# Denis Dzenyuy

4/13/2023

# CS 405 Project Two Script Template

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hello everyone, my name is Denis Dzenyuy in this presentation I will be explaining the outline security policy guide template for Green Pace. The goal here is for Software security at Green Pace to require a consistent implementation of security principles that will apply to all developed applications. This means consistent approaches and methodologies should be maintained through all policies that are implemented and maintained over time. |
| **2** | Here is a defense in-depth strategy overview. This is a representation of the core security process of utilizing multiple layers or redundancies of protection in the event that one or more of the layers of defense is breached. Adopting a secure coding standard as early on as the ideation, and/or no later than the requirements phase, and throughout the SDLC will ensure that strong defense mechanisms are in place from the start.  This will include keeping servers under lock and key on premises with only privileged users able to access them. In the Supplier cloud, ensure privacy level agreements and service level agreements are in place. Within our own private cloud, establish a baseline of deny-by-default configurations and defense in depth strategies through external penetration tests and tried and true encryption mechanisms, as well as enabling Firewall and privileged account management (PAM), and Monitoring for unauthorized changes and access through logging. Vulnerability scans, patch management techniques, and updates of vulnerable APIs can be integrated into the CI/CD pipeline and automated.  By protecting our host security, we protect the critical assets of the system and data that attackers seek. Lastly, training of the team on the layered defense compliance protocols is critical. |
| **3**  **3** | 3The threat matrix is an action-oriented view to safeguard the system. It is a guide that helps in trying to locate, classify and ultimately address threats based on their respective degree of potential impact.  As showed here, there are four basic categories in which threats can be classified, from low, unlikely, likely, and high priority. High priority threats require swift and robust action.  Going from top left where we have our most likely threat; INT30-C which is to ensure that unsigned integer operations do not wrap. This behavior is more informally called unsigned integer wrapping. Unsigned integer operations can wrap if the resulting value cannot be represented by the underlying representation of the integer. In the bottom right we have the most unlike which is DCL12-C which refers to implementing abstract data types using opaque types. The use of opaque abstract data types, though not essential to secure programming, can significantly reduce the number of defects and vulnerabilities introduced in code, particularly during ongoing maintenance. The low priority threats (in the bottom left corner) ERR06-P4 which refers to understanding the termination behavior of assert() and abort(). Unsafe use of abort() may leave files written in an inconsistent state. It may also leave sensitive temporary files on the file system. Then on the opposite end of the (threat) spectrum, we have our high priority, STR38-C-P27 Do not confuse narrow and wide character strings and functions. Confusing narrow and wide character strings can result in buffer overflows, data truncation, and other defects.  All these threats are important, but they should be prioritized based on potential level of severity of impact to the system, and high priority threats should always take precedence over all other types or levels of threats. |
| **4**  **4** | 10 principles and the coding standards that apply to each policy. The objective here is to demonstrate the alignment between principles and standards.   1. Validate Input Data (STD 001, 002, 003, 004). We want to ensure data entering the workflow is correct. Validation of inputs on all trusted or untrusted sources ensures vulnerabilities or malicious data are eliminated from entering the system. This includes awareness of all data sources and types, especially external data. 2. Heed Compiler Warnings (STD 002, 003, 010). Compiler warming while they may seem “annoying” It is important not to ignore any warnings when developing code. All warnings must be addressed accordingly. When warnings are left unaddressed, security flaws persist, and this could have adverse effects. 3. Architect for Security Policies (STD 002, 004, 005, 006, 007, 008, 009, 010) When designing your software architecture, it is very important that you follow, adhere to, and address the security policies while still meeting the requirements for what the code is set to accomplish. 4. Keep it Simple (ALL) When designing your code, it is important to keep your code as clean, simple, and as readable as possible. Making code complex unnecessarily leaves room for avoidable errors that could lead to vulnerabilities. Simple code = simple fix. 5. Default Deny (STD 004) Standardize processes to where any deviation should/would always result in a (deliberate) denial. Denial as a standard guarantee’s that any instance of change or deviation will trigger and/or warrant suspicion of potentially malicious activity. 6. Adhere to Principle of Least Privilege (STD 004, 009, 010) Processes should be time boxed, such that access is limited to the time allotted for defined tasks, and only to those assigned to complete the specific task. This should apply to all processes including those that require higher access to complete the same task. Reducing access time reduces the chance of vulnerability. 7. Sanitize Data Sent to Other Systems (STD 003, 004). Sanitization should be in place to handle injection attacks. Filtering the flow of (incoming) data will ensure it is safe to be transmitted to/or through the system(s). 8. Practice Defense in Depth (STD 004). Combining layers of redundant and varying defense can help safeguard a system. This ensures that in the event one layer fails, another layer is in place to help defend against attacks. 9. Use Effective Quality Assurance Techniques (STD 001, 002, 003, 005, 006, 007, 008, 010). A properly working security protocol or system should include a process whereby there is testing, reviewing and an outside point of view. Quality assurance is huge of every process as they help find issues with any process/design. Finding issues by someone in the team is always better than being found by a customer or a malicious actor. 10. Adopt a Secure Coding Standard (STD 005, 006, 007, 008, 009, 010). Always emphasize and implement a secure coding standard that can be followed easily and aligns well with the language you are using. Designing code with security policies in mind and effective quality assurance techniques in forefront will prevent (costly) problems that may not come up until later down the road. |
