# CS 405 Project Two Script Template

Project Two: Security Policy Presentation

Justin Phillips

10/18/2025

<https://youtu.be/V3h1hQHya7I>

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

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Good morning, my name is Justin Phillips. I have been asked to review our policy today with this presentation, and I hope to give everyone a refresher on our best practices. |
| **2** | [Introduce your security policy. Explain why it was needed and how it will be used to support the defense-in-depth best practice.]  Here is our overview. We have a few goals here, but they ultimately cater to the policy of Defense in Depth. The strategy is that we will be following features principles, coding standards, triple-A security techniques, and encryption techniques to protect our data in overlapping layers. This helps protect everything we write, store, and process from attackers. |
| **3** | Here's just another better look at the Defense in Depth visual. The SOC operations are going to be outside of the scope of this presentation mostly, though we might touch on some things, but what’s important here is that we are focused on security in every layer of developing and running our programs. We are going to go in depth with static testing, endpoint and host security, network security, perimeter security, cloud security, and yes, we will also cover physical security. |
| **4** | [Populate the Threats Matrix table and provide explanations to summarize of all of your security risks.]  An essential part of our approach to security is modelling the threat, and understanding which vulnerabilities are our highest and lowest priority, and which are most and least likely to occur. We can expect a variety of threats to occur. What we know will occur are the typical situations – Phishing, attempts to spread malware to our systems, SQL injection attacks to access our database, but we also need to be aware of and work to prevent software vulnerabilities such as setting incorrect data values or choosing the wrong types during development, validating strings, protecting memory and resource usage. Every vulnerability is a chink in our software’s armor and quality.  Less likely are going to be issues we would expect from our team. Incorrect data types do fall in with this lot in the bottom right, as well as issues with implementing assertions, choosing obsolete methods for random number generation, logical errors such as wait function wrapping, and while physical threats are generally unlikely, we will be accounting for them.  Let’s move to the priorities. Our number one priorities are going to be setting up a secure infrastructure, ,meaning physical security and firewalls that protect us as we work, but also policy choices such as deny by default, and paying good attention to choices with data types, validating data values, defending against SQL injection attacks, defending against malware, but also defending against memory leaks, unhandled exceptions, poor resource management, and using correct techniques to generate random numbers.  Our lowest priority threats are still on our radar, but come down to us using our tools correctly as developers – Assertions and wait function wrapping should be used correctly and employed logically. |
| **5** | [List the 10 coding standards. Explain your own ranking system for vulnerabilities, using specific details from the coding standards in your security policy.]  Our team is employing various standards which you can see here. We have a pretty clear naming scheme, which is our standard, what number standard it is, and the language it applies to. What we want to do here is be mindful as we develop. All of these standards are about building robust code, and things to do and not to do. Let’s start from the top and work our way down. Remember that this is all in our coding standard guideline document that you can find on our Sharepoint, and if you’d like any code examples, there are examples for each of these.  First, we want to be careful selecting our data types. Between different systems, ints automatically allocate different bit widths. This means some operating systems may allocate one byte to an int, but others may allocate two. If we don’t define our bitwidth, we are allowing circumstances where the width may be different between two machines processing the same or similar data, and one program may crash while the other continues to run. If we want consistency across all of our users, we should fix with width of our variables so they are always known and accurately accounted for.  Next, we want to be checking our limits and bounds whenever we are modifying data or defining it. This is pretty obvious, we do not want overflow or underflow errors to be possible within anything we write.  String correctness for us refers to using the correct string data types, the standard library string function works great because it automatically allocates the correct bit width unlike character arrays. Using character arrays is a vulnerability for fields of an unfixed length, and is generally an outmoded practice.  Using parameterized queries for SQL is a big one. SQL injections are a very common attack, and are easy to pull off. We prevent this by telling SQL the next command to expect by formatting parameterized queries, which keeps our database safe from dumps, deletes, and illegitimate access.  We do not want to access freed memory or dereferenced pointers. This means successfully detecting end of data streams, not accessing removed or deleted objects, and handling missing information.  We want to use assertions specifically for validating our own assumptions about how our code is going to work in order to catch logical flaws. We also do not want the assertions to be structural functions – Compiling a release build without these assertions should not impact the end user experience.  Whenever an exception is thrown or expected to be able to be thrown, it should be anticipated, caught, and handled by our software.  When an object fails to construct, it should be deconstructed. When exceptions are handled, working data should be gracefully removed to avoid resource leaks.  We should be using up-to-date techniques to generate random numbers for cryptography.  Finally, when we have conditional wait functions, they should be wrapped in a while loop to ensure the thread times itself properly for the condition to be met. |
