# Green Pace logo

**Green Pace Security Policy Presentation Script**

# Nicholas Feero

August 17th, 2025  
Project Two: Security Policy Presentation | CS-405: Secure Coding  
Presentation Link: <https://youtu.be/Dg8BnAWCj3k>

**Slide 1 – Introduction**

Hello everyone, my name is Nicholas Feero and today I’ll be presenting the Green Pace Security Policy. In this presentation, I’ll cover the policies, standards, principles, and best practices that help prevent potential security vulnerabilities in both code development and systems architecture.

**Slide 2 – Overview**

The Green Pace Security Policy will ensure all staff are creating, deploying, and supporting custom software in a safe, informed manner. This policy is needed to protect Green Pace’s assets and establish regulations for security as the company expands its operations. It will be used as a framework for constructing and implementing security layers by addressing vulnerabilities. Utilizing this policy supports the best practice of Defense in Depth by addressing aspects of the perimeter security, network security, host security, endpoint security, and app security layers.

**Slide 3 – Threats Matrix**

The threats matrix consists of four categories: priority, likely, low priority, and unlikely. Priority threats indicate coding standards that have a high likelihood of occurrence and a high severity risk, which can be potentially compromising to the integrity, stability, and functionality of the code. Likely threats indicate coding standards that commonly occur and have a high severity risk but can be caught with appropriate testing and patched early in the development process. Low priority threats indicate coding standards that occur regularly but have a lower severity risk than the previous categories. Unlikely threats indicate coding standards that aren’t likely to occur due to their inclusion in code or due to the specific implementations that are required for these to be applicable. The severity of risks is generally low but can rise to a medium severity. Although there are coding standards that reside in the Unlikely category, it’s still good practice to ensure the quality and maintainability of these standards to minimize potential vulnerabilities.

Starting from priority and moving down to unlikely, the data value standard addresses sensitive information from being leaked due to invalid data values. String Correctness refers to the allocation or termination of strings to prevent buffer overflows. SQL Injection refers to sanitizing data passed into SQL or other complex subsystems to prevent tampering. Memory Protection covers the allocation and deallocation of memory, along with accessing memory, in a safe way to prevent injection or arbitrary code execution. Exceptions involve the handling of exceptions to prevent resource leaks that can cause resource exhaustion or denial-of-service vulnerabilities. Data Type addresses the use of proper data types to prevent logic errors or unexpected behavior. Assertions refer to the ineffective use or overuse of assertions that can result in hiding warnings that may be a weakness to the system. Object-Oriented Programming refers to the focus on data rather than logic, which when implemented incorrectly, can lead to unexpected behavior. Concurrency refers to a system’s ability to handle multiple tasks at once without compromising the operational integrity. Input Output refers to the communication between a system that is sending data and a system that is receiving data. If this is implemented incorrectly, disruptions to the system’s functions can occur.

**Slide 4 – 10 Principles**

The ten core security principles are listed below.

First, there is the principle of validate input data. Validating input data ensures that all data acquired through an input system (users, devices, etc.) is appropriate and safe. Part of this process involves checking that inputs conform to the type and range requirements for acceptable inputs. This prevents attacks such as malicious data injections and buffer overflow attempts. The coding standards that apply to this principle are string correctness (#3) and SQL injection (#4).

The next principle is heed compiler warnings. Compiler warnings inform the developer of risky or unintended code behavior that can lead to vulnerabilities. As compiler warnings are presented, developers should address and treat them as errors; warnings should be treated seriously and resolved promptly. Ignoring, quieting, or suppressing warnings may allow execution errors, bugs, and security issues to remain unresolved. The coding standards that apply to this principle are data type (#1), data value (#2), memory protection (#5), assertions (#6), and object oriented programming (#8).

The next principle is architect and design for security policies. Security should be integrated into the design and architecture of a system from the beginning of development, not added into it later. This principle addresses a secure structure to a system by considering issues like threats, mitigations, and compliance from a foundational necessity. Alongside this approach, this principle enforces security policies like data encryption and user authentication methods when constructing the system. I don’t have any specific coding standards that utilize this principle, but this principle is still a great principle to adhere to. It enforces strong practices that build resiliency across a code, system, and team.