| **5** | 1. Implement abstract data types using opaque types. The use of opaque abstract data types, though not essential to secure programming, can significantly reduce the number of defects and vulnerabilities introduced in code, particularly during ongoing maintenance. 2. Ensure that unsigned integer operations do not wrap. This behavior is more informally called unsigned integer wrapping. Unsigned integer operations can wrap if the resulting value cannot be represented by the underlying representation of the integer. 3. Do not attempt to modify string literals. Modifying string literals can lead to abnormal program termination and possibly denial-of-service attacks. 4. Sanitize data passed to complex subsystems. Failure to sanitize data passed to a complex subsystem can lead to an injection attack, data integrity issues, and a loss of sensitive data. 5. Explicitly construct and destruct objects when manually managing objects lifetime. Failing to properly construct or destroy an object leaves its internal state inconsistent, which can result in undefined behavior and accidental information exposure. 6. Understand the termination behavior of assert() and abort(). Unsafe use of abort() may leave files written in an inconsistent state. It may also leave sensitive temporary files on the file system. 7. Handle all exceptions. Allowing the application to abnormally terminate can lead to resources not being freed, closed, and so on. It is frequently a vector for denial-of-service attacks. 8. Do not delete a polymorphic object without a virtual destructor. Attempting to destruct a polymorphic object that does not have a virtual destructor declared results in undefined behavior. In practice, potential consequences include abnormal program termination and memory leaks. 9. Use correct integer precisions. Mistaking an integer's size for its precision can permit invalid precision arguments to operations such as bitwise shifts, resulting in undefined behavior. 10. Do not confuse narrow and wide character strings and functions. Padding bits contribute to the integer's size, but not to its precision. Consequently, inferring the precision of an integer type from its size may result in too large a value, which can then lead to incorrect assumptions about the numeric range of these types. Programmers should use correct integer precisions in their code, and in particular, should not use the sizeof operator to compute the precision of an integer type on architectures that use padding bits or in strictly conforming (that is, portable) programs. Mistaking an integer's size for its precision can permit invalid precision arguments to operations such as bitwise shifts, resulting in undefined behavior. |
| **6**  **6** | **Encryption IN FLIGHT:** Use up to date, secure libraries, use Public Key infrastructure for end-to-end protection on message bodies or attachments, use Managed File Transfer or SSH with expiration date on the link, password access, utilize Data leak prevention mechanisms built into cloud service.  **Encryption AT REST:** We should always encrypt data at rest using full-disk encryption at the server level, as well as database encryption using MySQL Server, and provide a backup strategy. This applies to both SDD’s and backup media like jukeboxes, stackloaders etc.  **Encryption IN USE:** Utilize identity management mechanisms to confirm user roles and identity, allow conditional access to the tools functionality based on the user roles and other parameters. Use IRM digital rights management, to apply persistent protection to documentation. |
| **7** | **Authentication:** Process of identifying a user, using valid credentials of user and password. Control how a user is authenticated using a secured local database or external AWS server, prefer to use tried and trusted protocol.  **Authorization:** After the user has been authenticated, authorization shall be used to determine which resources and functionality the user is allowed to access, and which operations can be performed.  **Accounting:** This refers to the process of monitoring and logging any user events while they are logging in/out or utilizing the resources, as well as user uptime or any other configured parameter to meet security and compliance rules. |
| **8** | Unit tests is generally a way to ensure that each unit of a software product works as intended. In most programming languages, that is a function, a subroutine, a method or property. Here we are going to demonstrate four coding vulnerabilities from four mixed tests comprising positive and negative results. At Green Pace we will be using google test framework for unit tests, reason being, google allows for independent and repeatable tests. We will be testing for out of bounds vulnerabilities (STD-001 DataValues, and STD-005 Memory Protection)  Testing for out of bounds vulnerabilities (STD-001 DataValues, STD-005 Memory Protection)  Google Primer guidelines to be adhered to: ASERT\_\* versions generate fatal failures when they fail and abort the current function.  **Test 1**.  Verify adding 5 values to the collection. |