| **6** | [List the 10 coding standards. Explain your own ranking system for vulnerabilities, using specific details from the coding standards in your security policy.]  On this page, we can review those coding standards in relation to their severity, likelihood, remediation cost, and priority. You may notice that priority is a function of severity, likelihood, and remediation cost. You will want to note the priority level, in which 1 is our most urgent quality control standard, and 3 is our least. Vulnerabilities that are likely to be used by attackers are typically overflow/underflow bounding errors, using char arrays rather than strings which also could lead to a bounding error, and SQL injection concerns.  Our next highest rung of priority is ensuring standards that relate to the ability of the program to run properly on end user systems indefinitely. This means verifying we are assigning variables their intended bit width explicitly, not accessing freed memory, handling all exceptions, and not leaking resources.  Our lowest priority are standards that might not have a substantial impact on security or software functionality immediately. This has to do with avoiding programmer errors by implementing assertions, not using the outdated std::rand() function, and being sure to wrap wait functions in loops to allow threads to sleep until a condition wakes them. |
| **7** | [List the 10 principles. List the coding standards that apply to each principle. This should demonstrate the alignment between principles and standards.]  These standards are all guided by our ten principles, which are fundamentally mission statements in relation to continuous code quality. We want to be certain we follow these every day.  First, we emphasize validating input data. Allowing invalid data to come through is a sure way to open a vulnerability within the software we produce.  Second, we want to heed compiler warnings. If the compiler throws any errors, follow them up. Investigate it and close it.  Third, architect and design for our security policies. This means considering all of those standards we went over through the planning phase all the way to the maintenance phase.  We also want to keep it simple. We do not want overly complicated programs which could lead to sloppy or inefficient code.  Default deny; We do not want vulnerable ports or accessible hidden endpoints, so it is best to deny access to our application resources by default, waiting for essential authorization checks to validate the request.  Principle of least privilege is the notion that a process, user, or routine should only have the permissions necessary for the functions they need to use to perform their job functions. We do not want one set of compromised credentials to be able to access and dump all data.  Sanitizing data that is being pushed into transit is also important. We do not want to feed incorrect or malicious information into MongoDB, for instance, or SQL.  We also want to practice defense in depth. As discussed before, defense in depth is constructing several overlapping layers to armor our program.  We also want to ensure our quality assurance techniques are effective. This means not wasting time on ineffective techniques.  Finally, we want to adopt secure coding standards. This means being sure that our policy and our way of writing software is industry standard compliant, as well as that we are using modern and relevant functions. |
| **8** | [List the 10 principles. List the coding standards that apply to each principle. This should demonstrate the alignment between principles and standards.]  Here we can see the relationship between our principles and our standards. Many of these items overlap, which is just want we want to see in Defense in Depth infrastructure. Abstract principles used in the planning phase might not be tied directly to our standards when writing software code, but they are still important to keep in mind as we continue to build our software infrastructure. Default deny and principle of least privilege, for instance, have to do with web configurations, the design of our endpoints, and the design of our permissions and accessibility systems, and are principles which guide us in developing the overarching structures of our software. |
| **9** | [Explain the policies for encryption in flight, at rest, and in use.]  Everything we’ve gone over so far is in the service of defending our data, and one of the best technologies we have available as developers to do this in the modern age is to encrypt it. Encryption can protect data in flight, or transit. It can also protect data at rest, or in storage. Finally, encryption can also defend our data during use, or runtime processing.  Flight involves defending our data between systems including over internet. Our best tools for these are selecting encryption algorithms that will keep our data safe, such as SHA-256, or using secure data transmission protocols such as HTTPS.  Rest involves defending data on both our systems as well as the client. For us, this means encrypting our codebase and access points with tools such as BitLocker as well as using multi-factor authentication to log into our databases or codebase and alter or view them.  Use involves encrypting data before sending it through the CPU and then decrypting it, which can be done using tools such as AES using approved libraries such as perhaps libsodium. |
| **10** | [Explain the policies that support authentication, authorization, and accounting.]  We also strive to follow triple-A policies, which are policies regarding authentication, authorization, and accounting. Authentication takes place at several stages. Whenever data is requested to be accessed, viewed, retrieved, or augmented, and authorization check should take place to ensure an authorized user or process is submitting the request. We can implement this by having tokens in header data whenever a packet is sent from client to server, for instance. This involves things such as the TCP handshake process, as well as multifactor authentication for users and administrators.  Authorization is control over the rights of the person or process accessing our program’s data. We should strive to allow the minimum amount of access for any particular role through least privilege and deny by default.  Accounting is accruing forensics records for data transactions, such as login histories, access and write requests, and monitoring data for compliance with expected activity. If someone downloads terabytes of personal data, we should have the credentials, the address, the timestamp, all of that data on a recording. |