The next principle is keep it simple. This principle refers to keeping the design of a system as simple and small as possible. Complex designs increase the likelihood of introducing errors in their implementation, configuration, and use. With greater complexity, more effort is needed to reach an appropriate level of security assurance. Thus, keeping the design simple leads to an easier and more maintainable system design. The coding standards that apply to this principle are exceptions (#7), concurrency (#9), and input output (#10).

The next principle is default deny. Access to resources or functions should, initially, be denied by default and specifically allowed when its implementation is necessary. Systems should not make any assumptions that users, functions, or resources have access to it. Strict permissions are needed to minimize unauthorized access and increase overall system security. I didn’t put this into any coding standard, but it is a good practice to make a habit. Utilizing the ideology of always having access to resources or functions denied can create a foundationally strong code as there won’t be any lingering resources or permissions that are accessible.

The next principle is adhere to the principle of least privilege. The principle of least privilege refers to users, systems, and processes being permitted the minimum level of access necessary to perform their expected functions. If elevated permissions are required, they should only be accessed for the least amount of time required to successfully perform the privileged task. Using this approach reduces the opportunities of malicious attacks through methods of executing unpredictable code with elevated privileges. Similarly to the principle of default deny, I also didn’t include this principle with any coding standard. Ensuring users, systems, and processes are only granted the minimum level of access necessary to perform their functions is a concept that can be applied to the foundation of security. This reduces the risk of role escalation and data breaches in any context it’s applied.

The next principle is sanitize data sent to other systems. When data is sent to other systems like databases, APIs, or external applications, it should be sanitized to prevent potentially harmful content from being sent or intercepted. This principle prevents injection attacks and ensures the systems receiving the data can do so safely to maintain intersystem security. The coding standards that apply to this principle are string correctness (#3) and SQL injection (#4).

The next principle is practice defense in depth. Practicing Defense in Depth (DiD) involves using multiple layers of security, both redundant and unique, to protect a system. This principle ensures that as many areas of security are addressed as possible, and each area of security has support if one layer fails or is breached. This practice increases the assurance across consumers that their sensitive data is safe. The coding standards that apply to this principle are SQL injection (#4), memory protection (#5), exceptions (#7), concurrency (#9), and input output (#10).

The next principle is use effective quality assurance techniques. Using effective quality assurance techniques like code reviews, unit testing, and penetration testing are good methods to identify security flaws prior to a software’s release. Performing these techniques can lead to more secure systems by ensuring the systems operate as expected under expected and malicious conditions. The coding standards that apply to this principle are data value (#2), string correctness (#3), memory protection (#5), assertions (#6), exceptions (#7), object oriented programming (#8), concurrency (#9), and input output (#10).

The final principle is adopt a secure coding standard. Adopting a secure coding standard involves examining the development language and platform a system operates in, and following the best practices, guidelines, and rules for the circumstance. Once these details have been established, it’s expected to become familiar with the specific vulnerabilities of the language and platform to avoid them. Using a secure coding standard such as CERT can outline common mistakes and vulnerabilities to promote consistency and secure practices among developers. All coding standards apply to this principle.

**Slide 5 – Coding Standards**

I’ve listed the ten coding standards in priority order, labeling which number in the priority chain each is (1 being the highest priority, and 10 being the lowest priority), and spaced the groups to match me layout of the threats matrix. The first (top) group is the priority group, the second group is the likely group, the third group is the low priority group, and the fourth (bottom) group is the unlikely group. As mentioned before in my discussion regarding the threats matrix, each coding standard is examined based on the likelihood of occurrence and the associated level of severity. When a coding standard involves vulnerabilities that have a high occurrence and a high level of severity, they need to be a priority as these will influence the code’s functionality the most. When a coding standard involves vulnerabilities that have a low occurrence and a low level of severity, they’re less likely to be considered a threat and will be ranked lower in the priority order. With that being said, all coding standards are important to address and maintain regardless of the order. Vulnerabilities can exist in many capacities, and it only takes one to compromise an entire system and all its data.

**Slide 6 – Encryption Policies**

Here are the encryption policies.

The encryption at rest policy states that {read from slideshow}.

