**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 | Ensure all input data is checked for correctness to prevent vulnerabilities such as SQL injection or buffer overflows. |
| 1. Heed Compiler Warnings | Pay attention to compiler warnings as they often indicate potential vulnerabilities. |
| 1. Architect and Design for Security Policies | Incorporate security policies into the design phase to mitigate risks early. |
| 1. Keep It Simple | Avoid unnecessary complexity in code, as it can introduce security flaws. |
| 1. Default Deny | Restrict access by default and explicitly allow only what is necessary. |
| 1. Adhere to the Principle of Least Privilege | Assign only the minimum permissions necessary to reduce potential damage from exploits. |
| 1. Sanitize Data Sent to Other Systems | Ensure data is sanitized to prevent injection attacks or improper handling by external systems. |
| 1. Practice Defense in Depth | Use multiple layers of defense to protect systems from various attack vectors. |
| 1. Use Effective Quality Assurance Techniques | Regularly test and verify code to identify and fix vulnerabilities early. |
| 1. Adopt a Secure Coding Standard | Follow established coding standards like SEI CERT to maintain consistent and secure practices. Follow established coding standards like SEI CERT to maintain consistent and secure practices. Follow established coding standards like SEI CERT to maintain consistent and secure 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 non-compliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | This standard ensures that data types are correctly validated to prevent type mismatches and other issues that could lead to undefined behavior or vulnerabilities. For example, improper type handling can cause buffer overflows, crashes, or corrupted data. |

| **Noncompliant Code** |
| --- |
| This code does not validate the user input, leading to potential unexpected behavior if incorrect data is provided. |
| int userAge;  cin >> userAge;  if (userAge < 0) {  // Invalid input is not adequately handled.  } |

| **Compliant Code** |
| --- |
| This code validates the input to ensure correctness, rejecting negative values and handling invalid input gracefully. |
| int userAge;  cin >> userAge;  if (!cin || userAge < 0) {  throw runtime\_error("Invalid age entered.");  } |

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

| **Validate Input Data:**  Ensures all input data is checked for correctness to prevent vulnerabilities like buffer overflows or improper data handling.  **Keep It Simple:**  Reduces code complexity, ensuring straightforward validation processes that are easier to maintain and debug. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.5 | Data Validation Rules | Static analysis to detect improper validation of data types |
| Veracode | 2024 | Input Validation Checker | Scans for common input validation issues in C++ code |
| CheckMarx | V2024 | Boundary Checking Rules | Detect vulnerabilities related to incorrect data handling. Such a missing input validation |
| Clang-Tidy | V13.0 | Range Enforcement Rules | Provides rules for catching potential issues with data type mismatches and improper validation |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-003-CPP | This standard ensures that all data values are validated to confirm they fall within acceptable ranges. It prevents errors caused by invalid or maliciously crafted data, which could lead to unexpected behavior or system failures. For instance, ensuring that numeric values do not exceed predefined boundaries is critical to maintaining system stability. |

| **Noncompliant Code** |
| --- |
| The code below does not validate the range of userScore, potentially allowing invalid values that could disrupt downstream processes. |
| int userScore;  cin >> userScore;  // No validation of acceptable value range. |

| **Compliant Code** |
| --- |
| The code below ensures that userScore falls within the acceptable range of 0 to 100, rejecting any out-of-bound values. |
| int userScore;  cin >> userScore;  if (!cin || userScore < 0 || userScore > 100) {  throw runtime\_error("Invalid score entered. Please enter a value between 0 and 100.");  } |

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

| **Validate Input Data:**  Ensures that all inputs are checked for correctness, reducing the likelihood of invalid or malicious values entering the system.  **Practice Defense in Depth:**  Adds a layer of protection by validating critical data at the input level, minimizing risks further downstream. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.5 | Data Range Validation | Detects missing range checks for numeric values |
| Veracode | 2024 | Input Validation Checker | Identifies potential issues with data inputs not being validated |
| Checkmarx | 2024 | Boundary Checking Rules | Flags code where boundary validation is missing for critical data |
| Clang-Tidy | 13.0 | Range Enforcement Rule | Ensures numeric data complies with predefined acceptable ranges |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-004-CPP | This standard ensures proper handling of strings in C++ to prevent buffer overflows, memory corruption, and security vulnerabilities. Unsafe string handling can result in application crashes, corrupted data, or unauthorized access to sensitive information. |

| **Noncompliant Code** |
| --- |
| The code below uses strcpy without validating the length of the source string, which can lead to a buffer overflow. |
| char destination[10];  char source[] = "This string is too long for the destination buffer";  strcpy(destination, source); // No bounds checking |

| **Compliant Code** |
| --- |
| The code below uses strncpy to ensure the source string fits within the destination buffer, preventing a buffer overflow. |
| char destination[10];  char source[] = "Safe";  strncpy(destination, source, sizeof(destination) - 1);  destination[sizeof(destination) - 1] = '\0'; // Null-terminate the string |

