# CS 405 Project Two Script

Hatcher Blair

Project Two: Security Policy Presentation

12/10/2023

Video Link: <https://youtu.be/2zzeLGPo3_Q>

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | My name is Hatcher Blair and today we are going to discuss the new Security Policy being implemented here at Green Pace. |
| **2** | The goal of this policy is to ensure that all potential data breaches will be prevented in the future. Obviously, you can never be 100% secure from breaches. However, with this policy we will have multiple layers of defense at every point of attack. This policy outlines ways to prevent vulnerabilities and best practices when moving the code from development to production. |
| **3** | The impact matrix covers the biggest potential threats to our organization, how likely they are and the potential impact. The impact is measured based off the cost to remedy the code base to prevent this threat. We will go into each of these threats in greater detail on a later slide. |
| **4** | We have 10 coding principles that must be followed at all times. Let’s work through them one by one. Some of these may seem like common sense but the ones that are the easiest to comply with are often the most forgotten about. These principles link to standards that we are going to hold our code base to from now on.  Validate Input Data: Validating input data most closely links to the standard of preventing SQL injection. Preventing injection is just one form of validating input data. We are going to talk about this more later and so I will leave it at this for now.  Heed compiler warnings: This is more general than one specific standard however, this applies to two standards that we are implementing today. Do not create a string from a null pointer and ensure that integer conversion does not result in lost or misinterpreted date.  Each of these on their own may not cause a compiler warning but they could. When prototyping it is ok to have some compiler warnings and in deployment it is also ok to have some compiler warnings but, it is important to assess each warning that the compiler throws. Warnings can give key insight into potential issues with the code. It will still compile and execute, meaning that any vulnerability caused by a warning will be deployed if not addressed.  Architect and design for security policies: This applies to everything we do. My team has spent a lot of time designing this security policy and now we are going to need to spend a lot of time refactoring around this policy. In the future we are going to need to design new features and implementations around this policy.  Keep it Simple: Again, this does not apply to one specific standard that we are going to talk about today. This applies to everything we do, usually the simplest answer is the best answer. This is usually true for several reasons. One, the simplest answer is the easiest to read, meaning that in the future other team members can easily understand your code. Two, it is most likely going to run the fastest out of all the solutions. Three, less room for vulnerability. If your solution is 10 lines long, there are only 10 lines of code that could have a vulnerability in them. If the solution is 100 lines long, there are 100 lines that could have a vulnerability in them.  Default Deny: Currently, we don’t have a standard for this principle. From now on, we are going to have a policy of default deny on everything. Meaning that every user by default is denied access to everything. Permissions are set to allow access, not deny access.  Adhere to the Principle of Least Privilege: This goes together with default deny. Each user is going to have the least amount of privilege in the system as possible. I expect this to require a lot of refactoring, but it will all be worth it. Example: Currently the marketing team has access to all our tax filings, does the marketing team really need that information? No, instead the marketing team will only have access to exactly the parts of the system they need to access and nothing else. Why do they have access to the tax fillings? Because one of them wanted to make an ad talking about how successful we are. Now they are going to have to message someone on the business team or get up from their desk and go ask them in person.  Sanitize Data Sent to Other Systems: This can also be applied to SQL injection like validating input data. However, this is for data going out not data going in. When sending data to a user it needs to be cleaned and only contain the information they requested. This applies to data being sent to other systems internally as well as other systems externally. Not only can this help prevent attack methods, but it also will reduce packet size and network traffic.  Practice Defense in Depth: This is what we are implementing today. Practicing defense in depth is an organization wide effort. To ensure this, we are implementing the policy and going to be auditing the existing code base to ensure that multiple checks are in place at every part of the system.  Use Effective Quality Assurance Techniques: This is all our standards. Applying the standards that I am going to introduce on the next slides will help with quality assurance but is not all of it. We also need to be testing our own code and auditing each other’s code to ensure that it holds up to company standards.  Adopt a Secure Coding Standard: The secure coding standard that we are adopting is what I am introducing today. However, the standard is not in its’ final form. I need the help from everyone in this room to keep the standard up to date and relevant to the work that we do. This ensures that we are protected against today’s threats and the threats that we are going to be facing in the future. |