Encryption at rest protects stored data by converting it into an unreadable format that requires a specific key to decrypt. This should be applied in practice by implementing encryption across the entirety of a storage method and storing the encryption keys in a dedicated, secure location. Encryption at rest should be used to prevent data breaches if a storage method is compromised. This prevents data from being stolen and distributed.

The encryption in flight policy states that {read from slideshow}.

Encryption in flight protects data as it moves between systems, networks, applications, or users by converting it into an unreadable format while in transmission. This should be applied in practice by configuring the necessary services to handle encrypted transmission, securing connections, and using encrypted channels with APIs and remote communication. Encryption in flight should be used to prevent attacks from intercepting or manipulating data during transmission.

The encryption in use policy states that {read from slideshow}.

Encryption in use protects data while it is being processed in memory. This should be applied in practice by encrypting data fields so sensitive data can continue being encrypted while in memory. Encryption in use should be used to reduce the risk of data breaches that involve attacks on system memory.

**Slide 7 – Triple-A Policies**

Here are the Triple-A policies.

The authentication policy states that {read from slideshow}.

Authentication is the process of confirming a user’s identity before granting access to data or a system. This should be applied in practice by enforcing the use of authentication methods like multi-factor authentication (MFA) to access sensitive data and systems, implementing secure password requirements, and managing logs to track sensitive information access. Authentication should be used to verify only expected and verified users are accessing systems with sensitive information. This reduces the risk of unauthenticated access which can result in stolen and distributed data.

The authorization policy states that {read from slideshow}.

Authorization is the process of determining what an authenticated source (user, system, etc.) is allowed to do and strictly limiting the source from performing any actions outside of its given permissions. This should be applied in practice by implementing role-based access to sensitive data and systems, performing frequent permission reviews, and examining data logs to ensure specific roles are accessing the appropriate data. Authorization should be used to prevent privilege escalation attacks and mitigate harm if a user’s account is compromised.

The accounting policy states that {read from slideshow}.

Accounting is the process of recording user and system activity for examination. This should be applied in practice by tracking user actions, system changes, and data access while monitoring logs for unusual behavior. Accounting should be used to detect unusual behavior due to a malicious attack, or to track suspicious behavior and policy violations.

**Slide 8 – Unit Testing 1 of 6**

The next six slides will focus on unit testing, how they were structured, and how to apply each test’s framework.

This slide shows the unit test for CanAddToEmptyVector, which is testing whether it’s possible to add an entry to an empty vector. This is a positive test, so an expected successful result will be that it is possible to add an entry to an empty vector. On the right side of the screen, you’ll see two prompts for this unit test. The first prompt has a prefix of RUN indicating the test is being accessed and run. The following prompt has the prefix of OK, which indicates that the test ran successfully. With a successful result, this is telling us that the test was able to assert that a collection was empty, being a size of exactly zero, then add an entry, and finally assert that the collection is no longer empty and has a size that reflects the added entry. This framework can be added to systems managing databases when creating new databases for new data sets. Adding data to an empty vector deals with memory management, and shows how data can be allotted to memory.

**Slide 9 – Unit Testing 2 of 6**

This slide shows the unit test for MaxSizeGreaterThanOrEqualToSize, which is testing whether the max size of a collection is greater than or equal to the current size of the collection. This is a positive test, so an expected successful result will be that the max size of the collection is greater than or equal to the current size of the collection. This framework can be applied in the context of buffer overflows. If the size of the collection is greater than the max size of the collection, buffer overflow could be induced, and malicious tampering can happen. Thus, it’s important to create a functional test that checks if data is within the maximum allowed quantity for the memory allotment.

**Slide 10 – Unit Testing 3 of 6**

This slide shows the unit test for AccessingClearedVectorThrows, which is testing whether an exception will be thrown from accessing an index that has been cleared. This is a negative test, so an expected successful result will be an exception being thrown if an out-of-range index is attempting to be accessed. This framework can be applied to prevent dangling pointers and accessing freed memory. These can lead to system instability, arbitrary code execution, and undefined behavior.