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

| **Adopt a Secure Coding Standard:**  Ensures the use of safe string manipulation functions to avoid common pitfalls.  **Practice Defense in Depth:**  Adds layers of security by validating and limiting operations to the safe bounds of memory. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.5 | String Manipulation Rules | Detects unsafe string functinons like strcpy and suggests safer alternatives |
| Veracode | 2024 | Buffer Overflow Scanner | Identifies potential overflows in string operations |
| Checkmarx | 2024 | String Handling Checks | Flags improper usage of unsafe string manipulation function |
| Clang-Tidy | 13.0 | Safe String Rules | Ensures use of secure string functions and adherence to modern C++ practices |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-005-CPP | This standard ensures that all database queries are constructed securely using parameterized queries or prepared statements. SQL injection attacks can allow unauthorized access, data breaches, or corruption of critical information. Proper validation and sanitization of inputs are critical to preventing this vulnerability. |

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

| **Compliant Code** |
| --- |
| The code below uses a prepared statement to securely parameterize user input, ensuring that it cannot be executed as SQL code. |
| string username = getUserInput();  string query = "SELECT \* FROM users WHERE username = ?";  PreparedStatement preparedStatement = databaseConnection.prepareStatement(query);  preparedStatement.setString(1, username);  preparedStatement.executeQuery(); |

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

| **Validate Input Data:**  Ensures all inputs are checked for correctness to prevent malicious commands.  **Sanitize Data Sent to Other Systems:**  Prevents injection of unintended commands by sanitizing user inputs. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.5 | SQL Injection Rules | Detects concatenated queries and recommend parameterized queries |
| Veracode | 2024 | Query Injection Scanner | Identify queries vulnerable to SQL injection attacks |
| Checkmarx | 2024 | SQL Injection Checks | Flags unsafe string concatenation in database queries |
| Fortify Static Code Analyzer | 21.1 | Database Query Analyzer | Ensures database interactions are secure and follow best practices |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-001-LLL] | Data Type Validation |

| **Noncompliant Code** |
| --- |
| Ensures type correctness to avoid type mismatches or overflows. |
| int userAge = 0;  cin >> userAge;  if (userAge < 0) {  // This can lead to unexpected results if a large unsigned int is passed.  throw runtime\_error("Invalid age");  } |

| **Compliant Code** |
| --- |
| Validates input and ensures type safety. |
| int userAge = 0;  cin >> userAge;  if (!cin || userAge < 0) {  throw runtime\_error("Invalid age");  } |

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

| **Validate Input Data:**  Ensures input is checked to avoid invalid or dangerous data that could lead to memory corruption or crashes.  **Keep It Simple:**  Avoids complexity in code and keeps it simple, reducing the risk of errors when validating memory or data types. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.5 | Memory Safety Checks | Detects potential memory leaks or improper memory accesses |
| Veracode | 2024 | Memory Access Analyzer | Identify unsafe memory operations and checks for type mismatches |
| Checkmarx | 2024 | Memory Safety Rule | Flags memory issues like buffer overflows and improper validation |
| Clang-Tidy | 13.0 | Memory Management Rules | Ensures safe memory management practices in the code |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | This standard ensures that assertions are used appropriately to verify assumptions and preconditions in the code, especially in debugging or development environments. Assertions can catch bugs early by ensuring that certain conditions hold true during execution. However, they should not be used for error handling in production code, as they may be disabled in optimized builds. |

| **Noncompliant Code** |
| --- |
| The code below uses assertions incorrectly for runtime error handling. Assertions should not be used for runtime exceptions in production. |
| assert(userAge >= 0); // Checks that userAge is valid, but should not throw errors in production. |

| **Compliant Code** |
| --- |
| The code below uses assertions to check assumptions during debugging and development but uses proper error handling for runtime conditions. |
| assert(userAge >= 0); // Validates input during development.  if (userAge < 0) {  throw runtime\_error("Invalid age.");  } |