| **5** | These are the 10 coding standards that are being introduced today. Using these standards is a good start to ensuring that your code is secure, but it is not everything. Over time new standards will be added to the list to make it more complete. If you want more information about any of these standards, refer to the policy or google the name of the standard. These are in order of the remediation cost; it can be treated as a priority for which of these standards to focus on implementing first. Now let’s go over them!  INT31-C: This applies to the C language but can be generally applied to all languages that we use. When converting from integers we all know that we need to check for overflow however, in some instances this check has been forgotten about. This is often forgotten about when working with a variable type that is not an integer but is converted to an integer by the compiler. We must always be checking for overflow and underflow!  STR51-CPP: This applies to the C++ language and is not verry applicable to the other languages that we use. You might be wondering who is going to create a string from a null pointer? Well, if we have a bad function call, a null pointer could have been passed in by accident. We must check for null pointers before all string creation.  STR32-C: Another C language one. Again, this one could happen by accident most of the time. It is easy to assume that our character array is null terminated but what happens if we don’t leave enough room in our array? It is also possible that the character array was just not null-terminated to begin with. When calling a function make sure that the input data is properly formatted and if the character array is not terminated, terminate it.  IDS00-J: This is for Java, but we take data in using several different languages. If you are front-end, back-end, or full-stack this is very important to you. We will talk more about this later so I will leave it at that.  EXP33-C: Yet another C one. This really only applies to C as all the other languages we use manage memory automagically for us. All of you that write C make sure that the memory you are reading to has been initialized properly. Obviously, you guys know the damage of reading from uninitialized memory, but it needs to be second nature to allocate and initialize the memory you need the second you start working on something.  MSC03-J: This is tagged as a Java one, but it applies to everyone. I don’t think we have any data that is hard coded in our system, but this can NEVER happen. Anytime the program is working with sensitive data it needs to be requested from the system that is storing it and returned or deleted once we are done with it. We are going to talk more about encryption later and this one might come up again. 😉  FIO51-CPP: This is another one that is generally applicable to all languages we use and might come up again later. Files need to be closed once we are done with them, not 5 minutes later or at the end of the file because we forgot to do it earlier. The second we are done with them, close them and return or delete them.  ERR50-CPP: This applies to everything we do here. The program should never be terminated abruptly. Obviously, we never intentionally end the program abruptly, but we need to make sure that when there is a crash, we can safely exit the program. This is because we need to make sure all memory is returned to the system, all data that is being used is safely saved and exited, and that there is no chance for an attacker to gain access during a crash.  MSC60-J: Pretty self-explanatory, instead of verifying that there are no errors, address the error where it could happen. Example: When opening a file, don’t just assert that it was opened, check to make sure it was opened and handle the case where it was not.  ERR51-J: Last one guys, hold in there. If there is not a descriptive exception for what you are catching, make your own. It will make your code easier to read and when that exception is caught it will help diagnose the issue. |
| **6** | Alright, now let’s talk about encryption! Remember when I said we were going to talk about this later on the last slide? Well, that time is now. There are three types of data that we are going to be focusing on. Data in flight, at rest, and in use.  Data in flight is being sent from one system to another, data at rest is in long term storage, and data in use is currently being used to perform calculations.  For data in flight, we already have some basic security measures in place, but we are going to be improving our checks to ensure that it is working as intended.  For data at rest, we have good systems in place already, but again we are going to be expanding our checks to ensure that it is working as intended. This will also need to be worked on with our new policies for default deny and least privilege.  For data in use, we are going to have a lot of work to do. Currently we decrypt all data that is being used. We are switching to a policy that decrypts as little data as possible. But how can you work on data that is encrypted? Great question! We are going to be implementing homomorphic encryption where possible, this will allow us to perform calculations on the cipher text directly instead of having to decrypt the data first. This is not possible everywhere and, in some instances, we will have to decrypt data, but it is possible for most instances and will be implemented in those places. |
| **7** | Now let’s talk about triple-A, authentication, authorization, and accounting.  Authorization policies are already in place for all users. However, we are going to being requiring MFA for more groups of users in the future. The business team and you guys, the dev team, are already used to this but all internal users are going to be required to use MFA from now on. External users will still not be required to use MFA but we are going to add some additional nudges towards users enabling it.  Authorization: We are going to have our work cut out for us on this one. Switching to our default deny and minimal access policies means we are going to have to do a lot more authorization. That is ok, we will continue to work on it over time.  Finally, we have accounting, and not the kind of accounting that the business team is doing. This is keeping logs of users’ reads, writes, logins, and more. We are going to create separate logs for location, user groups, and the type of log. This will make auditing our logs easier in the future. The logs will contain the last 30 days of activity and after 30 days most of the logs will be deleted. However, certain types of messages in the logs will be kept in long term storage for 3 years. The team that is going to be working on developing the logs will get more information about this later. |
| **8** | We are also going to begin to do more unit testing throughout development. In the next couple slides I outline an example Unit testing procedure for one of the coding standards being implemented. This example is for the IDS00-J standard which is prevent SQL injection.  To verify that I am properly preventing injection attacks when submitting a form, I use a series of five tests. The first test is a positive test, meaning that the test will be completed without any errors. The next four are negative tests, meaning that an error is expected and required for the test to pass. |