**Slide 11 – Unit Testing 4 of 6**

This slide shows the unit test for ClearErasesCollection, which is testing whether the collection is erased when calling the clear method. This is a positive test, so an expected successful result will be the collection being empty and of size zero once the clear method is called on collection. This framework can be applied to the concept of memory protection. Ensuring memory is managed and maintained appropriately prevents vulnerabilities like abnormal program termination, denial-of-service attacks, and arbitrary code executions.

**Slide 12 – Unit Testing 5 of 6**

This slide shows the unit test for ThrowsExceptionWhenAccessingOutOfRange, which is testing whether an exception is thrown when attempting to access an index in collection that isn’t within the size limit of the collection. This is a negative test, so an expected successful result will be an exception being thrown to indicate an out-of-range index was attempting to be reached. This can be crucial in supporting defense in depth, as out-of-range errors can result resource leaks, denial-of-service attacks, and arbitrary code execution. Ensuring there’s a solid layer of security that only processes actions within the finite size of the collection can act as an obstacle for attackers.

**Slide 13 – Unit Testing 6 of 6**

This slide shows the unit test for ResizeIncreasesCollection, which is testing whether resizing a collection will increase the size of the collection. This is a positive test, so an expected successful result will be the collection being resized to the new size defined in the resize method. This framework can be applied to support scalability within a program, so as a company grows and needs to adjust their programs accordingly, it can safely happen without compromising data or security.

**Slide 14 – Automation Summary**

This diagram is the DevSecOps diagram, which shows the pre-production and production cycles of development in the relationship of a figure eight. Typically, this process starts at the assess and plan stage in the pre-production cycle. This is where threats are assessed, and security and functional requirements are established. Next, the process moves to the design stage. This is where security is implemented early to have a proactive approach to threats. During the design stage, best practices are considered and test-driven design is a priority. Next, in the build stage, the development team utilizes trusted repositories, libraries, and resources to create a build of the program that’s secure and accounts for the security and functional requirements. This is typically where the compiler will be used to execute the code and perform run-time security checks, view compiler warnings, and discover errors. In the last stage of the pre-production cycle, the program is ready to verify and test. At this stage, the program is scanned and tested for vulnerabilities, repositories, libraries, and resources are verified, and security is tested to ensure a solid foundation for the program before transitioning to the production cycle.

Once the pre-production cycle has concluded, the lifespan of the program moves to the transition and health check stage of the production cycle. At this stage, the program is configured and deployed. During this transition, security settings and penetration testing are implemented to test the resiliency of the program. After this stage, the monitor and detect stage is reached. At this stage, the focus shifts to log collection, analytics, event alerting, and intrusion detection to ensure all access and functions are occurring as expected. Then, after there is a determination from the monitor and detect stage, the respond stage occurs. At this stage, attacks are blocked, services are turned off, and a roll back of the program happens. Once this stage has concluded, the final stage of the production cycle is maintain and stabilize. During this stage, an assessment against the security baseline happens to revisit any dated or new security measures. The objective of this stage is to return the program to a stable state after a disruption, attack, or compromise. To do this, the cycle progresses to the assess and plan stage to undergo another cycle of pre-production before releasing another stable build of the program to the production stage.

**Slide 15 – DevSecOps Tools**

Here are some tools that are utilized in the DevSecOps diagram.

Threat modeling is a tool that identifies security risks early by examining the system’s architecture and motives against the system. There is software that can assist in the threat modeling process (IriusRisk, for example), but these may only targets system architecture threats without considering human motive. This tool will be utilized in the assess and plan stage to prevent vulnerabilities prior to coding.

OWASP provides guidelines and testing tools that align with industry best practices for security. This tool can be implemented across the development process, such as the assess and plan stage, the design stage, the build stage, and the verify and test stage. Being such a flexible tool that can be used across multiple stages, the reinforces the best practices for security across the development process.

SAST tests source code for vulnerabilities without executing it. This is primarily seen in the verify and test stage, but can make an appearance in the build stage. This tool can catch incorrect code structure early in development and provide immediate feedback.

DAST tests a running application from an outside perspective, which is useful for simulating real-world attacks. This is primarily seen in the verify and test stage, however, it can utilized in the transition and health check stage to perform ongoing tests against the security of the system.

