**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | "Validating Input Data" is a crucial principle in the realm of security, emphasizing the significance of thoroughly scrutinizing and sanitizing any data received by a system or application. This principle plays a pivotal role in safeguarding the integrity and functionality of the system against potentially malicious or unexpected input. By validating input data, one can fortify defenses against the likes of injection attacks, buffer overflows, and cross-site scripting, among others. This process involves meticulously verifying the type, format, length, and range of input data, ensuring it aligns with the expected criteria before further processing or storage. Employing this principle enables us to proactively thwart a plethora of security threats while cultivating a sense of reliability and trustworthiness throughout the system. (Seacord, 2018) |
| 1. Heed Compiler Warnings | "Heed Compiler Warnings" is a vital principle of security that emphasizes the significance of paying close attention to warnings generated by the compiler during the code compilation process. Compilers analyze the code and provide warnings to alert developers about potential issues or vulnerabilities. By heeding these warnings and taking appropriate actions, developers can proactively identify and address possible security weaknesses in the code. Ignoring compiler warnings may result in the introduction of vulnerabilities or bugs that could be exploited by attackers. Therefore, following the principle of heeding compiler warnings helps ensure the overall security and reliability of the software or system being developed. (Seacord, 2018) |
| 1. Architect and Design for Security Policies | "Architect and Design for Security Policies" is a fundamental principle of security that emphasizes the importance of considering security requirements from the early stages of system architecture and design. By incorporating security policies into the design process, organizations can ensure that security controls, mechanisms, and safeguards are implemented effectively to protect the system and its assets. This principle involves identifying potential threats, assessing risks, and defining security objectives and policies. It also entails designing secure authentication mechanisms, access controls, encryption protocols, and other security features to align with the defined policies. By adhering to this principle, organizations can build robust and resilient systems that are capable of withstanding and mitigating various security threats and challenges. (Seacord, 2018) |
| 1. Keep It Simple | "Keep It Simple" is a crucial principle of security that emphasizes the importance of simplicity in designing and implementing security measures. It suggests that security systems, processes, and controls should be kept as straightforward and uncomplicated as possible. Complexity often introduces additional risks and vulnerabilities, as it becomes harder to understand, manage, and maintain. By embracing simplicity, security measures can be more transparent and easier to audit, reducing the likelihood of errors or oversights. It also promotes usability and user acceptance, as overly complex security measures can hinder productivity and frustrate users. Striving for simplicity in security helps strike a balance between effectiveness and practicality, ensuring that security measures are both robust and manageable. (Seacord, 2018) |
| 1. Default Deny | "Default Deny" is a critical principle of security that advocates for a proactive approach to access control and permissions. In this principle, the default stance for granting access or permissions is set to "deny" by default. This means that unless explicitly authorized, all access attempts or requests are automatically denied. By implementing a default deny policy, organizations can minimize the risk of unauthorized access, reduce the attack surface, and enhance overall security. It ensures that only approved and validated entities or processes are granted access to resources, systems, or sensitive information. Default deny is often implemented through robust access control mechanisms, such as firewalls, permission settings, or role-based access control (RBAC) systems. This principle forms a fundamental part of a defense-in-depth strategy, reinforcing the security posture and protecting against potential breaches or unauthorized activities. (Seacord, 2018) |
| 1. Adhere to the Principle of Least Privilege | "Adhere to the Principle of Least Privilege" is a critical principle of security that emphasizes restricting access rights and permissions to the minimum level necessary for users or entities to perform their tasks or functions. According to this principle, users should only be granted the privileges required to fulfill their specific roles and responsibilities, and no more. By limiting privileges, organizations can minimize the potential damage caused by compromised or malicious accounts. This principle reduces the attack surface and mitigates the risk of unauthorized access, accidental misuse, or intentional abuse of sensitive resources. Implementing the principle of least privilege involves conducting a thorough analysis of user roles, clearly defining access permissions, and regularly reviewing and adjusting privileges based on changing needs. By strictly adhering to this principle, organizations can enhance the security posture, maintain data confidentiality, and minimize the impact of security incidents. (Seacord, 2018) |
| 1. Sanitize Data Sent to Other Systems | "Sanitize Data Sent to Other Systems" is an essential principle of security that emphasizes the need to cleanse and validate data before transmitting it to external systems or entities. This principle aims to prevent the inadvertent disclosure of sensitive information or the introduction of malicious data into other systems. By sanitizing data, organizations remove or neutralize potentially harmful elements, such as special characters, escape sequences, or code snippets that could exploit vulnerabilities in the receiving system. Sanitization techniques may include input validation, data filtering, or encoding to ensure that only safe and trusted data is shared with external systems. This principle helps to safeguard against data breaches, injection attacks, or data corruption, maintaining the integrity and confidentiality of information throughout its journey across different systems. (Seacord, 2018) |
| 1. Practice Defense in Depth | "Practice Defense in Depth" is a vital principle of security that advocates for the implementation of multiple layers of security controls and measures to protect systems and data. It emphasizes that relying on a single security measure is not sufficient to defend against sophisticated and evolving threats. Instead, a multi-layered approach should be adopted to create overlapping layers of defense. This principle involves combining various security technologies, policies, procedures, and practices to create a robust security framework. Examples of defense-in-depth measures include firewalls, intrusion detection systems, encryption, access controls, strong authentication mechanisms, regular security assessments, and user awareness training. By implementing defense in depth, organizations can significantly enhance their ability to detect, prevent, and respond to security incidents, minimizing the impact of potential breaches and ensuring the overall resilience of their systems. (Seacord, 2018) |
| 1. Use Effective Quality Assurance Techniques | "Use Effective Quality Assurance Techniques" is a crucial principle of security that emphasizes the importance of thorough testing and quality assurance processes to ensure the reliability and security of systems and applications. This principle recognizes that vulnerabilities and weaknesses can be introduced during the development lifecycle. By employing effective quality assurance techniques, such as code reviews, penetration testing, vulnerability scanning, and security testing, organizations can identify and address potential security flaws before deployment. Quality assurance techniques help validate the effectiveness of implemented security controls, assess the resilience of the system against attacks, and ensure compliance with security standards and best practices. By incorporating these techniques into the development process, organizations can enhance the overall security posture of their systems and mitigate the risk of security breaches or unauthorized access to sensitive information. (Seacord, 2018) |
| 1. Adopt a Secure Coding Standard | "Adopting a Secure Coding Standard" is a key security principle that emphasizes the importance of following established guidelines for writing secure code. By adhering to these standards, developers can prevent common programming errors and vulnerabilities, reducing the risk of security breaches. Secure coding standards cover areas such as input validation, data handling, and secure communication, promoting consistency and enhancing the overall security of software applications. (Seacord, 2018) |