| **11** | Let’s move onto our unit testing standards. These tests are written using the Google Unit Testing Framework, which comes natively implemented into Visual Studio 2019-2022. This is the data structure we will be testing. It’s pretty straight forward, and compliant with Green Pace standards.  You can see we give collections a unique pointer, and we have constructors and deconstructors for Google Unit Testing’s vernacular. It is a straightforward vector function with the ability to extend the size of the vector with so and so many random values between 0 and 100. Rand is used here, but that is just for ease of example, and not something we’d want to do in the field. |
| **12** | [Identify the coding vulnerability you chose to test.]  We perform both positive and negative tests, the latter being tests where an exception is expected. Here you can see we want to throw an error when a 5-entry-long collection is accessed at entry 10. We call this test exactly what it is, a test for Out of Range Exception Thrown When Callling At With Index Out of Bounds. |
| **13** | [Identify the coding vulnerability you chose to test.]  This test is for reserving greater than the maximum possible length of a collection. As you can see, this is also a negative test. Both exceptions get thrown. |
| **14** | [Identify the coding vulnerability you chose to test.]  We also like to test in sets. These are reciprocal tests that verify a collection can be resized positively, that is to increase, and that a collection can be resized negatively, that is to decrease. |
| **15** | [Identify the coding vulnerability you chose to test.]  This suite of tests all verify different ways to empty a collection. Resizing a collection to zero, using the clear command, and erasing elements between the beginning and end of the collection all empty it out. |
| **16** | Testing takes place throughout the entire process using automated tools. We use tools like OWASP to automatically test libraries for security vulnerabilities as we plan. During the build, we use automated tools such as our compiler’s error checker as well as CPPCheck and Google Unit Testing to verify our code is secure and functional. We move into the testing phase where vulnerability scanners run as well as penetration testers. We continue to test during the transition from build to deployment, and we use our accounting to monitor activity and detect issues with vulnerabilities such as intrusions.  We remain dedicated to security through the maintenance phase and the lifetime phase, responding to attacks and maintaining our software. The practice of incorporating security into the entire software development life cycle is called DevSecOps. |
| **17** | Here is a quick overview of the tools we might want to use. We’ll go over these a little more in a moment. |
| **18** | [Explain the DevSecOps pipeline.]  [Summarize the external tools and where and how they are used in the context of the diagram.]  These tools are pretty diverse. CPPCheck is a static analysis tool that can review a visual studio project’s code base for bugs and vulnerabilities that a compiler may not be able to recognize. Coverity is a static application security testing tool, and it executes our software to run vulnerability and bug tests, as well as quality defects. Intruder is a tool that runs on our live service and assesses our host framework such as AWS for attack surfaces, including services which may have vulnerabilities that are acting as a component of our project. |
| **19** | [Describe the problems, the solutions, and the risks or benefits involved if you act now or wait. Where is the strategy lacking? What are the risks of using this strategy? Which steps should be taken?]  Let’s take the time now to go over the benefits of incorporating DevSecOps into our development ecosystem as well as the risks of not doing that.  When we implement security from the start, it is easier to build security into the entire application as well as account for it in time, budget, and compute. We also then comply with industry standards, maintaining an appropriately high level of code quality that lets us compete in the computer science market. Finally, we can keep data secure, which builds user trust and avoid breaches.  If we implement security late, it can become costly to retrofit security policies and code into the existing codebase. It also increases complexity and makes it messy to work on the existing codebase. Over time, there is also an increased risk of a breach. Finally, we run the risk of reputational damage and punitory action if we do allow a breach to take place or a vulnerability that was otherwise preventable to be exploited.  SecDevOps is a policy that allows us to be serious about competition and data security. |
| **20** | [Identify gaps in the security policy.]  Our security policy is still undergoing adaptation to the modern software development ecosystem, and we are still determining many of our standards. Here are my recommendations:  Multifactor authentication should be mandatory for accessing our systems, especially our essential infrastructure. We also need to choose reliable hosts. Amazon Web Services has a suite of scalability and security features that make it appealing and reputable.  We should form a robust data collection and storage policy including lifespan of collected data as well as a finite amount of data to collect on users, use cases, and client hardware.  Finally, we need to define a rigid list of approved encryption techniques and protocols. SHA-256 is good for transit, AES is good for runtime. We should also define a list of explicitly approved and disapproved data transfer protocols. HTTPS, SSH, and TLS are reliably secure. Telnet, HTTP, and FTP are reliably insecure and dangerous to implement into our development ecosystem. |
| **21** | [Identify standards that should be adopted to prevent future problems.]  To conclude, we should be adopting the full suite of these standards and principles into our daily work. Specific areas I would like to emphasize from this presentation are the implementation of principle of least privilege, default deny, keep it simple, authenticate, authorize, account, and follow SecDevOps’ fullstack security implementation pipeline.  I will also ask you to review our policy semi-annually. Whenever we receive a policy update, it will be announced via email for your review. We will also be keeping a changelog of what changed and why. Keep yourself updated and refreshed on our policies and best practices, and keep in mind this is a living document. Things may be subject to change as we continue to build our best techniques for maintaining security. |
| **22** | [Provide APA-style references with links to resources, articles, and videos that you used in your presentation.]  And here are the works cited. When you receive a copy of this presentation you can read up on these to learn a little more about our concepts today.  Thank you. |