Unit testing is a tool that can automate testing small pieces of code to catch bugs in the functionality by examining expected results. This can occur in the build stage or the verify and test stage.

SIEM is a tool that collects and analyzes security events to detect and appropriately respond to threats. This is seen during the monitor and detect stage, where the focus is on examining the status of the program’s security. This tool supports incident response and monitors for anomalies, attacks, or violations to the program.

**Slide 16 – Risks and Benefits**

On this slide, I’ve included some key words that will guide my discussion on the problems, solutions, and risks or benefits of engaging in proactive or retroactive security practices

“Don’t leave security to the end” aligns with best practices in coding. This supports the side of engaging in proactive security practices rather than retroactive security practices. The ideology behind this quote is that addressing security during the early stages of development will decrease the number and severity of vulnerabilities by preparing ahead of time, rather than reacting to a breach in security later down the road. Being proactive when approaching security builds a stronger defense in depth. Engaging with security requirements early in the development process will allow for a secure foundation to build additional layers on. A problem that may present itself with proactive practices is the idea of sunk cost. Sunk cost refers to a past cost (time, resources, etc.) that can’t be recovered and has no applicability later. One can argue that spending a large amount of time early in the development process for a program that may be scrapped or needs to be expedited is counterproductive. However, when reflecting on the concept of sunk cost, developing secure methods early on even if the confidence in the program being distributed isn’t one hundred percent will build the skills needed for future development. It can create re-usable code to save on development time in the future. Thus, if there’s a benefit to future application, the sunk cost is beneficial. However, if the security practices are developed in a very specific manner that makes it unusable in future projects, the sunk cost is negative.

When examining proactive versus retroactive security practices from the perspective of an attacker, targeting a company that engages in retroactive security practices would be more appealing. A risk to retroactive security practices is that the security measures upon the release of a program may be minimal, and anyone who has malicious motive now has an easy target. Whereas with proactive security practices, an attacker’s motives would be considered early in the development process. Thus, the methods of which an attacker may attempt to compromise a company’s security will already be addressed and reinforced with specific vulnerability security measures in place.

If a company is considering proactive security practices, one consideration that needs to happen is the feasibility of the situation. Depending on the status and size of the company, it may be difficult to allot the employees and resources required to perform proactive security measures in a timely manner. With retroactive security practices, the development of the program can begin earlier due to freeing the tasks planning and designing the integration of secure methods and practices in the code. In this argument, it’s a determination of what’s more valuable: the confidence of security in your program, or the quick development of the program?

Although there are some arguments for both sides of proactive versus retroactive, the cost-benefit analysis for engaging in proactive security practices far outweighs the cost-benefit analysis of engaging in retroactive security practices in terms of less cost, and more benefits.

**Slide 17 – Recommendations**

On this slide, I’ll be providing recommendations on how to fill the current gaps in the security policy.

First, {read from slideshow}. Then, {read from slideshow}. Next, {read from slideshow}. Finally, {red from slideshow}.

**Slide 18 – Conclusions**

In conclusion: {read first bullet from slideshow}

In the Health Insurance Portability and Accountability Act (HIPAA), there exists a Security Rule that establishes a national set of security standards to protect certain health information that is maintained or transmitted in electronic form. This rule establishes administrative, physical, and technical safeguards, which provides comprehensive coverage to support the sensitivity of personal data. Under the scope of HIPAA, there exists a Breach Notification Rule, which establishes guidelines for what HIPAA-abiding companies and individuals need to do if sensitive data is compromised. This aligns with the recommendation I made involving the establishment of an incident report and response protocol. Although this example doesn’t directly relate to the security of developing a program, HIPAA is a great real-world example of how a policy can be well-defined, have clear acceptance criteria, and have appropriate response protocols.

Depending on the location and scale of operation, legal and regulatory standards need to be considered. Although a small-scale company may not have many locational standards to follow, engaging in acceptable regulatory and legal standards across numerous locations is good practice for scalability of a company. This can prepare the company to grow as needed without any expansion-related setbacks.

And finally, {read from slideshow}.

**Slide 19 – References**

Thank you for listening to my presentation, and I hope you’ve enjoyed learning about the implementation of Green Pace’s new security policy.