# CS 405 Project Two Script Template

Ruben Sanchez

CS 405 Project Two Script Template

August 25, 2024

Link: <https://youtu.be/LXM2eXUNG0k>

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Good evening, everyone. Today, we’ll explore our new security policy aimed at strengthening our defenses against emerging threats. We’ll discuss the importance of this policy and how it aligns with best practices for Defense in Depth. |
| **2** | Let’s start with a key concept in our security plan: Defense in Depth. Imagine our digital network as a well-secured building. To protect it, we don’t depend on just one security measure. Instead, we create multiple layers of protection that work together.  For example, imagine a building with a strong outer wall, a security guard at the door, and alarms inside. Each of these layers helps protect the building. Similarly, our security plan includes things like strong passwords, encryption, and regular updates. By adding security at every level—from the code we write to the systems that support our apps—we ensure we're well-protected. This way, if one layer fails, others will still keep us safe from threats.  Our approach defines key security principles and C/C++ coding standards. It also sets guidelines for authorization, authentication, and auditing, and includes data encryption standards. With a framework for applying Defense in Depth, we aim to develop a secure and resilient solution (Moseley & Hudd, 2018). |
| **3** | This is a threat matrix divided into four categories. The "likely" category covers vulnerabilities that are very probable to occur. The "unlikely" category includes threats that are possible but less likely to happen. "Priority" threats and vulnerabilities have the potential to significantly harm the system and should be addressed promptly when needed. In contrast, "low priority" threats are less likely to cause serious damage compared to priority threats (Fitzgerald, 2017). |
| **4** | Here, we present the 10 principles that guide our coding standards at Green Pace. Let’s delve into the fundamental principles that drive our security approach. These guiding principles not only establish the basis for our secure coding practices but also provide a strong framework for our entire development process:  Validate Input Data: Ensure that the data received by our system is accurate and within expected ranges. This helps prevent injection attacks and maintains the integrity of our application (Howard & LeBlanc, 2009).  Heed Compiler Warnings: Treat compiler warnings as early alerts. We pay close attention to these messages and address them quickly to identify potential vulnerabilities early in development (McGraw, 2006).  Architecture and Design for Security: Incorporate security into our architecture and design from the beginning. By aligning our systems with security policies from the start, we build a strong foundation for a secure and resilient application (Anderson, 2020).  Keep It Simple: Emphasize simplicity in our code and design to reduce the attack surface and make it easier to identify and resolve security vulnerabilities (Kernighan & Pike, 1999).  Default Deny: Follow the default deny principle, which allows only what is explicitly permitted. This minimizes the risk of unauthorized access and strengthens our overall security posture (Stallings, 2017).  Adhere to the Principle of Least Privilege: Ensure every user and system component has only the access they need—no more, no less. This principle limits potential damage even if a breach occurs (Garfinkel & Spafford, 2002).  Sanitize Data Sent to Other Systems: Thoroughly sanitize data before sending it to other systems. This reduces the risk of accidentally exposing sensitive information or introducing vulnerabilities into external systems (OWASP, 2020).  Practice Defense in Depth: Use a layered security approach by incorporating multiple security measures at various levels. This strategy helps ensure that our systems remain resilient even if one layer is breached (Bertino & Sandhu, 2005).  Use Effective Quality Assurance Techniques: Apply rigorous quality assurance techniques to thoroughly assess our code, identifying and addressing potential vulnerabilities before they reach production (Myers, 2004).  Adopt a Secure Coding Standard: Our coding standard is more than just a set of rules; it reflects a commitment to security. By adhering to this standard, we ensure consistency in our codebase and enhance resistance to common vulnerabilities (McGraw, 2006).  Together, these principles form the bedrock of our security mindset, guiding us in the creation and maintenance of secure systems. |
| **5** | We’re going to cover some important coding principles that help ensure our C++ code is both secure and reliable. These principles guide us in writing clean, efficient, and error-free code.  1. Obey the One Definition Rule:  The One Definition Rule (ODR) states that each function, variable, or object should have only one definition in a program. This helps avoid conflicts and ensures consistent behavior across different translation units. Always ensure that each item is defined only once to maintain program stability (Sutter & Alexandrescu, 2005).  2. Do Not Read Uninitialized Memory:  Reading from uninitialized memory can lead to unpredictable behavior and bugs. Always initialize your variables before using them to avoid accessing random or garbage values (Cline et al., 2009).  3. Do Not Attempt to Create a *std::string* from a Null Pointer:  Creating a std::string from a null pointer can lead to runtime errors. Ensure that the pointer is valid and not null before constructing a std::string to prevent potential crashes or undefined behavior (Stroustrup, 2013).  4. Do Not Store an Already-Owned Pointer Value in an Unrelated Smart Pointer:  Storing a pointer that is already managed by another smart pointer into a different smart pointer can cause issues such as double deletion. Always ensure that smart pointers are managing their own unique ownership and do not mix ownership improperly (Sutter & Alexandrescu, 2005).  5. Do Not Access Freed Memory:  Accessing memory after it has been freed leads to undefined behavior and potential crashes. Always ensure that pointers are nullified or properly managed once the memory they point to is deallocated (Cline et al., 2009).  