Resources

Seacord, R. (2018). *Top 10 secure coding practices*. Top 10 Secure Coding Practices - CERT Secure Coding - Confluence. <https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices>

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Name: Data Type Coding Standard  Rationalize the Standard: This standard ensures consistent and proper usage of data types in the code, promoting code clarity, maintainability, and minimizing potential bugs or vulnerabilities associated with incorrect data type usage. |

| **Noncompliant Code** |
| --- |
| This code assigns a floating-point value to an integer variable, which violates the data type coding standard. |
| int count = 5.7; |

| **Compliant Code** |
| --- |
| This code assigns an integer value to an integer variable, adhering to the data type coding standard. |
| int count = 5; |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** Adopt a Secure Coding Standard:  The principle of "Adopt a Secure Coding Standard" directly maps to the "Data Type Coding Standard" (STD-001-CPP) mentioned in the provided code example. By adopting this coding standard, developers ensure consistent and proper usage of data types throughout the codebase. In the noncompliant code example, assigning a floating-point value to an integer variable violates the data type coding standard, which could lead to unexpected behavior or vulnerabilities. However, in the compliant code example, the proper data type (integer) is used for the assignment, adhering to the coding standard and promoting code clarity and maintainability. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Moderate | Medium | Low | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.11 | Cppcheck Data Type Analysis | Cppcheck's Data Type Analysis enforces the "Data Type" coding standard by automatically detecting potential issues with data type usage in C and C++ applications. Integrating this tool enhances code clarity, maintainability, and minimizes potential bugs or vulnerabilities associated with incorrect data types. |
| Clang Static Analyzer | 10.8 | Clang Static Analyzer Data Type Checks | Clang Static Analyzer's Data Type Checks enforce the "Data Type" coding standard by automatically detecting potential issues with data type usage in C and C++ applications. Integrating this tool enhances code clarity, maintainability, and minimizes potential bugs or vulnerabilities associated with incorrect data types. |
| CppDepend | 2020.1.0 | CppDepend Data Type Rules | CppDepend's Data Type Rules enforce the "Data Type" coding standard by automatically detecting potential issues with data type usage in C++ codebases. Integrating this tool enhances code clarity, maintainability, and minimizes potential bugs or vulnerabilities associated with incorrect data types. |
| Microsoft C++ Core Check (CppCoreCheck) | 16.3 | CppCoreCheck Data Type Analysis | CppCoreCheck's Data Type Analysis enforces the "Data Type" coding standard by automatically detecting potential issues with data type usage in C++ codebases. Integrating this tool enhances code clarity, maintainability, and minimizes potential bugs or vulnerabilities associated with incorrect data types. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Name: Data Value Coding Standard  Rationalize the Standard: This standard ensures the proper initialization and usage of data values, promoting code reliability, preventing undefined behavior, and reducing the risk of bugs or vulnerabilities associated with uninitialized or improperly assigned values. |

| **Noncompliant Code** |
| --- |
| This code declares an integer variable without initializing it, resulting in an undefined value that may lead to unpredictable behavior. |
| int quantity; // Variable declared but not initialized |

| **Compliant Code** |
| --- |
| This code initializes the integer variable with a default value, adhering to the data value coding standard. |
| int quantity = 0; // Variable properly initialized with a default value |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  Keep It Simple  Explanation: The "Data Type Coding Standard" (STD-001-CPP) aligns with the principle of "Keep It Simple" in software development. By adhering to this standard, developers ensure consistent and straightforward usage of data types, promoting code clarity and maintainability. In the compliant code example provided, assigning an integer value to an integer variable follows the data type coding standard, making the code easy to understand and reducing the risk of potential bugs or vulnerabilities.  Use Effective Quality Assurance Techniques  Explanation: The "Data Type Coding Standard" supports the principle of "Use Effective Quality Assurance Techniques" by providing a specific guideline for data type usage in the code. Following this standard aids in code quality assurance by ensuring that data types are correctly applied, minimizing the chances of defects and security vulnerabilities resulting from data type errors. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | High | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.11 | Data Value Analysis | Cppcheck's Data Value Analysis enforces the "Data Value" coding standard by automatically detecting potential issues with data values in C and C++ applications. Integrating this tool enhances code accuracy and reliability, minimizing potential bugs or vulnerabilities associated with incorrect data values. |
| Clang Static Analyzer | 10.8 | Clang Static Analyzer Data Value Checks | S Clang Static Analyzer's Data Value Checks enforce the "Data Value" coding standard by automatically detecting potential issues with data values in C and C++ applications. Integrating this tool enhances code accuracy and reliability, minimizing potential bugs or vulnerabilities associated with incorrect data values. |
| CppDepend | 2020.1.0 | CppDepend Data Value Rules | CppDepend's Data Value Rules enforce the "Data Value" coding standard by automatically detecting potential issues with data values in C++ codebases. Integrating this tool enhances code accuracy and reliability, minimizing potential bugs or vulnerabilities associated with incorrect data values. |
| Microsoft C++ Core Check (CppCoreCheck) | 16.3 | CppCoreCheck Data Value Analysis | CppCoreCheck's Data Value Analysis enforces the "Data Value" coding standard by automatically detecting potential issues with data values in C++ codebases. Integrating this tool enhances code accuracy and reliability, minimizing potential bugs or vulnerabilities associated with incorrect data values. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Name: String Correctness Coding Standard  Rationalize the Standard: This standard promotes correct and secure handling of string data to prevent common issues such as buffer overflows, format string vulnerabilities, and improper memory management when working with strings. |

| **Noncompliant Code** |
| --- |
| This code copies a string that exceeds the buffer size, leading to a buffer overflow vulnerability. |
| char buffer[10];  strcpy(buffer, "This is a long string that exceeds the buffer size"); |

