkins, which would integrate the security information and event management for log collection and event alerting. During the respond phase, we'd have Contrast real-time monitoring, which allows for the immediate response to security threats and facilitating quick decisions to block attacks or rollback changes. And finally, we have the maintain and stabilize phase, which is where we use Jenkins to support the continuous monitoring process of our application to maintain system stability with Contrast additionally securing runtime security and confirming that is up to our baseline standards and requirements. We have different risks and benefits associated with the application or the applying of the security policy, including acting now where we have benefits that improve the immediate security posture and ensures our early detection of vulnerabilities and integration of security into the development lifecycle. We have rapid changes that might lead to disruptions in the current projects and require extensive training. We could potentially wait, which would allow for more time through planning and phased implementation, but would also lead to continued vulnerability to security threats and potential non-compliance within the for by industry standards. Furthermore, we have potential strategy gaps where we have a lack of specificity in the general guidelines without particular specific actionable items for benchmarks. We also do not have sufficient tool integration potentially through a lack of detailed steps on how tools should be configured or used within our DevOps pipeline in our transition to DevSecOps. So the recommended steps for these potential risks and benefits would be to refine our coding standards to fill in all the blanks within the coding standards with specific examples. We would also want to quantify risk assessments, so attaching specific metrics and scenarios to our risk levels. We would want to have a detailed automation integration process, so providing step-by-step instruction guidelines for each tool within the DevOps and DevSecOps workflows as we transition. We would also require regular ongoing security training for the development teams to adapt and update policies as we go through the transition to establish a regular review process for security policy to adapt to new threats and technologies as they arise. So furthermore, we have additional outstanding gaps in security policy that include a lack of specific incidence response protocols, which ultimately means that the policy lacks detailed steps for responding to security breaches. We also don't really, without these specific guidelines, have any potential to respond to incidents in a way that isn't potentially delayed or ineffective. We also have incomplete coverage of mobile and remote work environments, given that we have a hybrid working policy. The current policy does not currently explicitly address security for remote or mobile work scenarios, which would lead to a potential increased risk of data breaches due to unsecured remote access points. Furthermore, we lack a continuous improvement mechanism, so this policy ultimately does not specify the mechanisms for ongoing evaluation and updating this evaluation based on new threats, as well as, as a result of that, the security framework could become obsolete as new threats emerge. So in conclusion, we primarily wanted to focus on the SQL injection prevention, so the essentials to safeguard databases from unauthorized access to data manipulation. We also wanted to have proper memory allocation handling within most of our C++ apps to prevent crashing and security vulnerabilities due to insufficient system resources. And finally, we wanted to avoid information leakage when passing class objects across trust boundaries, which protects sensitive data from being exposed unintentionally. Thank you all very much for your time.