# CS 405 Project Two Script Template

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hi, my name is Erin Walter, and I am here to present on the security policy and suggested updates to the security policy for Green Pace |
| **2** | First, we will go through an overview of what Defense in Depth is. Defense in Depth means that there is security at every level of the application, so that multiple layers of security work together to protect the system from any possible threats. There are outer features such as firewall protection, anti-virus and anti-malware software running, which is already utilized at Green Pace, but also automation tools to make sure any packages and dependencies used are kept up-to-date and bug and security vulnerability-free. There will also be security enacted at the code level through implementing secure coding practices. |
| **3** | The image of this Threats Matrix details an example of likelihood versus impact in terms of how urgent it is to fix a threat. If something is very likely to happen and will have a high impact, it should be immediately addressed and remediated. This does not mean that things that are low likelihood and low impact should be ignored, as it is still a threat or bug that could lead to a security hole that could cause a breach, but this gives a framework of how bugs and issues should be prioritized when looking at threats. |
| **4** | Looking further into the ten secure coding standards that were laid out, four are both high likelihood of happening, and high impact, so these should be addressed right away. These are String Correctness, SQL Injection, Memory Protection, and Arrays. Two are likely to happen, but medium impact, and these are Exceptions and Data Value, two are medium impact, but unlikely to happen, so they are lower priority. These are Random Numbers and Data Type. And then the last two are both unlikely to happen and have low impact, and these are File In/Out and Assertions.  When looking at reworking the code to remediate any security vulnerabilities, they should be prioritized according to this matrix with the highest priority ones being looked at first. |
| **5** | Next we have the Ten Core Security Principles listed. These ten principles will be use to guide how Green Pace from planning, to design, to architecture, to coding, and to maintenance of the system will make sure that there is security throughout the process. Each coding standard also abides by one or more of these security principles and they are also listed on this slide.  The First is Validate Input Data, meaning that any place in the system where a user is able to input data, we will work to validate it to make sure that it is the correct data type, that it is from a trusted user, an that it does not contain any extra phrases or data that it should not contain.  The second is heed compiler warnings, meaning that during the coding process, if there is a warning thrown by a line of code, we will read, react, and re-work the code if necessary to get rid of any warnings.  The Third is Architect and Design for Security Policies, which means that security is not an afterthought, but it is going to be architected and designed into the system as a whole.  The Fourth is Keep it Simple. Keeping the code simple and readable and maintainable will help keep it safer and more secure.  The Fifth is Default Deny, meaning that our default setting will be to deny access to our system, and only authorized users will be able to access areas of the system that their role dictates. This relates to the sixth which is Principle of Least Privilege, meaning that users will be assigned roles and only allowed to access areas of the system that they have been given access to and that is the minimum amount of access that they need for their job or role.  The seventh is Sanitize Data Sent to Other systems, which means that data will be sanitized, converted to correct format, encrypted as needed, and checked for any possible security attacks before being sent to another system or database.  The eighth is Practice Defense in Depth, which was described earlier, and means using a multi-layered approach to security.  The ninth is use effective assurance techniques, meaning we are going to make sure that the code is being tested and analyzed incrementally throughout the development process, instead of just at the end.  The tenth principle is Adopt a Secure Coding Standard, meaning that developers are using secure coding patterns throughout the development process, and not just adding security features at the end. |
| **6** | This slide details the ten secure coding standards that Green Pace is focusing on and implementing. When looking at these coding standards, they were also ranked as shown in the previous threats matrix slides according to how probable and likely they were to occur and what the impact of each coding standard would be, and they will be addressed in that order. With each of these coding standards, there is also listed in the security policy what a compliant versus non-compliant code block would look like, and which automation tools can be used to help with detection and remediation of each of these coding standards. |
