**CS405 Module Project Two Script**

Dante Trisciuzzi

10.12.2023

CS-405 SNHU

Prof. Ahlam Alhweiti, M.S

**Slide 1 - Title**

{Introduce the presentation}

**Slide 2 - Overview**

Our security policy adds multiple layers of protection, from early detection to controlled access. We’re adding various layers to embody the defense-in-depth principle.

**Slide 3 - Threats matrix**

We’ve defined a set of coding standards, this matrix briefly shows the distribution of the standards and maps their threat level.

1. Proper Use of Data Types, STD-001-CPP: Misusing data types is likely but has low severity and cost to fix, making it a medium priority.
2. Initialization of Data Values , STD-002-CPP: Uninitialized data values are likely and can lead to issues; its medium remediation cost makes it a high priority.
3. Safe String Handling , STD-003-CPP: String vulnerabilities are likely and have high severity, placing this at high priority.
4. Prevent Injection , STD-004-CPP: SQL injections are very likely and critical due to their high severity and remediation cost.
5. Safe Dynamic Memory Management , STD-005-CPP: Memory issues are likely with high severity but low cost to fix, keeping it high priority.
6. Over-use of assertions , STD-006-CPP: Misusing assertions is likely but with moderate severity, ranking it as medium priority.
7. Safe Exception Handling , STD-007-CPP: Poor exception handling is likely and can lead to significant issues, making it a high priority.
8. Safe Inheritance Practices , STD-008-CPP: Inheritance issues are possible but not guaranteed, leading to medium priority.
9. Guard Against Integer Overflow , STD-009-CPP: Integer overflows are very likely and of high severity, thus high priority.
10. Precision in Floating-Point Comparisons , STD-010-CPP: Precision errors are likely, but with moderate severity and low remediation cost, it's medium priority.

We can use automatic tools, like CPPCheck, for static analysis. These tools grade the codebase and the potential vulnerabilities therein. This lets us determine where to focus our energy while refactoring.

**Slide 4 - 10 Principals**

Here are the ten principles we’ve established {read principals side of table}. They are accompanied by the coding standards that demonstrate them.

**Slide 5 - Coding standards**

Here are the coding standards, along with their ranking from a system that highlights vulnerabilities based on their potential harm, occurrence probability, and the amount of resources needed for resolution. {read standards}

**Slide 6 - Encryption Policies**  
  
Encryption is our primary defense against unauthorized data access.

For Encryption in Rest, all sensitive data is encrypted. This ensures even if storage devices are compromised, the data remains inaccessible.

For Encryption at Flight, we address the risks of data transmission. All data, when sent over public networks, is encrypted using trusted protocols. This minimizes risks like man-in-the-middle attacks, helping to maintain data confidentiality.

Lastly, for Encryption in Use, when sensitive data is actively being processed, we either encrypt or obfuscate it. This protects data from threats like memory dump attacks and unauthorized system access

**Slide 7 - Triple-A**

We use the Triple-A Framework as our approach to access control.

Authentication is our first line of defense. Before any user accesses our system, they must provide valid credentials

Moving to Authorization, we adhere to the least privilege principle. Users are only granted the access they absolutely need, with every permission being explicitly defined.

Lastly, Accounting comes into play, where every user action should be logged. This not only promotes user accountability but also helps with investigations if need be.

**Slide 8 - Unit Testing**

I’ve chosen to test Safe String Handling, the subsequent slides will show a set of unit tests.

**Slides 9-13 Unit tests**

9 - This test shows that the string is correctly null-terminated.

10 - This test should throw an exception due to a string overflow.

11 - This test shows the string character access was within bounds.

12 - This test should throw an exception due to out-of-bounds access.

13 - This test should throw an exception due to a string concatenation overflow.

**Slide 14 - Unit testing Conclusion**

I am using the Google test framework since It simplifies the setup & teardown. It uses assertions to validate conditions, and its structure makes it easy to perform targeted testing.

To take testing further, we can use varying data complexities to challenge our code and discover edge cases. We can also simulate external systems where applicable to create a realistic testing environment. We can also use coverage tools to find parts of the code that aren't being tested. Lastly, we can integrate tests into our integration pipeline, and use automated testing tools to run relevant tests each time the codebase changes.

**Slide 15 - Automation diagram**

Here is the DevSecOps Diagram for reference, on the next slide I’ll explain where the automation tools fit into this pipeline.

**Slide 16 - Tools**

During the pre-production build stage, every code commit undergoes static analysis using tools like Cppcheck and the Clang static analyzer. This helps us spot and rectify vulnerabilities even before manual reviews, helping the main repository remain uncompromised. As we transition to the production health check phase, we can use Coverity since it covers multiple languages, and is especially good for finding SQL injection or poorly designed language interactions.

**Slide 17 - Risks and Benefits**

With this strategy, we have a focus on static tools, which is great for vulnerability detection and efficiency. However there is the potential for gaps in code review early on, since the tests need to be developed along with the code. It’s also worth considering dynamic threats, which this strategy really does not offer much defense against.

**Slide 18 - Recommendations**

Incorporating peer reviews and employing dynamic analysis tools are the most important first steps we can take from here since these fill the most pertinent gaps in our plan’s coverage. For the long run, we should increase the amount of training available to developers in regards to code security. And lastly, we can continue to implement defense-in-depth by adding layers of security to high-risk or crucial parts of the system.

**Slide 19 - Conclusion**  
  
Our policy is comprehensive and lays a solid foundation for the future. Keeping in mind the defense-in-depth principle, encryption, and Triple-A Framework, we've employed a broad spectrum of approaches to security. However, we still need dynamic threat analysis, more developer training, and peer reviews to elevate our security posture. These standards will help to ready our systems and keep them future-proof.