Southern New Hampshire University

CS 405 Project Two Script Template

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

Slide Number	Narrative
1	My name is Bryce Jensen and this is my Security Policy Presentation for Green Pace.
2	Our security policy is designed to guide the secure development of applications at Green Pace and ensure compliance with industry standards. It is part of our defense-in-depth strategy, a multi-layered approach to protecting systems and data. The policy applies to all software developers and IT staff at Green Pace. The policy covers key areas such as coding standards, encryption, and authentication, ensuring a structured approach to security risks.
3	The Threat Matrix outlines vulnerabilities we've identified, categorizing risks by severity, likelihood, and remediation costs. [Gesture at LIKELY and UNLIKELY] Here, all the Coding Standards are ranked by how likely they are. [Gesture at PRIORITY and LOW-PRIORITY] Here, the Coding Standards are ranked not just by likelihood but also by the priority they should be given when encountered. A priority greater than P10 means an extremely high priority, where a lower number signifies a lower priority. This doesn't mean they are not essential to take care of; they will just be bumped lower if a larger priority case shows up. Automation will play a key role in mitigating these risks, enabling consistent detection and resolution of vulnerabilities.
4	We adhere to 10 core principles to ensure secure development. These principles, such as validating input data, adhering to least privilege, and implementing defense-in-depth, guide our coding practices and align with recognized best practices. They are as follows, sorted by priority: ● **Sanitize Data Sent to Other Systems**: Clean data before sending it to external systems to prevent injections. ● **Validate Input Data**: Always verify user input to prevent attacks like SQL injections. ● **Least Privilege**: Grant only minimal permissions required to perform tasks.

Slide Number	Narrative
	● **Architect for Security**: Design software with built-in security from the start. ● **Defense in Depth**: Layer multiple security measures to protect systems. ● **Heed Compiler Warnings**: Address compiler warnings to fix potential vulnerabilities early. ● **Use Effective QA Techniques**: Regular testing and audits ensure security flaws are caught. ● **Adopt Secure Coding Standards**: Follow consistent coding standards to minimize vulnerabilities. ● **Default Deny**: Deny access by default, allowing only necessary permissions. ● **Keep It Simple**: Avoid complexity to reduce hidden security risks.
5	We've developed 10 coding standards to address the most critical security vulnerabilities. These include the One Definition Rule, ensuring proper use of signed data types, and preventing buffer overflows. Each of the Coding standards and a summary of each are listed as follows: 1. Follow the One Definition Rule (ODR) ● Each program can have only one definition for each (non-inline) function or variable as multiple definitions can cause errors. 2. Ensure Proper Use of Signed Data Types ● Integer overflow can lead to buffer overflows and potential security vulnerabilities. 3. String Storage Should Have Enough Space for Character Data and the Null Terminator ● Copying data to a too small buffer can cause overflow, leading to potential vulnerabilities. 4. Sanitize Data ● String data passed to subsystems can contain characters responsible for triggering commands, resulting in a vulnerability called SQL Injection. 5. Store New Values into Pointers After Being Freed ● Double-free and access-freed-memory vulnerabilities are possible when there are dangling pointers. 6. Static Insertions Should Be Used to Test Constant Expression Values ● Assertions are a tool for finding software defects in a system. static_assert() is more useful when working with server systems or embedded programs. 7. Ensure Exception Safety ● Proper error handling is necessary for software to operate correctly and avoid undesirable effects. 8. Do Not Invoke Virtual Functions from Constructors or Destructors

Slide Number	Narrative
	● Calling virtual functions can be tricky during an object's construction or destruction. The base class's version of the function is called, not the derived class's version. 9. Close Files That Are No Longer Needed ● Resource leaks can occur when files are left open unnecessarily. Closing them as soon as possible ensures the program's security and stability. 10. Don't Terminate a Program Abruptly ● std::abort(), std::quick_exit(), and std::_Exit() can be used to terminate a program immediately and should only be used with critical errors.
6	Encryption is a key part of our strategy to protect data. ● **Encryption at Rest**: Encrypts data stored on disk to prevent unauthorized access even if the storage medium is compromised. ● **Encryption in Flight**: Protects data during transmission using protocols like TLS/SSL to prevent interception. ● **Encryption in Use**: Secures data while it's being processed, ensuring protection during active use in memory.
7	Our policy enforces strong authentication with MFA, ensuring that only authorized personnel access systems. ● **Authentication**: Verifies user identity through methods like multi-factor authentication (MFA). ● **Authorization**: Grants users access to only the systems or data necessary for their role. ● **Accounting (Auditing)**: Tracks and logs user activities for auditing and compliance, helping detect security incidents.
8	Here are some examples of how we could implement security into our systems. These are unit tests for memory and the storage of data.
9	This test checks that reserving more space increases the array's capacity but keeps the current size unchanged. This could be further tested by reserving a space equal to the existing capacity to verify that there is no change in size or capacity.
10	This test verifies that attempting to access an element outside the bounds of the collection throws a std::out_of_range exception. Further testing could mean attempting to access a negative index and looking for a similar exception.