| **7** | Next, I want to go over the three encryption policies: Encryption in flight, encryption at rest, and encryption in use. Each of these serve an important part in keep the data safe in all parts of the system.  Encryption in flight is when data is currently in transit and actively moving into or through the system. Transport Layer Security will be used, as well as security certificates to authenticate the sender of the data. All data will also be encrypted and not decrypted until it has reached its final destination or is being ready to be used.  Encryption at rest pertains to data that is being stored in a database. All sensitive information will be encrypted in the database, and will need a specific key to decrypt this. Also, there will be many layers of protection including authenticating and authorizing users to access the data using login and access roles, firewall security, anti-virus and anti-malware software, and the principle of default deny to keep this data safe.  The last type is encryption in use. Data in use is being read, accessed, updated, modified, or deleted. This will also follow the principle of least privilege and default deny, so only users who are authorized to use this data will be allowed to have access. Also, the data will be encrypted until absolutely necessary to be used, then will be decrypted only while serving its function, and then encrypted again. We will also be using tools to make sure that all packages and dependencies are up to date and do not contain any bugs or security vulnerabilities, as well as utilizing a other security measures at a systems level to keep the data safe while being used. |
| **8** | Next, the triple A policies of authentication, authorization, and authoring will be detailed. These policies are critical to making sure that only users who we have given access to our system are able to have access, and that they are not only tracked and logged while using our system, but only given access to what they absolutely need to be able to function within the system, and no additional permissions.  Authentication is the process of identifying a user. This is through the user logging into the system with a username and password or requiring an Auth0 token to prove their identity.  For authorization, we will be granting access levels via roles for each user, so that their access to the system is limited to only what they absolutely need to have access to, and nothing more. This also allows for their access to be taken away quickly if they are acting suspiciously in the system.  For authoring, this is the process of logging and keeping track of a user’s footprint while they are accessing the system, as well as the amount of data they are consuming. This way, any suspicious activity can be flagged and monitored and further looked into to make sure that users are not attempting any sort of attack. |
| **9** | The next few slides give examples of unit testing and how to set up unit tests to make sure that the code is functioning as expected, and throwing the appropriate exceptions if an error occurs.  The unit tests are through the Google Test framework, and to set these up a new Google Test C++ Project must be built. When writing the unit tests, the ASSERT and EXPECT keywords are used in order to make claims on how you are expecting the code to behave. Then, you can run the unit tests using the debugger and the console window is able to tell you the results of the tests. |
| **10** | This first test is a positive test which is checking to see if resizing the collection will increase its size. First, a new collection with size 0 is created, and an assertion is made to show that the collection’s size at this point is equal to 0. Then it is resized, and an assertion is made to assert that its new size is 5.  The result of the test is shown in the bottom right, showing that it was run, and that it passed with “OK.” This is what would display in the console window |
| **11** | The second test is an example of a negative test, to test to make sure an exception is thrown in the case where an out of range location is called to in a collection. To test this, 5 entries are added to the collection, and then an EXPECT statement is called to test whether the expected exception was thrown in this case. |
| **12** | This third positive test is checking to see if calling clear() will erase a collection. A number of assertions are made to make sure that at each step. First, a collection of size 5 is created, and then an assertion is made to prove that its size is 5. Next, the clear() function is used on the collection, and we assert that it is now empty() and that its size is now 0. |
| **13** | This fourth unit test is a negative test, and is expecting an exception to be thrown when a collection is first resized, and then calling to a now out of range location that didn’t used to be out of range when the collection was larger in size. So, assertions are made to assert that it is first equal to a size of 5, that it was correctly resized to a size of 3, and then an EXPECT is called to make sure that when calling to a location of at(4), an out of range exception is thrown |
| **14** | An important concept that is related to unit testing is code coverage. When writing out unit tests, it is important to make sure that each section of the code is tested using a unit test. This means for if/else statements, switch/case statements, and each branch of a try/catch statement, these should all be tested out in individual unit tests, to make sure that each area of the code behaves correctly.  If you only test the “if” section of an if/else statement, and while the code is running the “else” statement doesn’t behave as expected ,this could introduce a bug or crash the code at runtime. Using a unit test to test out this “else” branch of the statement could have prevented this from happening |