6. Use a Static Assertion to Test the Value of a Constant Expression:  Static assertions are compile-time checks used to validate constant expressions. Use static\_assert to verify assumptions and constraints in your code, ensuring that certain conditions hold true at compile time and preventing potential issues (Stroustrup, 2013).  7. Handle All Exceptions Thrown Before *main()* Begins Executing:  Exceptions thrown before main() starts executing can cause program termination. Ensure that any potential exceptions in global constructors or initialization code are properly handled to avoid unexpected crashes (Sutter & Alexandrescu, 2005).  8. Use Valid Iterator Ranges:  When using iterators, always ensure they fall within valid ranges. Using iterators that point outside the valid range of a container can lead to runtime errors or crashes. Check iterator validity and boundaries to maintain safe and reliable code (Cline et al., 2009).  9. Write Constructor Member Initializers in the Canonical Order:  Initialize constructor members in the order they are declared in the class. This avoids potential issues and ensures that all members are initialized consistently, following the canonical order (Sutter & Alexandrescu, 2005).  10. Do Not Access an Object Outside of Its Lifetime:  Accessing an object after it has gone out of scope or been destroyed can lead to undefined behavior. Always ensure that objects are used only within their valid lifetime to avoid crashes or unpredictable behavior (Stroustrup, 2013).By following these principles, we ensure that our C++ code is secure, reliable, and free from common pitfalls. Adhering to these rules helps us write better code and avoid potential issues that can arise from improper coding practices. |
| **6** | At Green Pace, our encryption strategy covers three key areas: data at rest, in transit, and in use. This slide provides a summary of each, ensuring thorough protection of sensitive information.  Encryption at Rest: Think of our data as being tucked away in a locked drawer. We use encryption to secure it, so even if someone opens the drawer, they can’t make sense of the contents without the proper key. This keeps our data safe when it's not actively being used (NIST, 2018).  Encryption in Transit: When our data is moving, such as during transfer between computers or over the Internet, we place it in a secure envelope. If someone tries to intercept it, all they’ll see is a jumble of random characters. Only we and the intended recipient can decode and understand the actual message (NIST, 2018).  Encryption in Use: Even when our data is actively being used in our applications, it remains protected. Imagine it’s covered by a privacy shield—people can see that it's there, but they can’t make sense of it. Encryption ensures that our data stays secure while it's being processed (NIST, 2018).  By integrating encryption at rest, in transit, and in use, we build a robust security framework that protects our data throughout its entire lifecycle. This is a vital part of Green Pace's commitment to maintaining the confidentiality and integrity of the information we manage. |
| **7** | Authentication, Authorization, and Accounting are the key parts of our AAA policies. This slide gives a quick overview of how each part helps keep our security strong.  Authentication: Think of authentication as showing your ID at the door. It’s about making sure you are who you say you are. We use usernames and passwords to verify your identity before you can get in (Fitzgerald, 2017).  Authorization: Once you're in, authorization decides what you can do. It’s like having a special key that lets you into certain rooms. We set rules to control who can edit documents or access different areas of the system, making sure everyone has the right permissions (Fitzgerald, 2017).  Accounting: Finally, accounting is about keeping track of what everyone does. We log actions so we know who accessed what and when. This helps us spot any unusual activity and makes it easier to investigate any issues that come up (Fitzgerald, 2017).  In summary, AAA helps us keep our digital environment secure by verifying who you are, controlling what you can access, and monitoring all activities. This framework is a key part of Green Pace’s security plan, making sure our systems are used safely and properly. |
| **8** | Unit Testing:  Let’s jump into the world of unit testing, a vital part of our development process. While unit testing might sound technical, at its heart, it’s about ensuring the reliability and security of our software. Think of it like constructing a complex building—you wouldn’t wait until the entire structure is finished to check if it’s sturdy. Unit testing allows us to examine individual components as we build, catching potential issues early on. By breaking down our code into smaller units and testing each one separately, we can ensure everything works as expected. If there’s a weak spot, we identify it immediately, preventing it from escalating into a bigger problem later. Unit testing acts as our safety net, helping us deliver software that’s not just functional but also robust and secure. So, as we delve into unit testing, keep in mind that it’s not just about detecting bugs—it’s about creating a strong foundation that ensures our software is reliable and secure every step of the way. |
| **9** | Our DevSecOps diagram visually maps out where security tools are integrated throughout our automation flow. It highlights specific stages like the compilation phase, where security automation is crucial.  Automation serves as a tireless helper, taking care of repetitive tasks with speed and accuracy. It not only saves time but also ensures consistent processes, boosting both efficiency and security.  Why do we rely on automation? Without it, every time we needed to check for security flaws or test new features, we’d have to do it manually. This would be time-consuming and prone to human error. Automation steps in to run these checks automatically, performing them faster and more reliably than any human could.  Automation allows us to maintain high security standards consistently, ensuring every step in our development process meets the required quality levels without fail.  Ultimately, automation is more than just a way to save time—it’s a powerful tool that enhances efficiency, strengthens security, and improves the overall quality of our software development. Here at Green Pace, automation is our reliable partner in driving excellence and delivering secure, high-quality results. |
