# CS 405 Project Two Script Template

Colton Berger | 4/21/2024

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Welcome to the Green Pace security policy initiative. My name is Colton Berger and I will be going over the various policies and standards we’ll be implementing here and why having a security policy is so important. |
| **2** | Creating a defense in depth approach to security will make compromising the systems we build much more difficult. Ideally, this will make the system so difficult to compromise that attackers will give up because the time outweighs the reward. |
| **3** | Different threats will have different impacts if they happen, but they also have different likelihoods of happening. Combining their likelihood and impacts together gives us this threat matrix. Events that have a large impact and are more likely to happen are located in the top right as a priority. An example of this is a buffer overflow being found. Something that has a low impact and will rarely happen is comparing floats in a function with no return. This has little impact of the results of the program but will still need to be evaluated and addressed. A SQL injection attack has a large impact but isn’t as likely to occur, this puts it in the unlikely category. A user password getting compromised is more likely to occur but has a lower impact since MFA is in place and we have limitations for how users can access our systems. Each new threat will be placed in a threat matrix so that we know its significance. |
| **4** | These are the ten core policies that we will be implementing and adhering to. Each principle that is on this list can be applied in some way to the coding standards that we will be reviewing next. These range all the way from being very specific, like validate input data, to the broader end of the spectrum such as practice defense in depth. |
| **5** | Here we have all ten of the coding standards, ranked in the order of importance. The ranking system was determined by a combination of severity, likelihood, and remediation cost. All of these factors were used to decide that SQL injections and memory protections rank as the most important while assertions, variable comparisons, and integer overflows rank at the bottom of the list. This doesn’t mean that these standards can be disregarded but rather that when issues arise, the ones that rank higher will have priority for remediation. |
| **6** | Moving on from the policies and standards, we have some encryption policies that will be implemented. The first policy is regarding encryption at rest. Encrypting data that is being stored in a database will make it so that it is unreadable in the event that someone is able to gain unauthorized access to it. Encryption in flight is the next encryption policy. Encrypting data in flight is essential so that if it is intercepted, the data is unusable. This will be done using end-to-end encryption so only the sender and receiver can accurately decrypt the data. The final encryption policy that is going to be implemented is encryption in use. When data is actively being used by someone it is in its most vulnerable state. This is why protecting data that is in use is so important. This will be done by securing the memory that the data is located on and also through the use of hashing. With these policies, our data will be secure throughout its entire lifetime. |
| **7** | Now we will be moving on to the Triple-A framework. Triple-A stands for authentication, authorization, and accounting. Authentication is the process of a user verifying their identity through a login before being allowed access to private data. Having an authentication process in place will help prevent people from accessing our data that shouldn’t be. Authorization will be used to determine what data a user should have access to once they complete the authentication step. This will provide the level of access for each user. Accounting is the final portion of the Triple-A framework and is applied with logs and record keeping. With accounting, we will know exactly what files a user accessed and what changes are made to a database. These logs will tell us who made the change, what the change was that was made, and when they made this change. Using all of these in conjunction with each other will maximize the benefits that they will provide us with. |
| **8** | With this new security policy, we will be introducing unit tests. Over the next few slides, we’ll look at what the unit test is testing, the results, and why each test is necessary. All of these tests are for testing vectors in the form of a collection. This first test is to prove that when a new collection is created, it is empty. We test this by creating a new collection and then asserting that the size is equal to zero. Since the results show that this passed, we know every time we create a new collection, it will be empty. |
| **9** | This test is verifying that we are able to add a single entry to the collection. To do this, we first verify that a new collection created and is empty and then we add one entry. After adding a single entry, we test that the size of the collection is equal to one. We can see by the test results that adding a single entry increases the size from zero to one. |
| **10** | Here we are testing to verify that the max size of the collection is greater than the size of the collection after adding entries to it. In this test, we create a collection, add an entry to it, and then check that the max size is greater than the current size with the entry. We do this for different collection sizes, verifying that the max size is greater than or equal to the collection size for each one. From the results we know that this is true, and the max size is always greater than or equal to the current size, regardless of the current size. |
| **11** | Now we are testing if resizing will change the size of the collection. Here we create a new collection with a specific size, use the resize function on it, and then test that the size is equal to the new size that we declared. This was done both by increasing and decreasing the size via resize. Looking at the test results, we can see that this is true, even if there are entries saved to the collection prior to decreasing the size. |
| **12** | Here we are testing to confirm that an out-of-range exception will be thrown if we attempt to access an index that is outside the size of the collection. By setting the size of the collection, we are then able to access an index that is outside that range while expecting an exception to be thrown. Since the result passed, we know that this did throw an exception, like we expected. |
| **13** | Our final unit test looks at using the reserve function to increase the capacity of the collection but not the size. For this test, we used the reserve function to change the capacity of a collection, and then verified that the size was the same as it was prior to using the reserve function but that the capacity has updated to the value used in the reserve function. Based on the test results, we can tell that the size does not change when using the reserve function while the capacity does. |
| **14** | Within the DevOps cycle, automation will be included in multiple places. Automation will be incorporated on the pre-production side into the design and verification and testing phases. On the production side, automation will be incorporated into the monitoring and detection and response phases. |
| **15** | Introducing automation to the DevOps cycle will improve our security while also providing a way to enforce these new policies throughout development. This is the start of turning our DevOps cycle into a DevSecOps cycle. The tools we will be adding in pre-production are Sonarlint, Sonarcloud, and Selenium. Each of these tools will play a role in checking code for security vulnerabilities and also running unit tests for bugs. On the production side we will be using automated logging to identify unusual activity and if unusual activity is seen, we will be using automation to block those connections until we can verify them. |
| **16** | When implementing a new policy, there will be risks and benefits. If we wait to implement this policy, the chance of a code vulnerability being exploited will continue to increase and all new code committed to the code base will go unchecked against these standards. If we act now, we will be able to correct these vulnerabilities from highest risk to lowest. This will take time and could possibly slow down development until all of the standards have been implemented into the current code base. |
| **17** | My immediate recommendations are to implement the 10 coding principles to begin the process of adapting to the DevSecOps pipeline. My next recommendation is to implement the 10 coding standards from highest risk down to lowest since it will interupt the current development pipeline. I would also recommend that we analyze our systems and processes so that we can begin implementing automation into our pipeline. My final recommendation is to review this policy yearly for any changes that need to be made. |
| **18** | In conclusion, all standards should be implemented to improve our security here. Since this will need to be a process and not immediate, the two policies that are most important are regarding SQL injection and memory protection. Patching any code that is vulnerable to SQL injection will prevent unauthorized access to the databases. Correcting code that doesn’t have proper memory protections will prevent the access of hidden data and also prevents unauthorized code from being ran. Finally, automation can be used to speed up development and we should begin the process of integrating it into our DevOps pipeline. |