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

| **Keep It Simple:**  Ensures that assertions are used for simple checks during development without introducing complex error-handling logic in production code.  **Fail Securely:**  Encourages catching runtime errors via proper exception handling rather than relying on assertions, which may be disabled in optimized builds. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.5 | Assertion Misuse Check | Detects improper usage of assertions in runtime code |
| Veracode | 2024 | Runtime Validation Checker | Flags the use of assertion for runtime checks in production code |
| Checkmarx | 2024 | Assertion Handling Rules | Ensures that assertions are only used in debugging environments and not for error handling |
| Clang-Tidy | 13.0 | Debugging Check Rules | Detects assertions used for runtime checks and suggests better alternatives |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | This standard ensures that exceptions are used appropriately for error handling. Using exceptions allows for graceful handling of unexpected conditions, preventing crashes and ensuring that errors are caught and addressed without corrupting the program state. It’s critical to distinguish between recoverable errors and programming bugs, handling the former via exceptions and not relying on them for control flow. |

| **Noncompliant Code** |
| --- |
| The code below uses exceptions for controlling normal program flow, which is inefficient and improper. Exceptions should only be used for exceptional or error conditions. |
| try {  for (int i = 0; i < 10; i++) {  if (i == 5) {  throw runtime\_error("Reached halfway"); // Unnecessary use of exception for control flow.  }  // Regular processing  }  } catch (const runtime\_error& e) {  // Handle the "exception" here  } |

| **Compliant Code** |
| --- |
| The code below uses exceptions only to handle errors that cannot be anticipated or recovered from normally. It does not misuse exceptions for regular flow control. |
| try {  // Regular processing  for (int i = 0; i < 10; i++) {  if (i == 5) {  // Do something, no exception needed  }  }  } catch (const runtime\_error& e) {  // Handle runtime error, e.g., failed file open, network failure, etc.  } |

**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.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.5 | Exception Handling Rules | Detects improper use of exceptions control flow |
| Veracode | 2024 | Exception Handling Analyzer | Flags exceptions used for non-error situations |
| Checkmarx | 2024 | Exception Misuse Check | Ensures exceptions are used only for exceptional error conditions |
| Clang-Tidy | 13.0 | Exception Misuse Detector | Identifies exceptions used outside of error handling |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Validation | STD-003-CPP | This standard ensures that all user inputs are validated before being processed. Input validation is crucial to prevent common vulnerabilities like buffer overflows, SQL injections, and cross-site scripting (XSS). Proper validation ensures that only properly formatted and safe data is processed by the system. |

| **Noncompliant Code** |
| --- |
| The code below directly uses user input without validating it, allowing potentially harmful or incorrect data to be processed. |
| string userInput;  cin >> userInput;  cout << "User input: " << userInput << endl; // User input is not validated |

| **Compliant Code** |
| --- |
| The code below validates user input to ensure it only contains acceptable characters, preventing issues like code injection or buffer overflow. |
| string userInput;  cin >> userInput;  if (userInput.empty() || !all\_of(userInput.begin(), userInput.end(), ::isalnum)) {  throw runtime\_error("Invalid input. Only alphanumeric characters are allowed.");  }  cout << "User input: " << userInput << endl; |

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

| **Validate Input Data:**  Ensures user input is checked and validated to avoid malicious or malformed data from being processed.  **Fail Securely:**  Guarantees that invalid input is properly rejected without causing unexpected behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.5 | Input Validation Rule | Detects lack of input validation and potential security risks |
| Veracode | 2024 | Input Validation Scanner | Flags unvalidated input that could lead to vulnerabilities |
| Checkmarx | 2024 | Input Integrity Check | Ensures input is correctly validated before processing |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Buffer Overflow Prevention | STD-009-CP | This standard ensures that memory buffers are properly managed to avoid buffer overflows, which can lead to data corruption, application crashes, and security vulnerabilities. Using safe memory handling functions and controlling buffer sizes is critical to maintaining program integrity. |

| **Noncompliant Code** |
| --- |
| The code below uses strcpy without bounds checking, which can cause a buffer overflow if the input exceeds the buffer size. |
| char buffer[10];  strcpy(buffer, "This string is too long for the buffer"); |

| **Compliant Code** |
| --- |
| The code below uses strncpy to ensure that the string copied to the buffer does not exceed its size, preventing a buffer overflow. |
| char buffer[10];  strncpy(buffer, "This string is safe", sizeof(buffer) - 1);  buffer[sizeof(buffer) - 1] = '\0'; // Null-terminate the string |

