# CS 405 Project Two Script Template

Complete this template by replacing the bracketed text with the relevant information.

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hi everyone, my name is Cade Bray and I’ll be presenting the security policy for Green Pace. |
| **2** | In this security policy you can expect to see the ten core security principles, ten defined coding standards, our encryption strategy, the Triple-A framework, implementing unit tests, Automation implementation options, Risks and benefits, and finally, the Recommendations and Conclusions of this policy.  The security policy at Green Pace was a natural next step as their security mission is enacting defense in depth however possible. Defense in depth is a strategy that takes a singular security measure which may be insufficient by itself but can become a robust security strategy when paired with many other singular measures. Think of each security measure as a metal ring. Holding metal ring in front of you would prevent little to prevent an attack but when linked together to form chainmail becomes an impenetrable defense.  In this threat matrix we can see some of Green Pace’s identified security risks which will be addressed in the upcoming coding standards.  Automation can help resolve a large portion of these issues. Automation tools might include static code analysis like CPPCheck. In this screenshot we can see an example of some questionable code in CPP check. The static code analysis tool will provide an analysis with different severity indicators and even industry standard issue descriptions to help developers resolve issues the first time. |
| **3** | In the next few slides, I’ll be covering the ten coding security standards.  - Input validation is one of the easiest to secure an application but is easily overlooked or not fully encompassed. Validating input isn’t a singular solution but instead a methodology for every variable type. Validating data for strings might include verifying the length of the variable is within size bounds to prevent buffer overflows. This validation is not concerned with the meaning of the string but the data itself. Validating a string for its intent can be useful for preventing SQL injections by preventing keywords or syntax within the string. This might include entire refusal of the data or sanitizing the data by removing aspects to alter its intentions. (Seacord, 2013, p. 44)  - Modern compilers are becoming a layer of security. In the GNU C (GCC) compiler flags can be passed to the compiler to check for various security mechanisms. These mechanisms include buffer overflow prevention by not overwriting surrounding variables but don’t throw any visible errors when a buffer overflow is detected unless specified with a different flag. Additionally, compilers are now tracking variable sources by called them “tainted” and propagating that flag with any variable that might utilize its data. This allows developers to protect vulnerable internal functions from being used with tainted variable types. (Seacord, 2013, p. 125)  - “The architecture and design of a system significantly influences the security of the final system” says security author Robert C Seacord. The author continues to say that if architecture or design is flawed, no standard can make your system secure (Seacord, 2013, p. 172). For instance, if a system can be divided so that minimal portions of code have access to privileged security levels we can mitigate our attack surface to a singular section of code.  -When projects begin to grow people tend to complicate them. Programs should be small enough to understand without compromising the original designs intention. When a mechanism is easily implemented and verified everyone can feel confident in the project’s integrity (Seacord, 2013, p. 173). When a projects team is able to maneuver its design easily it will not only hasten the development efforts but will allow haste in understanding the program for maintainers when vulnerabilities are found. Assuming your program will not encounter a vulnerability because ‘we built it right the first time’ is no more than hope as a security tactic.  - In a program or in terms of a user we should by default deny all privileges. We should explicitly allow a user to have any form of privilege whenever reasonably possible. This plays on the principle of zero trust where we always verify and never assume. This could apply to continuously authenticating a user’s bearer token or requiring user privileges to be regularly renewed (Lindemulder & Kosinski, 2024).  - Like default denying, in the principle of the least privilege we won’t always have the benefit of denying all access potentially because of business rules dictating an acceptable level of risk to cost benefit. In this event we should incorporate the least privilege to a user, program, database, or application whenever possible. Take for example a program responsible for changing a password. Passwords are naturally maintained by higher privileges but if a program were to have an exploit that allowed unplanned queries or events to occur, we could affect all components on an application or database by allowing a blanket privileged policy.  - Sanitizing data is an important task for receiving input to ensure that the input isn’t malicious, but project teams commonly forget to sanitize data sent. Trimming the data sent won’t just improve performance between systems but enhance your operational security as well. When every system does its part not just to protect itself but all operational components of an application you start to build a robust security atmosphere that embodies defense in depth ideals. Additionally, data sent to another system may contain privileged information that you’re attempting to protect. In a zero-trust application you must assume that the system you’re interacting will is compromised and should only receive the minimal data possible to perform its task.  - Defense in depth is a practice of layering multiple security mechanisms at various levels. This may extend development time and increase complexity but the trade off is an application with layers of redundancy. Betting on one security mechanism to work is poor practice because hackers are constantly evolving and by layering multiple security types together, we ensure that no singular development in exploitation can jeopardize our systems (Seacord, 2013, p. 182).  - Just as exploitation is consistently evolving as is security specialists. Security research allows us to bolster security mechanisms, but it isn’t always needed to recreate the wheel. Educating security experts on the latest trends is a singular aspect to a robust project team. When we contribute to industry standard security policies, we can retain our development and research time and devote those precise resources to unknown issues your specific application may have. Common policies have been tested from various angles by security research organizations that may have larger resource pools.  - When we adopt secure coding standards, we are allowing ourselves to take advantage of already known and harden defense mechanisms that haven’t failed other organizations. This allows us to keep implementations conformed across every application or component in your system. Instead of having four different query sanitizing approaches implemented for your application you’re faced with a singular implementation that can be easily verified. |
| **4** | - Using appropriate data types that are driven to the exact uses of your expected data not only makes a project enhanced in terms of efficiency but security as well. For example, if you’re expecting a field on an application to only be 10 characters long it may be more beneficial to use the ‘char’ data type over a string data type (NUM00-J. Detect or prevent integer overflow, 2024)  In these coding standard we’ve picked some security concerns that align with each of these topics. Starting off with Validating input and data:  - For Data initialization standard we’re discussing the data values. This standard focusses on data initialization. Your variables should always be initialized to a starting value to avoid unintended results. This might include initiating a string to a blank string value, so it doesn’t assume the value of NULL initially or initializing an integer to a value (EXP53-CPP. Do not read uninitialized memory, 2023)  - String operations can pose interesting effects if not handled correctly. This standard focuses on using correct replacement techniques. (Use valid references, pointers, and iterators to a basic\_string, 2024). Whenever possible using generic algorithms such as replace we’re able to enact industry standard algorithms that have been proven to resist a hacker’s grasp, are well documented of any vulnerabilities, or will be quickly updated as a collective effort to resist vulnerable code.   - SQL injections can be at the top of the list for most detrimental attacks because they target the most valuable asset a program has, its data. An application should go through the effort to sanitize queries and remove potentially harmful keywords, key characters, or should use prepared queries (IDS00-J. Prevent SQL injection, 2025).  - When an error occurs, we can normally gracefully exit the code but when we access memory that has been deleted, we get what’s called a segmentation error. These errors are difficult to troubleshoot and can cause unexpected results in your application. For these reasons and many more we should be protecting our memory from these types of calls (MEM50-CPP. Do not access freed memory, 2023).  - Assertions are a powerful tool found in most object-oriented programming languages. These tools allow you to make ‘rules’ that must be followed in your code such as ensuring a variable size stays below a specific size. This can be useful in many scenarios but as an example we can imagine an environment that is heavily controlled by its resources such as an embedded system which may be operating with a lightweight operating system such as a 16-bit OS. This would become an issue if we planned to write four bytes of data to a 2 byte integer as defined by a 16-bit OS (DCL03-C. Use a static assertion to test the value of a constant expression, 2025).  - Some objects or functions do not utilize automatic destructors for various reasons. The most common way these are handled is with an exit handler. Exit handlers are called with the atexit() function call. When the program is terminated it will execute code within all exit handlers but only if the program doesn’t come to an abrupt stop such as when exit() is called. If an exit handler itself errors out it will prevent further exit handlers from executing. It’s important that all exit handlers have robust error handling to ensure other exit handlers may execute.  - Data is the ‘life blood’ of any application and reading and writing from files are frequent operations in an application. There comes a time when you need to read a file, interpret, and rewrite the file. In the event that you don’t close the file and reopen it under a new fstream() object we can have undefined results when the accessing it the second time without resetting the file position (FIO50-CPP. Do not alternately input and output from a file stream without an intervening positioning call, 2023).  - Many developers struggle to grasp lower-level languages as you are needed to handle memory carefully. In C++ we’re required to use pointers for many operations. It’s imperative that we do not delete an array through a pointer of the incorrect type (EXP51-CPP. Do not delete an array through a pointer of the incorrect type, 2025).  - When working with strings it’s important to work with them carefully as creating a buffer overflow is extremely likely. We need to check that the string has room for all the data we’re assigning it and a final null terminator character (STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator, 2025). |
