# CS 405 Project Two Security Policy Presentation

Danielle Eeg, February 23, 2024

Link to Presentation: <https://youtu.be/l2UowaSjYf4>

## Introduction

Welcome to the Green Pace Security Policy Presentation. I am Danielle Eeg, the developer responsible for this project. In this presentation, I will discuss the security standards, risks, and automation practices for our organization's security policy. This presentation will cover the foundational principles, identify potential threats, and discuss the measures we take to secure our systems and data comprehensively.

## Slide 2: Overview: Defense in Depth

The security policy I am about to describe is rooted in the defense-in-depth strategy, which incorporates multiple layers of security measures to protect our systems. This approach includes redundancy, failsafe mechanisms, proactive threat mitigation, and continuous improvement through security policies that cover physical, cloud, perimeter, network, host, endpoint, and app security layers. Defense in depth is critical as it creates a robust and resilient lines of defense that can withstand various attacks. This will reduce risks related to single points of failure, and ensure comprehensive protection against diverse threats.

## Slide 3: Threats Matrix

The threats matrix shown here categorizes potential threats based on their likelihood and priority. Automated threat detection methods, such as static analysis, runtime validation, fuzz testing, and unit tests, as well as other automations, can help identify and mitigate these threats. The matrix includes threats ranked according to the consideration that risk = probability \* loss (Szyk, 2024). High risk leads to high priority, so for threats that have high loss and high probability, they will be prioritized higher than low probability or low loss threats:

* Likely Threats, such as:
  + Unexpected behavior from uninitialized variables and abnormal program termination.
  + These are threats that we expect to encounter often. We will want to mitigate against them, but perhaps their impact will not be as detrimental as the risks in the priority threats category.
* Priority Threats include high likelihood, high impact threats such as:
  + Buffer overflow, SQL injection, and memory allocation errors
  + These are ranked as high priority due to their high impact on system security. These may be more or less likely than the “likely” categorized events, but based on their potential impact their risk to the system is high enough to prioritize them most highly
* Low Priority Threats include items like:
  + Variable scope errors in switches and assertions with side effects
  + These threats are relatively unlikely and have relatively low impact so they are lower in priority than some of those high-risk, high-loss threats. Of course, these are still vulnerabilities that should be addressed, but they are certainly not as urgent as some of the high impact threats
* Unlikely Threats include items like:
  + Character type ambiguity for numeric values and function paths not returning errors when expected.
  + These can cause logical or computational issues with programs that can lead to severe consequences, but they are far less likely than other threats considered here.

Automation can be a useful tool for identifying risks such as these, and it should be used continuously throughout development and product lifecycles. As threats are identified, it is best practice to prioritize their remediation in a similar fashion that determines urgency based on probability and impact.

## Slide 4: 10 Principles

A core part of creating a secure system is defining principles to guide the development process. Our security policy is guided by ten core principles:

* Validate Input Data
* Heed Compiler Warnings
* Architect and Design for Security Policies
* Keep It Simple
* Default Deny
* Adhere to the Principle of Least Privilege
* Sanitize Data Sent to Other Systems
* Practice Defense in Depth
* Use Effective Quality Assurance Techniques
* Adopt a Secure Coding Standard

As you can see, all principles except for one are supported by a secure coding standard. Future work should be done to add secure coding standards related to heeding compiler warnings.

I am about to go into detail on the coding standards that support these principles, but note that defining them is only the first step. These principles ensure that we follow best practices to protect our systems and data, by following through with coding standards, automation, encryption policies, and more to creating a secure and resilient environment.

## Slide 5: Coding Standards

Our coding standards are designed to prevent common vulnerabilities and ensure consistent security practices. When code is developed with security as a central consideration, the resultant system will be inherently safer. Additionally, secure code will result in less issues down the line in dedicated testing and validation phases of development lifecycles, which in turn will lead to reductions in timeline and human resources required to get products in the hands of customers.