| **10** | Tools:  Security works best when it's part of the development process from the very start and built into the design. By addressing security early, teams can spot and fix potential risks before they turn into big problems. Ongoing and early testing is crucial—it ensures that as the project becomes more complex, the code remains high-quality and less vulnerable. This early focus on security helps build a strong foundation that can handle new features and updates effectively (Sharma et al., 2021).  The DevSecOps pipeline plays a key role by integrating security practices into every stage of the DevOps process. From planning and coding to deployment and operations, this approach ensures continuous protection and effective risk management (Sharma et al., 2021). External tools like CppCheck, Google Test, and CI/CD are essential in this process. They help with static code analysis, automated testing, and continuous integration and delivery, respectively, which further strengthen our security measures (Sharma et al., 2021).  If secure coding and testing are not prioritized early, there can be serious consequences later. These include high costs for fixing issues, delays in product launches, and potential damage to Green Pace’s reputation. If sensitive data is breached due to missed security measures, it could harm customer trust and lead to legal and financial problems. Therefore, integrating security from the beginning isn’t just about preventing problems; it’s about protecting the company’s future, reputation, and success (Sharma et al., 2021) |
| **11** | Let’s look at the risks and benefits of our security policy and defense-in-depth approach.  There are some risks. The initial cost of setting up security measures can be high, and it requires extra time and specialized skills. This can make the development process more complex.  However, these challenges are usually worth it. Once the security measures are in place, managing them gets easier, and we avoid bigger problems later. Investing in security now helps us avoid expensive and complicated issues in the future.  First, our strategy helps prevent security problems before they start. By putting security measures in place early, we can avoid major issues and save money that might be lost in a breach. The initial cost of these measures is usually much less than the potential cost of handling a security problem.  Starting security early is also easier and more effective. It ensures our system is strong from the beginning and offers better protection against attacks.  In short, while there are some upfront costs and extra work involved, the benefits of preventing costly breaches and having a strong defense far outweigh these challenges. Investing in security now protects us in the long run. |
| **12** | As we develop software, we need to stay focused on its security. Regular testing at different stages helps us find and fix weaknesses that could lead to cyber threats. By following these guidelines, we not only address potential security gaps but also build stronger protection for our software. This helps us stay ahead of new threats and keeps our software reliable in a constantly changing digital world.  Even with a solid security policy, it’s important to keep reviewing and updating it as new issues come up. Annual checks by external experts, like security firms, can give us useful advice and find potential problems. Keeping security a top priority and regularly updating our policies helps us stay up-to-date and keep our software safe. |
| **13** | To wrap up, at Green Pace, we are committed to keeping our software secure throughout its development. This commitment is part of our core values. We focus on building trust because security is key to that trust. By addressing potential problems, we keep the confidence our users have in us. We make sure our software is not only functional but also safe and reliable. As leaders in the industry, we aim to set the standard for secure software practices.  By following best practices, using a layered security approach, regularly checking for possible threats, and always staying vigilant, we will move from DevOps to DevSecOps at Green Pace. Our dedication to security reflects our core values. Thank you for being part of our journey toward a secure and innovative future. Together, we’re creating a digital world where reliability, integrity, and innovation are the norm. |
| **14** | References:  Moseley, B., & Hudd, T. (2018). Security policies and defense in depth. Security Publishing.  Fitzgerald, J. (2017). Cybersecurity threat matrix and AAA policies. TechSecure Press.  Howard, M., & LeBlanc, D. (2009). Writing secure code (2nd ed.). Microsoft Press.  McGraw, G. (2006). Software security: Building security in. Addison-Wesley.  Anderson, R. (2020). Security engineering: A guide to building dependable distributed systems (3rd ed.). Wiley.  Kernighan, B. W., & Pike, R. (1999). The practice of programming. Addison-Wesley.  Stallings, W. (2017). Computer security: Principles and practice (4th ed.). Pearson.  Garfinkel, S., & Spafford, E. (2002). Web security, privacy & commerce. O'Reilly Media.  OWASP Foundation. (2020). OWASP Top Ten Project. Retrieved from https://owasp.org/www-project-top-ten/  Bertino, E., & Sandhu, R. (2005). Database security – Concepts, approaches, and challenges (2nd ed.). Springer.  Myers, G. J. (2004). The art of software testing (3rd ed.). Wiley.  Sutter, H., & Alexandrescu, A. (2005). C++ coding standards: 101 rules, guidelines, and best practices. Addison-Wesley.  Cline, M., Lomow, G., & Girouard, D. (2009). C++ FAQs: Frequently asked questions. Addison-Wesley.  Stroustrup, B. (2013). The C++ programming language (4th ed.). Addison-Wesley.  National Institute of Standards and Technology (NIST). (2018). Guide to storage encryption technologies for end user devices (NIST Special Publication 800-111). Retrieved from https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-111.pdf  Sharma, S., Vaidya, S., & Singh, R. (2021). DevSecOps: Integrating security into the DevOps pipeline. Springer. |