| **Compliant Code** |
| --- |
| This code uses strncpy to copy a string while ensuring that it does not exceed the buffer size. It also manually adds the null termination character to the end of the buffer. |
| char buffer[20];  strncpy(buffer, "This is a safe string", sizeof(buffer));  buffer[sizeof(buffer) - 1] = '\0'; // Ensure null termination |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  Principle: Validate Input Data    Explanation: The "String Correctness" coding standard emphasizes the principle of "Validate Input Data." By adhering to this standard, developers ensure that all input data, including strings, is thoroughly validated and sanitized before processing. Proper input validation helps prevent common security threats like buffer overflows, SQL injection, and cross-site scripting (XSS) attacks, which could exploit vulnerabilities in string handling. Validating and sanitizing input data ensures that only expected and safe string values are processed, reducing the risk of malicious inputs compromising the system's integrity and security.    Principle: Use Effective Quality Assurance Techniques    Explanation: The "String Correctness" coding standard supports the principle of "Use Effective Quality Assurance Techniques" by providing specific guidelines for handling strings in the code. Following this standard aids in code quality assurance by ensuring that strings are correctly manipulated, preventing errors and security vulnerabilities related to improper string handling. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Moderate | Medium | Low | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.11 | Cppcheck String Correctness Analysis | Cppcheck's String Correctness Analysis enforces the "String Correctness" coding standard by automatically detecting potential issues with C-style string manipulation in C and C++ applications. Integrating this tool enhances string safety and robustness, minimizing potential bugs or security risks associated with incorrect string handling. |
| Clang Static Analyzer | 10.8 | Clang Static Analyzer String Correctness Checks | Clang Static Analyzer's String Correctness Checks enforce the "String Correctness" coding standard by automatically detecting potential issues with C-style string manipulation in C and C++ applications. Integrating this tool enhances string safety and robustness, minimizing potential bugs or security risks associated with incorrect string handling. |
| CppDepend | 2020.1.0 | CppDepend String Correctness Rules | CppDepend's String Correctness Rules enforce the "String Correctness" coding standard by automatically detecting potential issues with C-style string manipulation in C++ codebases. Integrating this tool enhances string safety and robustness, minimizing potential bugs or security risks associated with incorrect string handling. |
| PVS-Studio | 7.22 | PVS-Studio String Correctness Analysis | PVS-Studio's String Correctness Analysis enforces the "String Correctness" coding standard by automatically detecting potential issues with C-style string manipulation in C and C++ applications. Integrating this tool enhances string safety and robustness, minimizing potential bugs or security risks associated with incorrect string handling. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Name of Standard: SQL Injection Coding Standard  Rationalize the Standard: This standard focuses on preventing SQL injection vulnerabilities by promoting the use of prepared statements or parameterized queries, proper input validation, and sanitization techniques when working with SQL queries. |

| **Noncompliant Code** |
| --- |
| This code directly concatenates user input into the SQL query, making it susceptible to SQL injection attacks. |
| std::string username = getUserInput();  std::string query = "SELECT \* FROM users WHERE username = '" + username + "'";  executeSQLQuery(query); |

| **Compliant Code** |
| --- |
| This code uses a prepared statement or parameterized query with a placeholder for the user input, ensuring proper separation of the query structure and data values. |
| std::string username = getUserInput();  std::string query = "SELECT \* FROM users WHERE username = ?";  prepareAndExecuteSQLQuery(query, username); |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  Principle: Validate Input Data    Explanation: The "SQL Injection" coding standard aligns with the principle of "Validate Input Data." By adhering to this standard, developers ensure that all input data, especially data used in SQL queries, is thoroughly validated and sanitized to prevent SQL injection attacks. Proper input validation helps to eliminate malicious code or characters that attackers might use to manipulate the SQL query, thus reducing the risk of unauthorized access, data breaches, or data corruption.    Principle: Use Effective Quality Assurance Techniques    Explanation: The "SQL Injection" coding standard supports the principle of "Use Effective Quality Assurance Techniques" by providing specific guidelines for secure handling of SQL queries. Following this standard aids in code quality assurance by ensuring that SQL queries are constructed properly, minimizing the potential for SQL injection vulnerabilities. Implementing thorough testing, including security-focused testing techniques, can help identify and mitigate potential SQL injection risks during development. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Medium | High | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify Static Code Analyzer (SCA) | 22.2.1 | Fortify SQL Injection Analysis | Fortify SCA is a powerful static code analysis tool that includes a SQL Injection Analysis feature to enforce the "SQL Injection" coding standard. This analysis automatically detects potential vulnerabilities in SQL queries that could lead to SQL injection attacks. By utilizing Fortify SCA's SQL Injection Analysis, developers can ensure compliance with the coding standard and proactively address security risks related to SQL injection. Integrating Fortify SCA into the development process enhances code security and helps prevent potential SQL injection vulnerabilities in C and C++ applications. |
| Veracode Static Analysis | 0.5.0 | Veracode SQL Injection Analysis | Veracode Static Analysis is a comprehensive code analysis tool that includes a dedicated SQL Injection Analysis to enforce the "SQL Injection" coding standard. This analysis automatically detects potential vulnerabilities in SQL queries that could lead to SQL injection attacks. By utilizing Veracode's SQL Injection Analysis, developers can ensure compliance with the coding standard and proactively address security risks related to SQL injection. Integrating Veracode Static Analysis into the development process enhances code security and helps prevent potential SQL injection vulnerabilities in C and C++ applications. |
| Checkmarx | 9.5.0 | Checkmarx SQL Injection Analysis | Checkmarx is a powerful static code analysis tool that includes a dedicated SQL Injection Analysis to enforce the "SQL Injection" coding standard. This analysis automatically detects potential vulnerabilities in SQL queries that could lead to SQL injection attacks. By utilizing Checkmarx's SQL Injection Analysis, developers can ensure compliance with the coding standard and proactively address security risks related to SQL injection. Integrating Checkmarx into the development process enhances code security and helps prevent potential SQL injection vulnerabilities in C and C++ applications. |
| OWASP ZAP (Zed Attack Proxy) | 2.13.0 | OWASP ZAP SQL Injection Scanner | OWASP ZAP is a popular open-source web application security scanner that includes a dedicated SQL Injection Scanner to enforce the "SQL Injection" coding standard. This scanner automatically detects potential vulnerabilities in SQL queries that could lead to SQL injection attacks. By utilizing OWASP ZAP's SQL Injection Scanner, developers can ensure compliance with the coding standard and proactively address security risks related to SQL injection. Integrating OWASP ZAP into the development process enhances code security and helps prevent potential SQL injection vulnerabilities in C and C++ applications. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Name of Standard: Memory Protection Coding Standard  Rationalize the Standard: This standard focuses on ensuring proper memory management and protection to prevent memory-related vulnerabilities such as buffer overflows, use-after-free, and memory leaks. |