In priority order, these standards are as follows:

1. STD-004-CLG for SQL injection requires all input for SQL queries to be checked for suspicious patterns or characters that could lead to a SQL Injection attack
2. STD-005-CPP is a memory protection standard that requires that all memory allocation errors should be caught and handled. This will promote system safety and stability even in the case of resource exhaustion
3. STD-003-CPP is a standard that requires memory be properly allocated for character arrays, meaning they must be sized correctly including space for the null terminator to avoid buffer overflow. Whenever possible, string data types should be used instead of char data types to avoid buffer overflow.
4. STD-002-CPP mandates initializing variables at the time of declaration to prevent undefined behavior. Uninitialized variables can contain garbage data, leading to unpredictable results and security vulnerabilities.
5. STD-009-CPP prevents accessing pointers after memory deallocation. Referencing freed memory can cause undefined behavior and security risks. Pointers should be set to nullptr after deletion, and smart pointers should be used when possible
6. STD-007-CPP mandates that all exceptions be properly handled to prevent program crashes and resource leaks. Catch-all statements must log and rethrow exceptions to avoid silently ignoring errors.
7. STD-008-CPP requires all paths in a value-returning function to return a value. Missing return values can cause runtime errors and unpredictable behavior, so every conditional branch must explicitly return a valid value.
8. STD-001-CLG requires that signed char and unsigned char be used instead of plain char when storing numeric data. This ensures consistent behavior across different platforms by eliminating ambiguity in how the data is interpreted.
9. STD-010-CLG prohibits variable declarations inside switch statements. Declaring variables inside a switch block can lead to uninitialized memory issues. Variables should be declared before the switch statement to ensure proper initialization.
10. STD-006-CLG requires assertion statements to be free of side effects, meaning they should only be used for validation and must not modify program state. This ensures predictable execution regardless of whether assertions are enabled.

Each standard includes a rule, description, severity, likelihood, remediation cost, and priority level. For instance, STD-004-CLG addresses SQL Injection with high severity, high likelihood, medium remediation cost, and high priority. The priority for each standard was determined by considering severity and likelihood similar to the threat matrix we already discussed. In this case we also considered remediation costs as a tie-braker of sorts. Consider two threats that have a similar severity and likelihood. If one will be much quicker, easier, and/or faster to resolve than the other, it is in our best interest to address that one first to get the system to a safer position quickly, so it is better protected during the implementation of the more costly or time-expensive solution. By adhering to these standards, we can create more secure and reliable software.

## Slide 6: Encryption Policies

Our encryption policies cover data at rest, data in flight, and data in use:

* For data at rest, I recommend use of AES-256 encryption to protect sensitive data in storage. This encryption method is an efficient with an attractive balance of being both fast and strong. This method of encryption is also compliant with regulations such as HIPAA and PCI-DSS (Alder, 2025) (Baykara, 2021). This encryption should be used on any sensitive data such as user identification data, healthcare data, financial data, login credentials, or any other data protected by law. Secure key management systems are recommended also to safely store keys.
* For data in flight, meaning data being transmitted over networks, I recommend TLS 1.3 protocol. This mechanism prevents interception, forgery, and tampering (Cloudflare, n.d.). Encryption of data in flight will include encryption of web application traffic, file transfers, email transmissions, database connections, API communications, VPN connections, and cloud service interactions.
* Encryption of data in use is challenging as it can be difficult to manipulate or process data properly while it is encrypted. Still protect data even when it is in use, I recommend secure enclaves to protect data by isolating the memory while processing, preventing unauthorized access. If data must be processed outside secure enclaves, techniques like homomorphic encryption or secure multiparty computation should be used (Anjuna Security Inc, 2020). This policy applies to the processing of any sensitive data like payment processing, cryptography operations, password validation, and any other data protected by regulations.

These policies ensure that sensitive data is protected at all stages, reducing the risk of data breaches and unauthorized access.