| **15** | This slide shows the DevSecOps automation pipeline as it currently stands. It is broken down between pre-production and production pathways, and each step gives details on what is currently in place at Green Pace with the current security strategy.  In the pre-production route, currently, the flow is as follows: Assess and plan, where threats are assessed, prioritized, and plans to respond and remediate are made. Then, the code is designed by looking at current standards using OWASP best practices to remediate. After this, the code is built and these threats are remediated. Once remediated, the changes are verified and tested.  In the production route, there are health checks done at deploy time as well as penetration tests, and once deployed there is monitoring which includes log collection, and detection which includes alerting tools to show if there is an error in the code. The next portion is responding to any threats such as using default deny to block access, turn off certain services or problem areas of code, or roll back if the deployment was not successful and is breaking or causing issues. The final portion is to maintain and stabilize the system. While this strategy is a good start, some of it feels reactionary on the production’s part, instead of proactive with trying to prevent an attack from happening in the first place. |
| **16** | Some automation tools that could be implemented in this pipeline to help in the pre-production process are google test platform for unit testing and Cppcheck. Google Test can be used as shown before with unit testing and making sure there is good code coverage for unit tests. Cppcheck is a tool that will run a static coding analysis at compile time and gather compiler warnings and offer suggestions of how these can be fixed.  Other tools that can be implemented include various automation tools that are detailed in the security policy for each of the ten coding standards. A couple examples include Axivion Bauhaus Suite, Clang, SonarQube or CodeSonar. These can be added to monitor for any issues in the code that should be remediated before deploy time.  There are also other tools that can be used in production to make sure that all dependencies and libraries stay up to date. These will keep track of any new bugs or vulnerabilities found in any dependencies and suggest remediation steps like updating to a new version of the package. |
| **17** | Some risks to immediately fixing all vulnerabilities are that we may not have the full picture of the security vulnerability. Also sometimes the code re-work may cause other vulnerabilities if it is not properly analyzed. Another risk is that acting immediately could cause the issue to only partially be fixed, and some risks being accidentally omitted in the fix.  Some benefits to fixing all the vulnerabilities immediately are that for the high priority and high risk errors, these could be immediately remediated and this could be of great benefit to our overall defense level and risk at a systems level. This means if any attacks were to be attempted, they would have a lower likelihood of being able to get into the system.  Steps to think about when making a strategy to minimize the risk level of the system as a whole is to prioritize the bugs and vulnerabilities. The highest priority ones that are the most severe should be fixed first, to make sure that there are multiple layers of defense covering the system in this area.  For medium and lower priority risks, a more complete plan can be developed and these can be prioritized accordingly to make sure that the re-work and updates that need to happen are complete and not missing any parts and are done in the most efficient way possible so as to not waste time or money while trying to remediate them. |
| **18** | I do have some recommendations in terms of gaps I currently see in the security protocol as it stands, and opportunities to improve it. First, for any of the ten coding standards, I would make sure that these are looked at during the analysis and design of the architecture so that it can be correctly coded the first time through, instead of later re-worked after a bug is found. Second, I would make sure that unit testing and code coverage is made more of a priority and better integrated into the pre-production process. Third, more analysis tools can be added pre-production to make sure that a scan is done for any additional warnings that should be addressed, or any possible vulnerabilities or bugs that should be fixed before the code goes to production, rather than reactively finding these bugs in production. |
| **19** | In conclusion, I would make sure to look at implementing security in the pre-production process at all levels. When security is added in by design and planned in coding rather than done as an afterthought, it is much easier, more efficient, and less costly to code securely and eliminate risks the first time around, rather than catch these errors in production and have to re-work the code. I also would add more automation tools and unit testing to the code to make sure that while the code is being developed, it is being scanned, tested, and analyzed so that as many bugs and vulnerabilities as possible are caught immediately and able to be remediated before making it to a production environment. Third, when designing and making the architecture for the system, the ten principles should be applied in this process and designed into the system right away, not looked at afterwards like a check list. Security should not be applied at the end or thought of as an afterthought, but added proactively at all stages. |
| **20** | Here are some resources and additional information if you would like to look up any coding tools or documentation on security principles. |