| **9** | The first test is to verify that the form functions properly. To complete this test, the information requested will be filled out in proper format and submitted. |
| **10** | The next test verifies that the form checks for properly formatted input. This means that an email needs to have an @ and a domain name, the phone number is 10 digits and only digits, and the password meets security requirements. If this test was being implemented, this would be broken into separate tests for each section of the form but for example purposes these tasks are all grouped into one test. |
| **11** | This test verifies that the back end is checking the inputs before the database is being queried. This would also be broken up into three separate tests for implementation. The three tests would be checking the input format again, ensuring that prepared statements are being used, and the username and password are valid. For new account creation this would mean the username is unique and the password meets security requirements. For a returning user this would mean the database was queried, the user existed, and the password matched the one that was stored for that user. |
| **12** | The next two tests involve attempting an injection. The first one checks that most injections will not make it through the front end. This test will involve submitting an SQL statement into the username, password, and email lines. The username and email should block most injection attacks. This is because the username does not allow spaces or some of the characters commonly used in injection attacks. The password allows all characters and in most cases the injection will make it through in the password slot. |
| **13** | This test verifies that injection is blocked by the back end. For this test to begin, SQL statements must be passed to the back end from the front end. First, we begin by testing the back end allows a valid user to enter the website. Then we will test with a dangerous statement in the password, and then test again with a dangerous statement in the username. Finding a statement that can be injected in the username may be difficult, but it is possible. |
| **14** | We will be using automated testing and automated security at most points in this diagram. The sections that don’t require any automation are the asses and plan and design stages. These are the only two stages of the DevSecOps diagram that do not involve coding.  During the building phase we will use a combination of automated testing and manual testing to verify that the design is functional and meets security standards.  During the Verify and Test phase we will do more rigorous testing including testing the entire design functions together and testing the multiple security layers implemented.  During the transition and health check phase we will be using automated security tests to ensure that all the security measures in place made it into production.  During the monitor and detect phase we will be using the logging solutions that we designed earlier as well as some other third-party automated tools to ensure that the security defenses are functioning as intended.  The response phase will have some automation and some manual responses. The automated responses will be for lower security risks such as several login requests, maybe they forgot their password, unusual logins, and similar items. Any activity that is determined by the automated systems to be malicious will result in immediate account deactivation. From there a member of the security team can assess the account and determine how to move forward.  During the maintain and stabilize phase there will be little automation. Most of the maintain and stabilize phase is reviewing what has happened to the system recently and identifying problems. Once problems are identified, the problems will be passed back to us and we will start the cycle again at the asses and plan phase. |
| **15** | Here is a starting list of external tools that we will be implementing into our workflow. Not every single one of these tools will be applicable to your daily work, some are language specific. The goal of these tools is to provide automated security checks throughout development and into deployment.  The tools that we use will be ever expanding as new tools come out and there are new vulnerabilities discovered. Make sure that you are staying on top of new vulnerabilities and new versions or tools that come out. This will help all of us implement the best security policy we can. |
| **16** | It is important to understand there will always be holes in your defense strategy. However, this does not mean that we should sit around and wait for an attack to happen. In most cases if an attack has occurred it is too late to protect ourselves and our users from it. That is why we are implementing this policy. |
| **17** | There are two gaps that are currently in this policy, and I’m sure there are more but these are two categories that are not being addressed in this policy. Those are offensive security, or offensive defense, and physical security.  Offensive security is simulating attacks to test the system and potentially hiring third parties to test our systems. This will be implemented over time and some of the unit testing we do is a form of offensive security. Eventually we will have a team, or a third party, that is dedicated to attempting to gain unauthorized access to the system to ensure that our defenses are working and cover the areas that are most important.  Physical security is something that we may not have that much control over depending on the implementation. For our serverless solutions, we do not have much control over physical security. We can request that our database providers increase security or modify security in some way, but we don’t have actual control over it.  For our solutions that are self-hosted, we can control physical security and that is a portion of the policy that will be improved on soon. This includes protecting access to the servers, the facility, the power systems, and the cooling systems. Once we have clear policies in place to address these issues, we will have a third-party attempt to physically break into our sites to ensure that the defense is working as intended. |