# CS 405 Project Two Script

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Hello! This presentation will cover Green Pace’s policies regarding secure software development. As part of the developer team myself, I’ve come to realize the importance of consistency and best practice when it comes to secure coding, testing, and quality assurance. |
| **2** | The goal of a policy like this is to protect data and assets both here at Green Pace, and anyone who uses our applications and services. Defense-in-Depth - illustrated in the diagram here – is enforced by making sure that each standard covers at least one core security principle, discussed more on slide 4. |
| **3** | Here, the ten proposed coding standards - which are covered starting on page 5 of the policy - are each placed into a quadrant of the matrix. Not all threats are the same, so those such as SQL injection that are higher priority but still relatively unlikely to occur are placed in their own category. Similarly, medium severity threats such as possible RNG manipulation by using rand() are placed in their own ‘unlikely’ category.  After the initial build, the code is run through many automated static analysis tools to ensure these vulnerabilities are caught early. |
| **4** | This slide lists what each of the ten security principles are, and which standards enforce each one. To enforce Defense-in-Depth, there is some overlap and redundancy. |
| **5** | And here is a table summarizing the standards ordered by priority. This priority number is determined by the three previous factors – the threat’s severity, likelihood of the threat occurring, and the expected remediation cost if it occurs. |
| **6** | Any important data should never be left as plaintext, so here we have three different policies in place, depending on how it’s being used.  Encryption at rest applies to all data being stored, whether on a local hard drive or within a remote database.  Encryption in flight pertains to all data being transferred or sent. Data should always be encrypted during transit on both ends for network communication.  And encryption in use applies to all data actively being utilized or modified. |
| **7** | The other set of three policies are what are known as “Triple-A” policies.  Authentication is what is used for verifying all users within our systems.  Authorization gives users their actual needed permissions, which is based off their current job responsibilities and requirements.  And accounting is enforced by keeping in-depth logs of all system and file changes. |
| **8** | The next five slides show examples of unit tests that are conducted as part of pre-production, using the Google Test framework. This first ensures that a vector is empty when created as intended. Although unlikely, data already in a vector by unauthorized means would cause many potential issues. |
| **9** | This test checks for buffer overflow by inserting a very high, unsupported value. |
| **10** | And this one is run to make sure that undefined behavior does not occur, in cases where capacity was not properly increased. |
| **11** | Another check related to data corruption, which makes sure that data is inserted and in the right place within the vector. |
| **12** | And this last example is an out of bounds check, protecting against more undefined behavior possibilities.  For better efficiency, many of our unit tests could be improved by using parameterized tests, which wouldn’t necessarily affect performance or results, but would result in cleaner and more readable code. |
| **13** | The next slide will discuss some of the tools used for automation, and others that can be added to the existing set. Here in this diagram, we can see a summary of how our DevSecOps pipeline currently functions. |
| **14** | The compiler is used as a “first defense” during the build phase, as Visual Studio for example typically lists important warnings for some of the more common vulnerabilities.  Once we have a successful build that we’re happy with, many static analysis tools are used during the verification and testing phase to catch what the compiler didn’t. There are several, but this slide lists two of the more common ones.  Once we reach production, automation is still used for logging purposes during the monitoring and detection phase. Automatic rollbacks are not yet used but could be instated based on the detected threat and its severity. |
| **15** | In general, the sooner an attack is detected, the less damage it will cause. Malware and social engineering are two major examples where history of these types of attacks can be used to showcase how damage can be mitigated by prompt action.  Overall, learning from mistakes can help redefine and better reshape the policy. |
| **16** | A DDoS attack is just one example of something not strictly covered by this policy, so it may be something to consider adding protocols for in the future.  I also included some possible changes to the DevSecOps pipeline, which mostly involve moving up vulnerability testing and logging to earlier points in the development cycle. |
| **17** | Since the current policy is of course not perfect, I added a couple extra points on protocols that could be expanded on in the future.  Unit testing in particular could have stricter guidelines that cover more of our standards, such as input validation and SQL injection. |
| **18** | And that covers everything I wanted to talk about today! I hope this was informative, and please feel free to contact me with any questions or suggestions. |