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

| **Sanitize Data Sent to Other Systems:**  Ensures that data fits within allocated buffers to prevent overflow and corruption.  **Keep It Simple:**  Avoids complex and error-prone operations like strcpy in favor of safer, more manageable alternatives like strncpy. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.5 | Buffer Overflow Prevention | Detects unsafe string copy operations that could cause buffer overflows |
| Veracode | 2024 | Memory Safety Scanner | Flags potential buffer overflow risks in memory operations |
| Checkmarx | 2024 | Memory Protection Check | Identifies unprotected buffers and potential overflow issues |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secure File Handling | STD-010-CPP | This standard ensures that file handling operations, including reading and writing to files, are performed securely. Insecure file handling can lead to vulnerabilities such as file path manipulation, privilege escalation, and unauthorized access to sensitive files. |

| **Noncompliant Code** |
| --- |
| The code below opens a file using user input without validating the file path, leading to potential directory traversal attacks. |
| string filename;  cin >> filename;  ifstream file(filename); // No validation of user input, may open unauthorized files |

| **Compliant Code** |
| --- |
| The code below validates the file path to ensure it points to a safe location, preventing directory traversal or unauthorized file access. |
| string filename;  cin >> filename;  if (filename.find("..") != string::npos || filename.find("/") == 0) {  throw runtime\_error("Invalid file path.");  }  ifstream file(filename); |

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

| **Validate Input Data:**  Ensures user-provided file paths are validated to prevent unauthorized access.  **Practice Defense in Depth:**  Uses multiple layers of security to protect sensitive files from unauthorized access. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.5 | File Path Validation | Detects unsafe file handling that could lead to unauthorized file access |
| Veracode | 2024 | File Access Integrity | Flags insecure file handling that could lead to path manipulation |
| Checkmarx | 2024 | Path Validation Checker | Ensures file paths are validated to prevent unauthorized access |

### 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

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | This standard ensures that data types are correctly validated to prevent type mismatches and other issues that could lead to undefined behavior or vulnerabilities. For example, improper type handling can cause buffer overflows, crashes, or corrupted data. |
| **SQL Injection Prevention** | STD-002-CPP | This standard ensures that all SQL queries are constructed securely by using parameterized queries or prepared statements to prevent SQL injection attacks. SQL injections allow malicious users to manipulate SQL queries, potentially leading to unauthorized data access, data manipulation, or data corruption. Proper handling of user input and using secure database query methods is critical to protecting applications from such vulnerabilities. |
| **Data Value** | STD-003-CPP | This standard ensures that all data values are validated to confirm they fall within acceptable ranges. It prevents errors caused by invalid or maliciously crafted data, which could lead to unexpected behavior or system failures. For instance, ensuring that numeric values do not exceed predefined boundaries is critical to maintaining system stability. |
| **String Correctness** | STD-004-CPP | This standard ensures proper handling of strings in C++ to prevent buffer overflows, memory corruption, and security vulnerabilities. Unsafe string handling can result in application crashes, corrupted data, or unauthorized access to sensitive information. |
| **SQL Injection** | STD-005-CPP | This standard ensures that all database queries are constructed securely using parameterized queries or prepared statements. SQL injection attacks can allow unauthorized access, data breaches, or corruption of critical information. Proper validation and sanitization of inputs are critical to preventing this vulnerability. |
| **Assertions** | STD-006-CPP | This standard ensures that assertions are used appropriately to verify assumptions and preconditions in the code, especially in debugging or development environments. Assertions can catch bugs early by ensuring that certain conditions hold true during execution. However, they should not be used for error handling in production code, as they may be disabled in optimized builds. |
| **Exceptions** | STD-007-CPP | This standard ensures that exceptions are used appropriately for error handling. Using exceptions allows for graceful handling of unexpected conditions, preventing crashes and ensuring that errors are caught and addressed without corrupting the program state. It’s critical to distinguish between recoverable errors and programming bugs, handling the former via exceptions and not relying on them for control flow. |
| **Input Validation** | STD-008-CPP | This standard ensures that all user inputs are properly validated before being processed. Input validation is crucial to preventing common vulnerabilities such as buffer overflows, SQL injection, and cross-site scripting (XSS). Proper validation ensures that only well-formed, safe data is processed by the system, improving security and stability. |
| Buffer Overflow Prevention | STD-009-CP | This standard ensures that memory buffers are properly managed to avoid buffer overflows, which can lead to data corruption, application crashes, and security vulnerabilities. Using safe memory handling functions and controlling buffer sizes is critical to maintaining program integrity. |
| Secure File Handling | STD-010-CPP | This standard ensures that file handling operations, including reading and writing to files, are performed securely. Insecure file handling can lead to vulnerabilities such as file path manipulation, privilege escalation, and unauthorized access to sensitive files. |