| **Noncompliant Code** |
| --- |
| Noncompliant Code Description: This code performs a strcpy operation without validating the size of the buffer, potentially leading to a buffer overflow vulnerability. |
| char buffer[100];  strcpy(buffer, "This is a long string that exceeds the buffer size"); |

| **Compliant Code** |
| --- |
| Compliant Code Description: This code dynamically allocates memory for the buffer based on the length of the string, ensuring sufficient space to hold the string and preventing buffer overflow vulnerabilities. |
| std::string str = "This is a safe string";  size\_t bufferSize = str.length() + 1;  char\* buffer = new char[bufferSize];  strcpy\_s(buffer, bufferSize, str.c\_str()); |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** [Name the principle and explain how it maps to this standard.]  Principle: Keep It Simple    Explanation: The "Memory Protection" coding standard aligns with the principle of "Keep It Simple." By adhering to this standard, developers ensure that memory protection mechanisms are implemented in a straightforward and easy-to-understand manner. The goal is to minimize complex memory manipulation, which can introduce vulnerabilities and obscure potential security risks. By keeping memory protection simple and clear, developers enhance code maintainability and reduce the likelihood of memory-related security issues such as buffer overflows and memory leaks.    Principle: Architect and Design for Security Policies    Explanation: The "Memory Protection" coding standard supports the principle of "Architect and Design for Security Policies." This standard focuses on planning and designing the software architecture with robust memory protection mechanisms in mind. By defining clear memory access policies, developers can prevent unauthorized access or modification of memory areas, reducing the risk of security breaches and enhancing the overall security posture of the software. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Medium | High | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| AddressSanitizer | 4.8 | AddressSanitizer Memory Protection Analysis | AddressSanitizer's Memory Protection Analysis enforces the "Memory Protection" coding standard by automatically detecting potential memory-related vulnerabilities during program execution. Integrating this tool enhances code reliability and helps prevent memory-related vulnerabilities in C and C++ applications. |
| Valgrind | 3.21.0 | Valgrind Memory Protection Analysis | Valgrind is a powerful dynamic analysis tool that includes a Memory Checker designed to detect memory-related errors in C and C++ applications. By utilizing Valgrind's Memory Checker, developers can automatically identify memory leaks, invalid memory accesses, and other memory-related issues, aligning with the "Memory Protection" coding standard. Valgrind provides detailed information about memory errors, helping developers fix potential vulnerabilities and ensure memory protection. Integrating Valgrind into the testing and debugging process enhances the application's stability and security, reducing the risk of memory-related exploits and crashes. |
| Clang Static Analyzer | 10.8 | Clang Static Analyzer Memory Safety Checks | Clang Static Analyzer includes Memory Safety Checks that automatically detect memory-related errors in C and C++ applications. By leveraging these checks, developers ensure compliance with the "Memory Protection" coding standard, proactively addressing potential vulnerabilities and enhancing code reliability. Integrating Clang Static Analyzer aids in maintaining memory protection and promoting secure coding practices in C and C++ applications. |
| Microsoft Visual Studio / Visual C++ | 12.0 | Visual C++ Security Features | Microsoft Visual Studio / Visual C++ includes built-in security features that automatically detect and prevent memory-related vulnerabilities in C and C++ applications, aligning with the "Memory Protection" coding standard. These features include buffer overrun detection, address space layout randomization (ASLR), and stack cookie protection. By utilizing Visual C++ security features, developers can enhance the application's memory protection, proactively preventing potential security exploits and crashes caused by memory-related issues. Integrating Visual C++ security features into the development process helps ensure code reliability and security, mitigating the risk of memory-related vulnerabilities. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Name of Standard: Assertions Coding Standard  Rationalize the Standard: This standard focuses on the proper use of assertions in code to validate assumptions, catch unexpected conditions, and aid in debugging and development. |

| **Noncompliant Code** |
| --- |
| Noncompliant Code Description: This code includes an assertion that will always fail since num is not greater than 10. It can lead to unexpected behavior and may hinder proper debugging. |
| int num = 0; assert(num > 10); |

| **Compliant Code** |
| --- |
| Compliant Code Description: This code includes an assertion that is true since num is greater than 10, helping to validate assumptions and catch unexpected conditions. |
| int num = 15;  assert(num > 10); |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** [Name the principle and explain how it maps to this standard.]  Principle: Use Effective Quality Assurance Techniques    Explanation: The "Assertions" coding standard aligns with the principle of "Use Effective Quality Assurance Techniques." By following this standard, developers incorporate assertions in the codebase to validate assumptions and check for expected conditions during program execution. Assertions act as powerful debugging and testing tools, helping to identify and address potential issues early in the development process. When an assertion fails, it indicates the presence of a critical problem that requires immediate attention, streamlining the debugging process and improving code quality assurance.    Principle: Keep It Simple    Explanation: The "Assertions" coding standard also supports the principle of "Keep It Simple." Developers are encouraged to use concise and straightforward assertions that focus on specific checks and conditions. By keeping assertions simple and easy to understand, developers enhance the readability of the code and reduce the likelihood of introducing errors during implementation or maintenance. Additionally, simple assertions can contribute to better code review practices, facilitating the identification of potential security issues or logic flaws. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Low | Low | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppUTest | 3.8 | CppUTest Assertion Macros | CppUTest automates validation of expected results in C/C++ unit tests, ensuring compliance with the "Assertions" coding standard. Its concise and user-friendly syntax helps identify code discrepancies and logic errors, enhancing code reliability and stability. |
| Catch2 | 3.4.0 | Catch2 Assertion Macros | Catch2 automates validation of expected results in C++ unit tests, ensuring compliance with the "Assertions" coding standard. Using Catch2's assertions, developers identify code discrepancies and logic errors, enhancing C++ code reliability and stability. |
| Google Test (GTest) | 1.13.0 | GTest Assertion Macros | GTest's Assertion Macros automate validation of expected results in C++ unit tests, ensuring compliance with the "Assertions" coding standard. By using GTest's assertions, developers identify code discrepancies and logic errors, enhancing C++ code reliability and stability. |
| Boost.Test | 1.83.0 | Boost.Test Assertion Macros | Boost.Test's Assertion Macros automate validation of expected results in C++ unit tests, ensuring compliance with the "Assertions" coding standard. By using Boost.Test's assertions, developers identify code discrepancies and logic errors, enhancing C++ code reliability and stability. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Name of Standard: Exceptions Coding Standard  Rationalize the Standard: This standard focuses on the proper use of exceptions in code to handle exceptional or error conditions, improving code robustness, maintainability, and error handling capabilities. |