| **9**  **9** | **Test 2.** ASSERT\_TRUE(collection=>max\_size()>=10);  Verify that the max size is greater than or equal to size for certain values ( 0, 1, 5, 10) entries.  Our second test will be appropriately named Max greater than or equal to size, since we’re going to be testing if the max size of this collection is greater than or equal to a certain number of preset values. This is done by adding 11 into the add\_entries function which enters 11 random values into the collection. We then assert if the max size of the collection is greater than or equal to 0, 1, 5, & 10. This is to ensure that there is coherence in the data structure. |
| **10** | **Test 3.** EXPECT\_THROW(elements.at(10), std::out\_of\_range);  Expect an out of bounds error to occur if calling an out of range element.  For our third test, we are looking for a throw error at a certain out of bounds element access. That will be an out\_of\_range error. We can also expect an Out-of-bounds error to occur if calling an out\_of\_range element. Having defined a vector of size 9 elements we can expect an out\_of\_range exception to be thrown if we call the elements an indices 10 since we only have 9 elements. |
| **11** | **Test 4.** ASSERT\_TRUE (Collection=>Size();  By asserting whether the size of the container is greater than the initial size. We have the option to verify the resize has either a decreased or increased the collection. Verify resizing decreases the collection. |
| **12** | **Test 4.** ASSERT\_TRUE (Collection=>Size() ==0; Check that the resize has indeed decreased the collection to zero.  For our last test we will assert if the resize has decreased the collection to zero. First, we add 15 elements randomly, set variable initial size, size of collection. After resizing to zero, we can assert if the collection size has decreased to 0, and if that is true then we can conclude that the resize has worked. |
| **13**  **13** | This diagram shows the driving direction behind enforcement and compliance.  If software security is implemented correctly, it can be automated throughout the SDLC. Automation encourages team workflow, and can use Docker for container instances, GitLab for versioning, Jenkins for CI.    Monitoring and detection. Automate static application security tests into nightly builds on key sections of code. Embed dynamic application security testing into SDLC to look for vulnerabilities in real time. |
| **14** | Automation can be created upon Build, by automating manual processes such as compilation and static code checking into a CI/CD pipeline.  DevSecOps Pipeline is a CI/CD pipeline with integrated security practices and tooling. It adds practices and functions like scanning, threat intelligence, policy enforcement, static analysis, and compliance validation to the software development lifecycle.  External tools, can be used to verify and test in SecOps through automation of virtualized container deployment. Implementation of automated security tests and regression tests can be done in QA.  Tools such as OWASP can be used for Dependency-Check to check code dependency vulnerabilities.  Tools include: Cppchecker, ClangTidy, CodeSonar, Coverity, Docker, Gitlab, Jenkins, OWASP ZAP, Parasoft, Veracode.  Some of these tools can be used or are rather best for postproduction penetration testing. |
| **15** | DevSecOps Pipeline is a CI/CD pipeline with integrated security practices and tooling. It adds practices and functions like scanning, threat intelligence, policy enforcement, static analysis, and compliance validation to the software development lifecycle.  External tools can be used to verify and test in SecOps through automation of virtualized container deployment. Implementation of automated security tests and regression tests can be done in QA.  Utilize tools such as OWASP Dependency-Check to check code dependency vulnerabilities.  Tools include: Cppchecker, ClangTidy, CodeSonar, Coverity, Docker, Gitlab, Jenkins, OWASP ZAP, Parasoft, Veracode. |
| **16** | A software application can only be as secure as the tools being used to implement security policies and guidelines. A good approach toward enforcement and compliance is to have a more comprehensive approach. This can be done in several different ways including:  Implementing enforcement policies for each (SDLC) phase. It is often very challenging to implement security policies, practices and protocols that apply to every step of the development process. Determine who, what, when, where, why, and how policies and procedures apply (in the development, testing, release, and maintenance phases).  Cloud Security should be well elaborated in terms of approaches of “defense in depth” at each layer of security  Room for unaddressed concerns should be implemented as part of each phase of the process.  Security Layers include; Critical Assets + Data, App Security, Endpoint Security, Host Security, network security, perimeter security, cloud security, and physical security.  Adopt a set of tools/options/mechanisms for Cloud Security.  Identify a set of optimal (least privilege) security controls for network and host security.  Third party security audit should be a consideration. |
| **17** | Ensure coding standards and practices are well defined, consistent and implemented.  Ensure Defense in Depth strategies are followed, and security policies are followed and reviewed promptly.  Adopt a set of tools/options/mechanisms for Cloud Security  Identify a set of optimal (least privilege) security controls for network and host security.    Secure coding standards should top priority and better yet second nature to the development teams. |
| **18** | [Provide APA-style references with links to resources, articles, and videos that you used in your presentation.]   * Admin. Last modified by Svoboda. D. December 05, 2018. SEI CERT C Coding Standard. Carnegie Mellon University,   Retrieved April 4, 2023, from: [SEI CERT C Coding Standard - SEI CERT C Coding Standard - Confluence (cmu.edu)](https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard)   * Seker. E. September 30, 2023. Medium. Defense-in-Depth   Retrieved April 4, 2023, from: <https://medium.datadriveninvestor.com/defense-in-depth-d6c070eac12d>   * Code Signing. (ND). Secure SDLC: A Look at the Secure Software Development Life Cycle   Retrieved April 4, 2023, from: <https://codesigningstore.com/secure-software-development-life-cycle-sdlc> |