| **5** | Encryption is a must have to implement no matter what you’re working on. Keeping information safe is what retains consumer trust. So, what are the different types of encryptions and what can we do about it? Well let's start with:  Encryption at rest - At this time the data is stored in hard drives typically and can be vulnerable to physical attacks such as theft of the hard drive. To mitigate this, it’s recommended to encrypt the drive with services such as BitLocker which use AES 256 encryption.  Encryption in flight - During this time the data is being transferred over potentially unsecure methods and needs to be encrypted with TLS 1.2 or higher. This might include implementing HTTPS to sites to ensure that data is transferred and interpreted by only the two parties. This prevents man in the middle attacks and many others.  Encryption in Use - During this time the data is being worked upon by an application or user. This means that the data is loaded in memory and could be at risk of being stolen from malicious programs with access to that memory. This is especially important when it comes to SPII or other highly sensitive data sources. |
| **6** | The Triple A Framework stands for Authentication, Authorization, and Accounting, but what does that actually mean, well…  Authentication is an important step to the Triple – A Framework. During this step we ensure that all users undergo checks to ensure their identity. Adding as many noninvasive checks as possible is important. This could include multi-factor authentication (MFA) and enforcing strong passwords.  Authorization can be summarized as giving access to users that are minimal and exactly what is needed. Ensuring a user cannot utilize sections of a system restricted to them is a key portion of the Triple A framework. This might mean that users cannot access data from other users for example. This ensures that databases are managed with the least privilege necessary. Creating permission levels such as staff and average user is important distinctions at this stage.  All actions that a user makes need to be logged in detail to ensure that a recreation of events could occur. This makes the job of a rollback that much easier if you understand the actions that were taken. |
| **7** | Unit testing can be intimidating to many developers, but as the need for code that works the first time becomes increasingly important, we’ve seen leaps in how our unit testing frameworks behave. Most languages have a popular or recommended unit testing framework. For example, Google created the GTest framework that allows for a robust set of assertion tools. These types of frameworks are what allow developers to make a test-driven development setting manageable and worthwhile. |
| **8** | Automation can be implemented into almost every area of DevSecOps. Assessing and planning could lead you to tools that determine your threat landscape with various dependency checker, Designing a program means that you can build test driven development unit tests that are automatically ran when a pull request or push to a repository is done, Building your program isn’t any different with access to static code analysis tools like Clang or CPPCheck. Verifying and testing automation could include vulnerability scanning tools such as provided by OWASP.   Once your project is deployed you can rest easy as well as most production environments can be automated with docker or penetration tested with countless automation that are packaged together in the Kali Linux distribution for example. When you’re monitoring your program, you can implement event alerting and intrusion detection. Based on certain rules that you’ve configured we can also block suspicious activity on a platform. Finally, Maintaining and stabilizing can be a breeze with platforms such as AWS or Google cloud. This allows your application to dynamically scale its resources to user surges. |
| **9** | The risks of waiting to implement these standards, automation, and techniques can be costly. People like to save security until the end, but that can keep your sensitive data at risk and your consumer trust on the line needlessly. The longer you wait the more time it gives malicious actors time to canvas your attack surface finding your weaknesses.  By starting early with even basic security standards you can start your defense in depth journey. Each standard that you build acts like a brick. Once many layers are enacted you will build a wall capable of handling an array of issues that can be audited and verified time and time again. |
| **10** | Moving forward implementing all of the above standard coding standards can prove useful but focusing on standards are most directly inline with your application allows you to place key measures in action to create peace of mind early.  This security policy while it may seem like it has touched on a lot is only the surface as it focused mostly on coding standards and development. Physical security, network security, cloud-based security, and much more are further considerations to consider to better protect yourself. |
|  |  |