| **Noncompliant Code** |
| --- |
| Noncompliant Code Description: This code throws a string as an exception, which lacks proper exception handling and type safety. |
| void divide(int a, int b) {  if (b == 0) {  throw "Divide by zero error";  }  int result = a / b;  // ...  } |

| **Compliant Code** |
| --- |
| Compliant Code Description: This code throws a std::runtime\_error exception, which provides a standardized way to handle and propagate exceptions, improving error handling and providing better type safety. |
| void divide(int a, int b) {  if (b == 0) {  throw std::runtime\_error("Divide by zero error");  }  int result = a / b;  // ...  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  Principle: Use Effective Quality Assurance Techniques    Explanation: The "Exceptions" coding standard aligns with the principle of "Use Effective Quality Assurance Techniques." By following this standard, developers incorporate exception handling mechanisms in the codebase to effectively manage and respond to exceptional situations during program execution. Exception handling aids in graceful error recovery and prevents unexpected crashes, enhancing the overall reliability and robustness of the software. Properly handled exceptions improve code quality assurance by ensuring that potential issues are identified and addressed in a structured manner.    Principle: Keep It Simple    Explanation: The "Exceptions" coding standard also supports the principle of "Keep It Simple." Developers are encouraged to use simple and clear exception handling strategies that avoid unnecessary complexity. By keeping exception handling straightforward, developers make the code more readable and maintainable, reducing the chances of introducing errors and potential security vulnerabilities related to complex exception logic. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Moderate | Medium | Low | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 18.0.0 | Exception Checker | clang-tidy's Exception Checker enforces the "Exceptions" coding standard by automatically identifying improper exception handling in C++ code. By using this checker, developers proactively ensure proper exception usage, enhancing code reliability and promoting robust exception handling practices. |
| Cppcheck | 2.11 | Cppcheck Exception Analysis | Cppcheck's Exception Analysis enforces the "Exceptions" coding standard by automatically detecting issues with exception handling in C and C++ code. Developers use this analysis to identify and address exception-related vulnerabilities, ensuring code reliability and secure exception handling. |
| Clang Static Analyzer | 18.0.0 | Clang Static Analyzer Exception Handling Checks | Clang Static Analyzer's Exception Handling Checks enforce the "Exceptions" coding standard by automatically detecting issues with exception handling in C and C++ applications. Integrating this checker enhances code reliability and promotes robust and secure exception handling practices. |
| CppDepend | 2020.1.0 | CppDepend Exception Rules | CppDepend's Exception Rules enforce the "Exceptions" coding standard by automatically detecting issues with exception handling in C++ codebases. Integrating this tool enhances code reliability and promotes secure and reliable exception handling practices. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secure Input Validation | STD-008-CPP | Name of Standard: Secure Input Validation Coding Standard  Rationalize the Standard: This standard focuses on implementing robust input validation techniques to ensure the integrity and security of user input, preventing common vulnerabilities such as cross-site scripting (XSS), SQL injection, and command injection. |

| **Noncompliant Code** |
| --- |
| Noncompliant Code Description: This code directly uses user input without any validation, leaving the application vulnerable to various injection attacks. |
| std::string userInput = getUserInput();  processData(userInput); |

| **Compliant Code** |
| --- |
| Compliant Code Description: This code validates the user input using an isValidInput function before processing it, ensuring that only valid input is processed and reducing the risk of injection attacks. |
| std::string userInput = getUserInput();  if (isValidInput(userInput)) {  processData(userInput);  } else {  handleInvalidInput();  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  Principle: Validate Input Data    Explanation: The "Secure Input Validation" coding standard directly aligns with the principle of "Validate Input Data." By adhering to this standard, developers ensure that all user inputs and external data are rigorously validated and sanitized before processing. Secure input validation helps prevent a wide range of security threats, such as SQL injection, cross-site scripting (XSS), and command injection attacks. By validating and sanitizing input data, developers ensure that only expected and safe inputs are processed, reducing the risk of malicious inputs compromising the system's security and stability.    Principle: Adopt a Secure Coding Standard    Explanation: The "Secure Input Validation" coding standard is an example of adopting a secure coding standard focused on validating input data securely. By implementing this standard, developers proactively address potential security risks associated with improper input validation, promoting more secure software development practices. The coding standard encompasses guidelines and best practices to ensure that all input validation is done in a secure and consistent manner across the codebase. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | High | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Microsoft Secure Code Analysis (SAST) | 3.0.1 | Input Validation Rules | Microsoft SAST's Input Validation Rules enforce the "Secure Input Validation" coding standard in C++ code. Developers use these rules to identify and address potential vulnerabilities related to user input, enhancing code security and robustness. |
| Clang Static Analyzer | 18.0.0 | Clang Static Analyzer Input Validation Checks | Clang Static Analyzer's Input Validation Checks enforce the "Secure Input Validation" coding standard in C and C++ applications. Integrating this checker enhances code security and promotes robust input validation practices, mitigating potential security risks related to user input. |
| Cppcheck | 2.11 | Cppcheck Input Validation Analysis | Cppcheck is a widely used static code analysis tool for C and C++ applications that includes an Input Validation Analysis feature to enforce the "Secure Input Validation" coding standard. Cppcheck's Input Validation Analysis automatically detects potential issues with input validation, such as buffer overflows or unvalidated user inputs. By utilizing Cppcheck's Input Validation Analysis, developers can ensure adherence to the coding standard and proactively address security risks related to user input. Integrating Cppcheck into the development process enhances code security and promotes secure and robust input validation in C and C++ applications. |
| Fortify Static Code Analyzer (SCA) | 22.2.1 | Fortify Input Validation Analysis | Fortify SCA's Input Validation Analysis enforces the "Secure Input Validation" coding standard by detecting potential issues with input validation in C and C++ applications. Integrating this analysis enhances code security and promotes robust input validation practices, mitigating potential security risks related to user input. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secure File Handling | STD-009-CPP | Name of Standard: Secure File Handling Coding Standard  Rationalize the Standard: This standard focuses on implementing secure file handling practices to ensure the confidentiality, integrity, and availability of files, mitigating risks such as unauthorized access, information disclosure, or file manipulation. |