## Slide 7: Triple-A Policies

Our Triple-A policies cover authentication, authorization, and accounting:

* For authentication, I recommend multi-factor authentication (MFA), password security requirements, and secure password reset processes ensure only legitimate users can access our systems. One-time passwords should be generated by the system for functions like account verification, password resets, and multifactor authentication. These passwords should be completely random and expire after fifteen minutes. A maximum of five login attempts should be given for passwords and one-time passwords before an account is temporarily locked. In the case of a failed login attempt that results in the account being locked, the system admin and user whose account is locked should be notified by a verified mode of communication like email or sms.
* For authorization, role-based access control and the principle of least privilege limit access to necessary data and resources, with regular reviews and updates based on role changes. This will ensure that all user accounts do not have access to data beyond what is required for their role. This measure will help prevent privilege escalation attacks. When it comes to permissions for files, by default anyone who is not a document owner should by default only have read access unless write access is granted by the document author.
* For accounting, all system interactions should be logged. These logs should be retained for at least one year, or however required to meet compliance standards. This provides accountability and supports forensic investigations in case of security incidents. Alerts should be sent automatically to system administrators when abnormal logging data is detected. Logs should not be able to be changed or deleted before the retention period elapses by users of any privilege level. In the case where logging settings are changed, alerts should be sent to all security administrators. This will provide a thorough audit trail for security investigations.

These policies provide a robust framework for managing user access and ensuring accountability within our systems.

## Slide 8: Unit Testing: SQL Injection - Introduction

Unit testing is a crucial part of our security policy, helping us identify and address vulnerabilities early in the development process. These tests can automatically verify the behavior of functions or methods matches their intended purpose. This means we can check for logical accuracy, correct handling of boundary conditions and errors, and so on. These tests allow for early detection of bugs, and allow for automatic testing even when programs are revised.

We use a combination of positive and negative tests to validate input and check for unsafe patterns, ensuring the integrity of our data. I am going to show a sample of six unit tests that could be used to validate a SQL Injection checker function. This function is designed to verify user input does not contain unsafe patterns for sending forward to a SQL database, and it will throw an invalid argument exception if it does detect suspicious code.

## Slide 9: Unit Testing: SQL Injection - Positive Test 1

This test validates that all acceptable characters are not rejected by the SQL Injection Checker. It includes input to the checker that has a variety of characters such as alphanumeric, whitespace, newline, tab, period, single quote, dash, underscore, and ampersat. The expected result is that no SQL injection is detected for valid input. Notice that part of this test is first verifying that the input length is less than the maximum allowed length. This will ensure the test does not return a false negative because the input is outside the allowable range.

In future test, other forms of character input should be verified. In any given case, we are limited due to input length limitations, so it will be important to design additional tests to verify characters or combinations of characters in a similar fashion.

## Slide 10: Unit Testing: SQL Injection - Positive Test 2

This test verifies that the maximum string length is accepted as a length boundary case. The input is constructed to match the maximum allowable length, and then it is verified to ensure it is the maximum length. The expected result is that no SQL injection is detected, confirming that the system handles boundary conditions correctly.

For future improvements, minimum string sizes can be tested, as could string lengths between the minimum and maximum lengths.

## Slide 11: Unit Testing: SQL Injection - Negative Test 1

Here is our first negative test, where we will expect to see an exception thrown. This test checks that inputs longer than the maximum string length are rejected. The input exceeds the maximum allowable length, and the expected result is that an invalid argument exception is thrown during input validation, ensuring that the system rejects unsafe input lengths.

## Slide 12: Unit Testing: SQL Injection - Negative Test 2

This test verifies the rejection of the common pattern 'OR value=value', which can create an "always true" condition. This is a common injection pattern for bypassing authentication logic. The input includes this pattern, and the expected result is that an exception is thrown, preventing bypassing of logic that requires authentication.

For future improvements, other forms of this test should be tried outside of the one equals one pattern, for example string values could be used instead of integers to perform the same attack.

## Slide 13: Unit Testing: SQL Injection - Negative Test 3

This test checks for SQL statements using comments to disregard part of a query. This type of attack can comment out the remainder of the query and drop critical logic that could bypass critical authentication steps. For this test, the input includes a line comment, and the expected result is that an exception is thrown, ensuring the system rejects comment injection attempts.

For future improvements, inline comments could be trialed as they can be used for similar attack mechanisms.

