# CS 405 Project Two Script Template

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hello, my name is Daniel Coffey and I’m a software developer at Green Pace. Today, I will be presenting our brand new policy for the development team. |
| **2** | * Defense in Depth (DiD) is an approach to cybersecurity in which a series of defensive mechanisms are layered in order to protect valuable data and information. * If one mechanism fails, another steps up immediately to prevent an attack. * This multi-layered approach with intentional redundancies increases the security of a system as a whole and addresses many different attack vectors. |
| **3** | * The threat matrix can be utilized to prioritize threats. * The order of urgency goes from priority, to likely, to low priority, and finally to unlikely. * 10 Standards:   + 5 – Likely   + 3 – low priority   + 2 – unlikely |
| **4** | * 1. Do not cast an out-of-range enumeration value * 2. Do not rely on the value of a moved from object * 3. Do not attempt to create a std::string from a null pointer * 4. Do not store already-owned pointer value in an unrelated smart pointer * 5. Do not access freed memory * 6. Use a static assertion to test the value of a constant expression * 7. Do not let exceptions escape from destructors and deallocation functions * 8. Store a new value in pointers immediately after free() * 9. Be careful when using functions that use file names for identification * 10. Do not make assumptions about the size of an environment variable |
| **5** | * 1. Validate Input Data * 2. Heed Compiler Warnings * 3. Architect and Design for Security Policies * 4. Keep it Simple * 5. Default Deny * 6. Adhere to the Principle of Least Privilege * 7. Sanitize Data Sent to Other Systems * 8. Practice Defense in Depth * 9. Use Effective Quality Assurance Techniques * 10. Adopt a Secure Coding Standard |
| **6** | * Encryption in flight – Encrypting data in-flight means that you encrypt data when it’s being transmitted over a network. Regarding some applications, data that is in rest may be unencrypted, but it gets encrypted while it is being communicated to provide proper defense. Encryption at rest – refers to encrypting the data on the disk. * Encryption at rest - Encryption at rest is designed to prevent the attacker from accessing the unencrypted data by ensuring the data is encrypted when on disk. If an attacker obtains a hard drive with encrypted data but not the encryption keys, the attacker must defeat the encryption to read the data. * Encryption in use – Compromising data in use enables access to encrypted data at rest and data in motion. For example, someone with access to random access memory can parse that memory to locate the encryption key for data at rest. Once they have obtained that encryption key, they can decrypt encrypted data at rest. |
| **7** | * Authentication is the process of verifying one’s identity, and it takes place when subjects present suitable credentials to do so. When a user enters the right password with a username, for example, the password verifies that the user is the owner of the username. Essentially, authentication establishes the validity of a claimed identity. * Authorization is a security technique for determining a user’s privileges or eligibility to execute specific tasks in a system. The authorization procedure specifies the role-based powers a user can have in the system after they have been authenticated as an eligible candidate. * Accounting monitors the resources a user consumes during network access. This can include the amount of system time, or the amount of data sent and received during a session. |
| **8** | Unit Test Names: Define all unit test names to appropriately reflect the test condition.  Unit Testing: Successfully implement the 13 unit tests, as part of the Google Test fixture; run Google Test ASSERT and EXPECT functionality to prove the tests. Each test you run must explicitly prove the defined condition of the test.  Negative Unit Tests: Complete at least 2 of the unit tests as negative tests that demonstrate capturing the appropriate unit test result based on an expected negative result of the test code.  C/C++ Program Functionality and Best Practices: Demonstrate industry standard best practices, including in-line comments and appropriate naming conventions to enhance readability of code. Develop functional C/C++ code that illustrates a software design pattern approach. |
| **9** | Automation is used to continuously monitor the network for unexpected changes in the configurations. As you can see from the picture provided that during pre-productions there are 4 things that will be automated. One of these is Assess and plan, what this does is take care of the threat landscape, regulatory changes, change impact analysis, prioritize the backlog, and respond to new threats. The next step is to take care of the design. This is test-driven design, that follows application best practices. The next step is to build securely utilizing trusted repositories and secure open-source usage. The last step in pre-production is to verify and test. This includes vulnerability scanning, trusted sources, and functional, compliancy and security testing.  The production side gets automated as well. It will start with a transition and health check. This deals with configurations and deployment, security settings, and penetration testing. The next step is monitor and detect, which takes care of log collection, SIEM, analytics, event alerting, and intrusion detection. From here a response is automated. This includes blocking attacks, turning off services, and roll backs. Lastly, maintain and stabilize. This assesses against security baseline, helps return to baseline, and return to a stable state after an attack or compromise. |
| **10** | * + DevSecOps Pipeline refers to integrating security into your software development life cycle.   + Phases and tools utilized:   + Plan – Jira Software for tracking and management and Slack for communication   + Code – Security tools such as Gerrit, Phabricator, SpotBugs, PMD, CheckStyle, and Find Security Bugs   + Build – OWASP Dependency-Check, SonarQube, SourceClear, Retire.jsm Checkmarx, and Snyk.   + Test – BDD Automated Security Tests, JBroFuzz, BooFuzz, OWASP ZAP, Arachi, IBM AppScan, GAUNTLT, and SecApp suite   + Release – Principle of least privilege & configuration management tools such as Ansible, Puppet, HashiCorp Terraform, Chef, and Docker.   + Deploy – Runtime verification tools such as Qsquery, Falco, and Tripwire, which can extract information from a running system in order to determine whether it performs as expected.   + Ongoing Security – Runtime Defense tools such as Imperva RASP, Alert Logic, and Halo |
| **11** | Risks   * Too many layers can add complexity * Need expertise * Can be costly both financially and with time   Benefits   * Preventative * Early Identification and elimination can be less costly than having to fix damage from an attack * Offers secure protection from attackers |
| **12** | * + - Adhere to our proactive security model     - Leverage industry groups for standards support     - Make use of the tools provided to catch and prevent any problems early     - Keep monitoring for any chances or new threats     - Follow the Defense in Depth module     - Make use of:       * The 10 Principles       * The 10 Standards       * Encryption         + In Rest         + At Flight         + In Use       * Triple-A Framework         + Authentication         + Authorization         + Accounting |
| **13** | * Coding Standards that should be adopted to prevent future problems   + 1. Validate Input Data   + 2. Heed Compiler Warnings   + 3. Architect and Design for Security Policies   + 4. Keep it Simple   + 5. Default Deny   + 6. Adhere to the Principle of Least Privilege   + 7. Sanitize Data Sent to Other Systems   + 8. Practice Defense in Depth   + 9. Use Effective Quality Assurance Techniques   + 10. Adopt a Secure Coding Standard |
| **14** | References |