| **Noncompliant Code** |
| --- |
| Noncompliant Code Description: This code opens a file for writing without considering potential security risks, such as file permissions or file overwrite vulnerabilities. |
| std::string filePath = "/path/to/file.txt";  std::ofstream file(filePath);  file << "Sensitive data";  file.close(); |

| **Compliant Code** |
| --- |
| Compliant Code Description: This code opens the file for writing in binary mode with truncation, explicitly specifying the file access modes. It also performs proper error handling in case the file fails to open. |
| std::string filePath = "/path/to/file.txt";  std::ofstream file(filePath, std::ios::out | std::ios::binary | std::ios::trunc);  if (file.is\_open()) {  file << "Sensitive data";  file.close();  } else {  handleFileOpenError();  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  Principle: Architect and Design for Security Policies    Explanation: The "Secure File Handling" coding standard aligns with the principle of "Architect and Design for Security Policies." By adhering to this standard, developers ensure that the software's file handling mechanisms are designed with security in mind. This involves defining clear access controls, implementing proper file permissions, and using secure file handling APIs. By thoughtfully designing file handling procedures, developers can prevent unauthorized access, data leaks, and potential security breaches related to file manipulation.    Principle: Use Effective Quality Assurance Techniques    Explanation: The "Secure File Handling" coding standard supports the principle of "Use Effective Quality Assurance Techniques." By providing specific guidelines for secure file handling, this standard aids in code quality assurance by ensuring that files are managed securely throughout their lifecycle. Adopting secure file handling techniques, such as input validation, proper error handling, and file path sanitization, helps reduce the risk of file-related vulnerabilities and improves the overall security of the software. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | High | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.11 | Cppcheck File Handling Analysis | Cppcheck's File Handling Analysis enforces the "Secure File Handling" coding standard by detecting potential issues with file handling in C and C++ applications. Integrating this analysis enhances code security and promotes secure file handling practices, mitigating potential security risks related to file operations. |
| Fortify Static Code Analyzer (SCA) | 2.22.1 | Fortify File Handling Analysis | Fortify SCA's File Handling Analysis enforces the "Secure File Handling" coding standard by detecting potential issues with file handling in C and C++ applications. Integrating this analysis enhances code security and promotes secure file handling practices, mitigating potential security risks related to file operations. |
| Clang Static Analyzer | 18.0.0 | Clang Static Analyzer File Handling Checks | Clang Static Analyzer is a powerful static code analysis tool for C and C++ applications that includes dedicated File Handling Checks. This feature automatically detects potential issues with file handling, such as unsafe file operations or improper file permissions. By utilizing Clang Static Analyzer's File Handling Checks, developers can ensure compliance with the "Secure File Handling" coding standard and proactively address security risks related to file handling. Integrating Clang Static Analyzer into the development process enhances code security and promotes secure file handling practices in C and C++ applications. |
| CppDepend | 2020.1.0 | CppDepend File Handling Rules | CppDepend's File Handling Rules enforce the "Secure File Handling" coding standard by detecting potential issues with file handling in C++ codebases. Integrating this tool enhances code security and promotes secure file handling practices, mitigating potential security risks related to file operations. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secure Logging | STD-010-CPP | Name of Standard: Secure Logging Coding Standard  Rationalize the Standard: This standard focuses on implementing secure logging practices to ensure the confidentiality, integrity, and availability of logged information, preventing risks such as sensitive data exposure, log injection, or tampering. |

| **Noncompliant Code** |
| --- |
| Noncompliant Code Description: This code directly concatenates sensitive information (username) into the log message without considering potential security risks, such as log injection or exposure of sensitive data. |
| std::string logMessage = "User logged in: " + username;  writeToLogFile(logMessage); |

| **Compliant Code** |
| --- |
| Compliant Code Description: This code sanitizes the username using a sanitizeString function and includes it as a separate field in the log message, preventing log injection and ensuring sensitive information is properly protected. |
| std::string logMessage = "User logged in: [username: " + sanitizeString(username) + "]";  writeToLogFile(logMessage); |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):**  Principle: Architect and Design for Security Policies    Explanation: The "Secure Logging" coding standard aligns with the principle of "Architect and Design for Security Policies." By following this standard, developers ensure that logging mechanisms are designed with security in mind. This involves defining clear logging policies, avoiding the inclusion of sensitive information in logs, and implementing proper access controls for log files. Secure logging practices help maintain the confidentiality and integrity of sensitive data, reducing the risk of exposing critical information to unauthorized parties.    Principle: Use Effective Quality Assurance Techniques    Explanation: The "Secure Logging" coding standard supports the principle of "Use Effective Quality Assurance Techniques." By providing specific guidelines for secure logging, this standard aids in code quality assurance by ensuring that logging is performed securely and accurately. Following the standard, developers can verify that logs are comprehensive, properly formatted, and do not reveal sensitive details that could be exploited by attackers. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Moderate | Medium | Low | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Log4cpp | 1.1.4 | Log4cpp Secure Logging Module | Log4cpp is a C++ logging library that includes a dedicated Secure Logging Module to enforce the "Secure Logging" coding standard. The Secure Logging Module ensures that sensitive information is appropriately masked or redacted when logged, preventing exposure of sensitive data in log files. By utilizing Log4cpp's Secure Logging Module, developers can ensure compliance with the coding standard and proactively address security risks related to logging. Integrating Log4cpp into the logging process enhances code security and promotes secure logging practices in C++ applications. |
| Google's glog | 117.0.5 | glog Secure Logging Features | glog's Secure Logging Features enforce the "Secure Logging" coding standard by protecting sensitive information during logging in C++ applications. Integrating this tool enhances code security and promotes secure logging practices, mitigating potential risks related to sensitive data exposure. |
| Boost.Log | 1.71.0 | Boost.Log Secure Logging Module | Boost.Log is a C++ logging library that includes a dedicated Secure Logging Module to enforce the "Secure Logging" coding standard. The Secure Logging Module ensures that sensitive information is handled securely during logging, preventing unauthorized access to confidential data in log files. By utilizing Boost.Log's Secure Logging Module, developers can ensure compliance with the coding standard and proactively address security risks related to logging. Integrating Boost.Log into the logging process enhances code security and promotes secure logging practices in C++ applications. |
| Microsoft C++ Rest SDK (Casablanca) | 2.10.16 | Casablanca Secure Logging Features | Casablanca's Secure Logging Features enforce the "Secure Logging" coding standard by protecting sensitive information during logging in C++ applications. Integrating this tool enhances code security and promotes secure logging practices, mitigating potential risks related to sensitive data exposure. |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