## Slide 14: Unit Testing: SQL Injection - Negative Test 4

This test verifies the rejection of UNION injection, where data is returned from another table. For example, the statement tested here could pull password data from another table without the proper authorization. The input includes a UNION statement, and the expected result is that an exception is thrown, preventing unauthorized data access.

For future improvements, this test should be modified based on the SQL table structure to provide a more customized testing method, and other boundary tests should be added as well

## Slide 15: Unit Testing: SQL Injection – Other considerations

Beyond the test-specific recommendations I made for SQL Injection detection using unit tests, other tests should be added for testing other injection patterns and boundary cases. The ones provided here are just an example of how these tests could be designed.

Injection detection should not be the only layer of defense against injection attacks. Other layers of protection like parameterization of queries can be used to prevent user input from being read as code.

Finally, it will be important to consider how this is implemented to ensure proper functionality for the system. After all some of these patterns can be useful for legitimate functions, such as UNION statements or commenting. It will be important for functionality that some of these validations are only applied to user input and not the full final query.

## Slide 16: Automation Summary

Automation is a key component of our security policy, enabling us to enforce standards and detect vulnerabilities throughout the development process. By integrating security automation into every step of the development process, we can create a far more inherently safe system that does not wait until final testing stages or a singular dedicated security phase to address and revise to improve security.

## Slide 17: Automation Tools

We use various tools at different stages of the DevSecOps pipeline to make this process as seamlessly integrated into the already defined DevOps pipeline as possible.

* During the assess and plan phase:
  + We can automate the evaluation of the threat landscape using tools like ThreatModeler and determine how we can create a system that meets compliance requirements. Using compliance automation tools like Vanta and Drata
* During design
  + We can automate the development of a security roadmap template using tools like OWASP’s Software Assurance Maturity Model.
* During the Build phase,
  + Not only will we be able to build a more secure system by implementing the secure coding standards policy, but we can also automatically check for vulnerabilities in dependencies, in the built code, and in other security settings using tools like Snyk and Cppcheck
* During the verify and testing phase
  + We can test for vulnerabilities using automation like fuzz testing, which will throw randomized inputs at the system to see how it responds using a tool like beSTORM, and also take a more systemized approach to penetration testing using a runtime application self-protection testing tool like ThreatModeler
* During the Transition and Health Check phase,
  + We can automate the provisioning of the security infrastructure, such as firewall configuration, using tools like Terraform
* During the Monitor and Detection phase
  + We can log and analyze system and user behavior in real time to allow for anomaly detection using security information and event management tools like Splunk by Cisco
* During the Respond phase
  + We can use security orchestration, automation, and response operations to automate responses to threats, which can reduce the impacts of attacks by reducing response time once they are detected. This can also be done using Splunk, or IBM Resilient.
* In the maintain and stabilize phase,
  + We can automate the return to baseline security standards, also using Splunk. This kind of automation helps to prevent baseline creep over time after events.

Automation helps us create consistent and repeatable security practices, reducing human error and improving response times to security incidents.

## Slide 18: Risks and Benefits

Our current strategy faces several risks, including the lack of continuous monitoring for evolving threats, gaps in secure coding practices, delays in vulnerability resolution, and manual system reviews that can leave undetected vulnerabilities.

Acting now to address these risks allows us to reduce urgent security vulnerabilities, ensure compliance, and proactively manage security risks. However this will require significant budget and resource allocation, and leave less time for thorough planning. Acting later provides more time for budgeting, HR planning, and strategy planning but will also increases the risk of security breaches, fines, and higher long-term costs.

## Slide 19: Strategy for Immediate Implementation

To address the identified risks, we propose the following immediate actions:

* Adopt automated security testing and monitoring.
* Implement role-based access control with annual reviews.
* Begin logging and real-time anomaly detection and response.
* Roll out a full secure coding standard policy for all development team members.
* Define and regularly update an emergency incident response plan.
* Require mandatory security training for all employees based on their roles.

