# CS 405 Project Two Script

Vincent Snow

2/24/2024

<https://youtu.be/XJ-IOgguD0s>

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | [not narrated] |
| **2** | This security policy is one part of Defense in Depth – a multilayered practice of securing an application and the sensitive data that we store through physical as well as digital means at every access point. These deep layers of defense start from protecting the building and hardware where data is stored with locks, alarms, and CCTV monitors. Then, each layer of the system or software needs protection of its own – the host, the networks, each API endpoint, and anywhere else in the application when data is in storage or in transit. This policy is part of our application security – it gives us a proven standard to follow to prevent many vulnerabilities in the software itself. |
| **3** | This matrix summarizes the risks addressed in our security policy by how likely they are to occur, and how serious they are if they occur.  The most attention should be given to those that fall in the red area. The problems we address here are String-related Buffer Overflow, Uninitialized Memory Reads, Creating Strings from Null Pointers, Injections in Traveling Data and Input, and Integer-Related Under & Overflow. These are all problems that can disrupt a program or cause unexpected behavior that is difficult to trace. Our standards seek to prevent each one of these before they occur and test thoroughly for them.  Other problems it addresses include Memory Exhaustion, Unexpected Float Behavior, and Overwriting Reserved Identifiers in the ‘likely but low priority’ section, Unhandled Exceptions in the ‘unlikely but high priority’ section, and Unexpected Behavior from Invalid Data in the unlikely and low-priority section. |
| **4** | These are our ten Security Principles. All our standards support these principles.  The first is to Validate Data Input, or rather do not trust any input. All input is vulnerable to attacks such as code injection either by accident or maliciously, and this can either disrupt a service or expose sensitive data.  The second is to Heed Compiler Warnings. Warnings can create problems later and point out hidden mistakes in a code, so they should never be ignored.  Three is ‘Design for Security Policies’ which we will be doing during the design phase by following these standards and policies.  Four is ‘Keep it Simple’ – overly complex code is more error prone, and the simplest solution should be chosen.  Five, Default Deny, means to deny access to data by default. Do not allow access until identity has been proven.  Six goes hand in hand with this – the principle of Least Privilege. Do not grant access to data when it isn’t necessary. Give the privileges that are necessary to do one’s job and no more than that.  Seven, sanitizing data sent to other systems helps prevent code injections or leaks of data. This is another way of not trusting input.  Eight, defense in depth, as I talked about in a previous slide, means never to rely on one layer of protection alone. Always have multiple blockages to an attack through any route.  Nine, effective QA techniques include efficient responses to reports and bugs. Security standards during design and development can help make this process easier.  Finally, ten – Adopt a Secure Coding Standard. The standards we have adopted help prevent problems before they start by addressing common code vulnerabilities that can be exploited. |
| **5** | These are our Standards, adapted from the SEI-Cert C-language secure coding standards, ranked in order of highest priority to lowest based on their likelihood of occurring, severity and remediation cost of the problems that may arise if they are not followed.  Firstly, we must guarantee that storage for strings has sufficient space for character data and the null terminator. In other words, avoid overflow into strings by limiting input and readingone character at a time. This is how code injections or accidental DoS can occur.  Second, we must do the same for Integer data types, avoiding overflow and wrapping by ensuring that there is enough space in the data type to perform calculations and store the amount.  Third, sanitize data passed to other systems. Make sure it follows a certain format and check it for suspicious data during output, before it is even caught and checked again as input.  Fourth, we must never attempt to create a string from a null pointer, which will have unexpected behavior.  Fifth, detect errors without dictating the handling. Simply log and throw the error in a try-catch block and continue the program without executing the function.  Six, do not assume infinite heap space. Cleaning up data after creation is very important. We do not want to leave scraps in the heap.  Seven, we must never use floats as counters. Floats are not exact integers and may cause unexpected behavior.  Eight, do not read uninitialized memory. It is good practice to initialize all variables into something, even an empty string or zero, upon creation.  Nine, use Assertions for Diagnostic Testing. Every variable and function should have an assertion to ensure it is working properly during development and testing.  Ten, do not use reserved identifiers. We must avoid attempting to overwrite keywords from the language and libraries, and heed compiler warnings on this. |
| **6** | Our encryption policy tells us when and why to apply encryption on data – which should be done according to current legal standards for sensitive data. Passwords, credit card numbers, full addresses, names, and phone numbers are examples of data that should be encrypted during transit and storage. Both states are vulnerable to exposure because of accidents or deliberate attacks. While in use, decryption with the key should only occur when necessary, such as when verifying a password or sending a forgotten password to a user who has proven their identity. |