Explanation:

1. Assess and Plan (Sec - DevSecOps): In the "Assess and Plan" stage, security teams collaborate with development and operations to assess and plan security measures, including defining security coding standards and best practices.

2. Design, Build, Verify, and Test: During the "Design," "Build," "Verify and Test" (Dev - DevSecOps) stages, developers write code following the security coding standards. Integrate automated security code scanning tools (e.g., Cppcheck, Clang Static Analyzer) into the Continuous Integration (CI) pipeline to verify and test the code for security vulnerabilities and compliance with the standards.

3. Pre-Production (Verify and Test): In the "Verify and Test" (Dev - DevSecOps) and "Pre-Production" stages, enhance security testing to validate code against security standards before deployment to production. Use automated security testing tools (e.g., Fortify SCA, Checkmarx) to assess the application for security flaws, including SQL injection and memory protection issues related to the coding standards.

4. Transition (Ops): In the "Transition" stage, the validated code is transitioned to the staging or production environment.

5. Health Checks (Ops): In the "Health Checks" stage, perform automated health checks to monitor the application's performance and security in the production environment.

6. Monitor and Detect (Ops): In the "Monitor and Detect" stage, establish continuous monitoring and centralized logging mechanisms using SIEM tools. This allows for the prompt detection of security incidents or anomalies.

7. Respond (Ops): In the "Respond" stage, promptly respond to security incidents to mitigate risks.

8. Maintain and Stabilize (Ops): In the "Maintain and Stabilize" stage, conduct regular maintenance to ensure the application remains secure and stable over time.

By modifying the existing DevOps process to integrate security measures and automation at each stage as described above, Green Pace can efficiently enforce the security standards defined in the policy. This approach ensures that security is an intrinsic part of the software development lifecycle and helps prevent security vulnerabilities from being introduced into the system.

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | Medium | Likely | Low | Medium | 2 |
| STD-003-CPP | High | Likely | High | High | 4 |
| STD-004-CPP | Medium | Unlikely | Medium | Medium | 2 |
| STD-005-CPP | High | Likely | High | High | 4 |
| STD-006-CPP | Medium | Likely | Medium | Medium | 3 |
| STD-007-CPP | Low | Unlikely | Low | Low | 1 |
| STD-008-CPP | High | Likely | High | High | 4 |
| STD-009-CPP | Medium | Likely | Medium | Medium | 3 |
| STD-010-CPP | Low | Unlikely | Low | Low | 1 |
| STD-011-CPP | High | Unlikely | High | High | 3 |
| STD-0012-CPP | Medium | Unlikely | Medium | Medium | 2 |
| STD-013-CPP | Low | Unilkely | Low | Low | 1 |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption at rest is a security measure that involves encrypting data while it is stored in a non-volatile location, such as a database, disk, or backup storage. The policy applies to all sensitive data stored in databases and other storage systems within the organization. Data should be encrypted using strong encryption algorithms and keys to protect against unauthorized access, data breaches, and theft in case of physical device theft or unauthorized access to storage systems. The encryption keys should be securely managed to ensure the integrity and confidentiality of the data even if the storage media is compromised. Compliance with this policy ensures that sensitive data is protected from unauthorized access and data exposure in case of a security breach or physical theft. |
| Encryption at flight | Encryption in flight, also known as data-in-motion encryption, involves securing data while it is in transit over networks or communication channels. This policy applies to all communication protocols (e.g., HTTPS, TLS) used to transmit sensitive data, both within the internal network and over external connections. All sensitive data, including user credentials, payment information, and any other confidential data, must be encrypted using secure protocols during transmission. This ensures that the data remains confidential and protected from interception or eavesdropping by unauthorized entities. Compliance with this policy prevents data breaches and man-in-the-middle attacks by safeguarding the data during its journey across various networks. |
| Encryption in use | Encryption in use, also known as runtime encryption, refers to the practice of encrypting sensitive data while it is being processed or used by applications or in memory. This policy applies to applications handling sensitive data, especially those dealing with personally identifiable information (PII) and other confidential information. The policy mandates the use of encryption libraries and techniques to protect data during application runtime and in memory, minimizing the risk of data leakage through memory dumps or other runtime attacks. Compliance with this policy ensures that sensitive data remains encrypted and protected at all times, even during processing and memory usage, reducing the risk of data exposure and unauthorized access by malicious actors. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is a fundamental security strategy that ensures individuals or entities attempting to access systems or data are verified and granted appropriate access based on their identity. The policy applies to all user accounts, applications, and devices accessing the organization's resources. It requires implementing strong authentication mechanisms, such as multi-factor authentication (MFA), to strengthen the verification process. Proper authentication is crucial in preventing unauthorized access to sensitive data, systems, and applications, reducing the risk of data breaches and data loss due to unauthorized access. Compliance with this policy ensures that only authorized personnel can access critical resources and data, enhancing the overall security posture of the organization.  User Logins: Implement mechanisms to log user logins and logouts, capturing user identities, timestamps, and IP addresses. Retain login records for audit and monitoring purposes to trace user activities. |
| Authorization | Authorization is a security strategy that defines the privileges and permissions granted to authenticated users or entities based on their roles and responsibilities. The policy applies to all systems, applications, and data resources. It requires implementing role-based access control (RBAC) and least privilege principles to limit user access to only the resources necessary for their roles. Proper authorization ensures that users can only perform actions and access data relevant to their job functions, reducing the risk of data breaches and unauthorized access to critical data. Compliance with this policy ensures that the principle of least privilege is followed, minimizing the potential damage from insider threats or compromised accounts.  User Level of Access: Maintain logs of user access levels, permissions, and role changes. Regularly review and update user access rights based on job roles and responsibilities. |
| Accounting | Accountability is a security strategy that establishes responsibility and traceability for actions performed within the organization's systems and applications. The policy applies to all users and administrators with access to sensitive data or critical systems. It requires implementing audit logs and monitoring mechanisms to track user activities and detect any suspicious or unauthorized actions. Accountability plays a significant role in identifying security incidents, data breaches, and insider threats, ensuring timely responses and mitigation. Compliance with this policy ensures that all user actions are logged and can be audited if necessary, strengthening the organization's ability to investigate and respond to security incidents and ensuring the responsible use of data and resources.  Changes to the Database: Establish audit trails for database modifications, including insertions, updates, and deletions of sensitive data. Ensure that all changes to the database are logged and attributed to the relevant users or administrators.  Addition of New Users: Log the creation of new user accounts and associated access privileges. Keep records of user onboarding and removal to track user access over time.  Files Accessed by Users: Monitor and record files accessed by users, both locally and over the network. Maintain logs of file access activities to detect unauthorized access or potential data breaches. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