Slide Number	Narrative
11	This test ensures that duplicate values can be added to the collection and that their order is maintained. Further testing might include checking that adding duplicates to the collection doesn't increase memory usage through pointers.
12	This test ensures that accessing an element in an empty collection throws a std::out_of_range exception. Further testing could include adding an element to the collection after throwing an exception to check whether it works correctly after an exception is thrown.
13	A brief description of the DevSecOps process for a project might look like this. In the Pre-Production Phase: 1. **Assess and Plan**: We start with security risk assessments at the planning stage to identify potential vulnerabilities early on. 2. **Design**: During the design phase, we enforce secure coding standards through automated checks, ensuring that security is considered right from the beginning. 3. **Build**: Here, we implement automated security scanning tools, such as Snyk, to check for vulnerabilities introduced by external libraries. This is where the compiler comes into play, integrating security checks during the code compilation process. 4. **Verify and Test**: We add automated security scanning to our testing processes to ensure ongoing compliance. Moving on to the Production Phase: 5. **Transition and Health Check**: We conduct compliance and penetration tests to ensure our system adheres to established security policies. 6. **Monitor and Detect**: We deploy Intrusion Detection Systems to identify and monitor potential security incidents. 7. **Respond**: Automated incident response systems are implemented to reduce the impact of security breaches swiftly. 8. **Maintain and Stabilize**: Finally, we conduct continuous security audits to ensure that our systems remain secure over time. In summary, embedding security at each stage of the development cycle not only enhances our overall security posture but also ensures that security becomes an integral part of our development culture.
14	Some of the Risks and Benefits of switching to a DevSecOps process: **First, the current problems**: 1. **Security Gaps**: Traditional DevOps often overlooks security, leading to potential vulnerabilities. 2. **Increased Breaches**: The absence of proactive security measures raises the risk of data breaches and compliance failures.

Slide Number	Narrative
	3. **Slow Response**: When security issues are identified late, it results in costly recovery efforts and operational downtime. **Now, let's consider the proposed solutions**: 1. **Integrate Security Early**: We can identify vulnerabilities early by embedding security practices throughout the development lifecycle. 2. **Automate Security Processes**: Implementing tools for automated security assessments and incident responses can significantly improve our efficiency. 3. **Enhance Collaboration**: Fostering collaboration between development, operations, and security teams helps build a security-first culture. **What are the risks of waiting to switch?**: 1. **Escalating Threat Landscape**: Cyber threats continue to evolve, and delaying our response may expose us to greater risks. 2. **Increased Costs**: Addressing security vulnerabilities post-deployment can be more expensive than incorporating them from the outset. 3. **Compliance Violations**: We risk falling short of regulatory requirements, which may lead to fines or legal repercussions. **On the other hand, the benefits of acting now include**: 1. **Proactive Risk Management**: Identifying vulnerabilities early reduces the likelihood of breaches and associated costs. 2. **Enhanced Reputation**: Demonstrating our commitment to security can enhance customer trust and loyalty. 3. **Streamlined Processes**: Integrating security automation into our development lifecycle can improve efficiency and speed delivery.
15	**So, what steps should we take? My recommendation**: 1. **Conduct a Risk Assessment**: Identify and evaluate our current security gaps and their potential impacts. 2. **Prioritize Vulnerabilities**: Focus on addressing critical vulnerabilities that could have the most significant impact on our organization. 3. **Invest in Training**: Provide training for our development and operations teams on security best practices. 4. **Implement Security Automation**: Deploy automated tools for continuous security testing and monitoring. 5. **Foster Collaboration**: Create cross-functional teams to ensure security is a core consideration at every stage of development.
16	In conclusion, switching to DevSecOps is not just a security upgrade but a necessary evolution for maintaining secure, reliable software. By embedding security into every phase of the development lifecycle, we can significantly reduce risks, detect threats faster, and respond to incidents more effectively.

Slide Number	Narrative
	This proactive approach minimizes the chance of costly breaches or vulnerabilities while ensuring continuous compliance through automated security scans, audits, and testing. Although the initial transition may require investment, the long-term benefits in reduced risks and costs far outweigh the short-term efforts. DevSecOps fosters a culture of shared responsibility, where development, operations, and security teams collaborate to deliver more secure and resilient software systems.