| **7** | These are the policies we shall follow for Triple-A security.  Authentication should be done by the use of unique usernames and passwords consisting of at least 8 characters and containing only alphanumeric characters and these special characters shown. Identity should be checked at each API request or function that transfers data or grants access to different areas of the application.  Authorization should be given based on roles such as user or administrator, which should be granted access only to the specific data and parts of the application that are of interest to them, and necessary for their use cases.  Accounting shall be done in the form of activity logs, which keep track of all users’ actions with credentials and ISPs. This should be made available only to the highest privileged users such as trusted officers or the company owner. |
| **8** | On the next few slides I will go over some positive and negative unit test examples in C++. This example of a unit test was used on a Vector collection to check that pop\_back() is successful. Simple tests of basic functionality like this ensure that things in our code are behaving the way we assume they are and help narrow down or trace bugs as they occur while developing. This one starts with a few basic assertions first for each step of the way – it first checks that the collection has been created and is not null, and then that exactly five items were added to the Vector before testing pop back. |
| **9** | This test on the same Vector object tests if an ‘Out of Range Exception’ is thrown if the collection is empty. This ensures that exceptions are being caught and handled properly by the built-in exception, not handled otherwise, in keeping with our fifth coding standard. |
| **10** | This test ensures that a proper exception is thrown when accessing an out-of-range index on the vector, and it does not crash the program or throw the wrong exception. |
| **11** | This simple test would be used before others to assert that the created collection is not a null pointer. Our fourth standard is to be sure that we don’t try to access or create anything from null pointers. These unit tests in C++ are created using the Google Test or Gtest framework. This framework is included in Visual Studio as an optional add on during installation and can be used by creating the project as a ‘Google Test’ new project. |
| **12** | The DevSecOps process is a continuous development process that keeps security in mind every step of the way rather than waiting to test and fix problems in one step at the end of development or after production. Beginning with Planning, and then Design, we keep in mind security best practices and standards and design for that. We use secure coding practices and tools during the build and test phase, and then continue to monitor the application after production. Security tools that we use help to respond to attacks and errors and then maintain the application, sometimes sending us back to a planning cycle to design, build, and test and upgrade or new features. |
| **13** | Automated security fits into the DevSecOps process during both pre- and post-production stages. We use tools such as assertions in our code during the design, build, and test process. Unit tests and error handling with logging are useful not only in this Verify and Test process, but in the Monitor and Detect up to the Maintenance phase, because they continuously check for errors and log activity so that we can better find the root cause of errors and vulnerabilities. External code-reviewing tools can speed up the process of development and are used along the way from the build phase to the testing phase. |
| **14** | ‘Acting now’ versus ‘doing it later’ is far more beneficial and minimizes risks.  Following a DevSecOps process may be more work at first, require more training, and more effort during the development process. But this method will save time, effort, and even money and resources later on.  Saving security concerns for later means much more time spent trying to fix problems that could easily have been avoided with as little as a one-time addition of automated testing. When problems happen this way, you may end up having to untangle a chain of errors or find the problems very difficult to track down, because you have been building on top of error-prone, insecure code. One uncaught problem may lead to having to redesign and build an entire application or large part of it.  Acting now also delivers better quality assurance to your customers – you will be able to create safer products with less disruptions and errors, as well as respond much more effectively when attacks or problems do happen. |
| **15** | There are a few gaps in our security policy that should be addressed within our coding standards. Although string and integer safety have been addressed, we need to follow more specific standards to prevent memory over-use that makes the application more vulnerable to disruption of service. Input should never be trusted – therefore we need specific limitations on string input and standards for when and where to use which numeric data types. We also need to address the safe handling of files. |
| **16** | The following suggestions for standards should be adopted into the policy to address these gaps. STD-011 - Strings for usernames and passwords should be limited to 64 characters, while free text fields should be limited to 300 characters. STD-012 - Integers should be used as counters and iterators, floats for measurements, and shorts for numbers not expected to be very large. Currency shall not use floats and should be counted with integers. STD-013 - Files should always be checked if valid before reading, and always closed after use. |
| **17** | [Sources, not narrated] |