Principles:

1. Validate Input Data

2. Heed Compiler Warnings

3. Architect and Design for Security Policies

4. Keep It Simple

5. Default Deny

6. Adhere to the Principle of Least Privilege

7. Sanitize Data Sent to Other Systems

8. Practice Defense in Depth

9. Use Effective Quality Assurance Techniques

10. Adopt a Secure Coding Standard

1. Data Type Coding Standard

- Principles: Validate Input Data

- Justification: The Data Type Coding Standard ensures consistent and proper usage of data types in the code. By validating input data, developers can prevent common vulnerabilities like buffer overflows or type mismatches that may lead to security vulnerabilities. Proper data type validation helps to maintain the integrity and stability of the codebase, reducing the risk of security breaches due to incorrect data handling.

2. tring Correctness Coding Standard

- Principles: Validate Input Data, Keep It Simple

- Justification: String Correctness standard focuses on handling strings safely to prevent buffer overflows and other string-related vulnerabilities. By validating input data and keeping the code simple, developers can minimize the chances of malicious input causing unintended behaviors in string operations, thereby mitigating potential security risks.

3. SQL Injection Coding Standard

- Principles: Validate Input Data, Adhere to the Principle of Least Privilege

- Justification: The SQL Injection standard aims to prevent SQL injection attacks by properly validating and sanitizing user input. By following the principle of least privilege, the application limits database access only to the necessary operations and prevents attackers from gaining unauthorized access or manipulating the database using malicious input.

4. Memory Protection Coding Standard

- Principles: Heed Compiler Warnings, Practice Defense in Depth

- Justification: The Memory Protection standard focuses on safe memory handling and avoiding common memory-related vulnerabilities like buffer overflows. By heeding compiler warnings, developers can identify potential memory-related issues early in the development process. Additionally, practicing defense in depth by implementing memory protection mechanisms can help prevent exploitation of memory-related vulnerabilities.

5. Assertions Coding Standard

- Principles: Use Effective Quality Assurance Techniques

- Justification: The Assertions standard emphasizes the use of assertions in the code to detect and report unexpected conditions during testing. By employing effective quality assurance techniques, such as assertions, developers can identify and address potential security issues and ensure the code behaves as expected.

6. Exceptions Coding Standard

- Principles: Use Effective Quality Assurance Techniques

- Justification: The Exceptions standard focuses on proper handling of exceptions to prevent unhandled exceptions that could lead to system crashes or security vulnerabilities. By using effective quality assurance techniques, developers can thoroughly test exception handling scenarios to ensure robustness and identify potential weaknesses in the code.

7. Secure Input Validation Coding Standard

- Principles: Sanitize Data Sent to Other Systems, Adhere to the Principle of Least Privilege

- Justification: The Secure Input Validation standard aims to prevent security vulnerabilities arising from improper input validation. By sanitizing data sent to other systems and adhering to the principle of least privilege, the application ensures that only necessary and safe data is transmitted, minimizing the risk of data breaches or unauthorized access.

8. Secure File Handling Coding Standard

- Principles: Practice Defense in Depth, Default Deny

- Justification: The Secure File Handling standard addresses secure file operations and file access. By practicing defense in depth, the application implements multiple layers of security controls around file handling, preventing unauthorized access or tampering with critical files. Additionally, following the default deny principle ensures that file access is explicitly restricted, reducing the attack surface for potential threats.

9. Secure Logging Coding Standard

- Principles: Default Deny, Practice Defense in Depth

- Justification: The Secure Logging standard focuses on secure and appropriate logging practices to maintain audit trails and detect potential security incidents. By following the default deny principle, the application limits access to log files and ensures that only authorized personnel can view sensitive log data. Moreover, practicing defense in depth ensures that logs are encrypted, protected, and monitored, reducing the risk of unauthorized access to log files and preventing potential insider threats.

By linking the principles to the relevant standards, Green Pace can demonstrate that its security policy is based on widely accepted principles, providing a solid foundation for secure coding and system development practices.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial release of the security policy. | David Buksbaum | Sarah Thompson |
| 2.0 | 12/12/2021 | Updated encryption and authentication sections. | Jane Doe | Michael Johnson |
| 3.0 | 06/30/2023 | Added data protection measures for new services. | John Smith | Emily Anderson |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

| Language | Acronym |
| --- | --- |
| C++ | CPP |
| C | CLG |
| Java | JAV |