The current strategy has several deficits I would like to note These include:

* A lack of focus on training for security awareness of social engineering, phishing and other user-caused threats.
* Another weakness is a lack in focus on physical security in this proposed policy. Future improvements could include more in-depth gap analysis with recommended improvements for physical security and implementation of zero trust network architecture.
* Another gap is additional secure coding standards. While ten have been defined, future works should be done to develop out a more comprehensive list of coding practices to mitigate against risks.
* Finally, the development of a roadmap that has time-bound benchmarks for security improvements will be highly beneficial for staying on target with implementing a full defense in depth strategy
* Our strategy focuses on creating a proactive and resilient security posture, addressing both immediate and long-term security needs.

## Slide 20: Recommendations

To enhance our security policy, we recommend the following actions. As evidence of the importance of implementing these recommendations, consider the 2019 case study from the Capital One data breach:

* The lack of multifactor authentication made it much simpler for the attacker to penetrate the system for the Capital One data breach. Enforce multifactor authentication and require secure passwords to prevent unauthorized access.
* It was a misconfigured firewall and loose access control settings that made it so easy for the attacker to access the 106 million accounts for the breach. It is critical to implement strict role-based access control based on the principle of least privilege and regular security audits to review settings of security infrastructure and permission management.
* After the breach occurred, it took months for Capital One to discover what had happened. To prevent a similar delay in response, adopt logging policies with real-time analytics and response plans to detect and respond to security incidents promptly.
* The attacker was able to complete decryption of encrypted data that was stolen. It will be important that the security policy has strong protocols for data at rest, in use, and in flight, ensuring compliance with relevant standards.
* Finally, use automation throughout the DevSecOps cycle to ensure thorough vulnerability scanning and secure coding practices. This will help prevent vulnerabilities like the misconfigured setting that enabled the breach at Capital One.

By implementing these recommendations, we can strengthen our security posture and mitigate the risks identified in our current strategy.

## Slide 21: Conclusion

In conclusion, our security policy is designed to protect Green Pace's systems and data through a combination of best practices related to automation, encryption, authentication, authorization, accounting, secure coding, employee training, and regular reviews of policies. By following the principles of defense in depth, we create a robust security posture that can withstand various types of threats. Our strategy for immediate implementation and recommended enhancements will further strengthen our security measures, ensuring the safety and integrity of our systems and data

Slide 22: References

Alder, S. (2025, January 9). *HIPAA Encryption Requirements*. Retrieved from The HIPAA Journal: https://www.hipaajournal.com/hipaa-encryption-requirements/

Anjuna Security Inc. (2020). *Secure Enclaves: The Powerful Way to Make Data Secure by Default*. Retrieved from Anjuna: https://cdn.prod.website-files.com/63f08d2744acc65d1d4d6d85/6405c9d5d2eb32d4786d1891\_case-study-what-is-a-secure-enclave.pdf

Baykara, S. (2021, July 2021). *What Are the PCI DSS Encryption Requirements*. Retrieved from PCI DSS Guide: https://pcidssguide.com/what-are-the-pci-dss-encryption-requirements/

Cloudflare. (n.d.). *What is TLS (Transport Layer Security)?* Retrieved from Cloudflare: https://www.cloudflare.com/learning/ssl/transport-layer-security-tls/

Graham, K. (n.d.). *Why MFA is Crucial for Business Security*. Retrieved from Forthright: https://blog.forthright.com/tech-insights/why-mfa-is-crucial

Neto, N. N., Madnick, S., Paula, A. M., & Borges, N. M. (2020). *A Case Study of the Capital One Data Breach.* Cambridge: Massachusetts Institute of Technology .

Novoseltseva, E. (2019, August 30). *8 Benefits of Unit Testing*. Retrieved from DZone: https://dzone.com/articles/top-8-benefits-of-unit-testing

*SQL injection cheat sheet*. (n.d.). Retrieved from inviciti: https://www.invicti.com/blog/web-security/sql-injection-cheat-sheet/

*SQL injection UNION attacks*. (n.d.). Retrieved from PortSwigger: https://portswigger.net/web-security/sql-injection/union-attacks

Szyk, B. (2024, May 24). *Risk Calculator*. Retrieved from Omni Calculator: https://www.omnicalculator.com/statistics/risk