### Risk Assessment

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Data Type | STD-001-CPP |  |  |  |
| Medium | Medium | Low | Medium | 3 |
| **String Correctness** | STD-004-CPP |  |  |  |
| Medium | Medium | Low | Medium | 3 |
| **String Correctness** | STD-004-CPP |  |  |  |
| High | Medium | Medium | High | 4 |
| **SQL Injection** | STD-005-CPP |  |  |  |
| High | Medium | Medium | High | 4 |
| **Memory Protection** | [STD-001-LLL] |  |  |  |
| Medium | Medium | Low | Medium | 3 |
| **Assertions** | STD-006-CPP |  |  |  |
| Medium | Medium | Low | Medium | 3 |
| **Exceptions** | STD-007-CPP |  |  |  |
| High | Medium | Medium | High | 4 |

### Automated Detection

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.5 | Exception Handling Rules | Detects improper use of exceptions control flow |
| Veracode | 2024 | Exception Handling Analyzer | Flags exceptions used for non-error situations |
| Checkmarx | 2024 | Exception Misuse Check | Ensures exceptions are used only for exceptional error conditions |
| Clang-Tidy | 13.0 | Exception Misuse Detector | Identifies exceptions used outside of error handling |

### 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.

[Insert your written explanations here.]

### 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 | High | Medium | Medium | High | 4 |
| STD-003-CPP | Medium | Medium | Low | Medium | 3 |
| STD-004-CPP | Medium | Medium | Low | Medium | 3 |
| STD-005-CPP | High | Medium | Medium | High | 4 |
| STD-006-CPP | Medium | Medium | Low | High | 3 |
| STD-007-CPP | High | Medium | Medium | High | 4 |
| STD-008-CPP | High | Medium | Medium | High | 4 |
| STD-009-CPP | High | High | Medium | High | 4 |
| STD-010-CPP | High | Medium | Medium | High | 4 |
| STD-011-CPP | Medium | Low | Low | Low | 2 |
| STD-012-CPP | High | Medium | Medium | High | 4 |
| STD-013-CPP | Medium | Medium | Medium | High | 3 |
| STD-014-CPP | High | High | High | High | 5 |

### 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 at rest | Encryption at rest prevents unauthorized access to sensitive data in the event of breach, theft, or loss of physical devices (e.g., external hard drives, servers). It ensures compliance with privacy regulations such as GDPR and HIPAA. |
| Encryption in flight | Encryption in flight ensures that data cannot be intercepted, modified, or accessed by unauthorized parties during transmission. This is crucial for preventing **Man-in-the-Middle (MitM)** attacks and securing communication channels. |
| Encryption in use | Encryption in use protects data during its active state, preventing unauthorized access or manipulation, especially in multi-tenant environments or when data is processed on shared systems. This type of encryption is critical for protecting sensitive data from being exposed during operations or computations. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying the identity of a user or system to ensure that only authorized entities can access the system or network. Authentication ensures that only legitimate users can access systems or sensitive data. It helps mitigate risks like unauthorized access, data breaches, and identity theft. |
| Authorization | Authorization determines the level of access granted to authenticated users based on their roles or permissions. It ensures users only have access to the resources and actions they are permitted to perform. Authorization ensures that users can only perform actions or access data that they are explicitly permitted to, preventing unauthorized access and minimizing the risk of insider threats or accidental data exposure. |
| Accounting | Accounting refers to the process of tracking and logging user activities to monitor for suspicious behavior, ensure compliance, and provide an audit trail for investigations. Accounting ensures that any unauthorized access or data manipulation can be traced back to the responsible user. It provides a crucial layer of visibility for detecting security breaches and ensures compliance with legal or regulatory requirements. |

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

* **User login:** Ensure users authenticate using strong methods (e.g., MFA) and their logins are logged.
* **Changes to the database:** Track and log all database modifications, especially those related to sensitive data.
* **Addition of new users:** Log the creation of new accounts and assign appropriate roles and permissions.
* **User level of access:** Implement and monitor role-based access control (RBAC) or other mechanisms to restrict access.
* **Files accessed by users:** Track and log file access by users to ensure sensitive files are not accessed inappropriately.

### Map the Principles

**Principles Overview:**

1. **Validate Input Data**
2. **Sanitize Data Sent to Other Systems**
3. **Practice Defense in Depth**
4. **Use Secure Authentication**
5. **Keep It Simple**
6. **Keep It Modular**
7. **Ensure Proper Access Control**
8. **Use Safe and Secure Algorithms**
9. **Ensure Accountability**
10. **Fail Securely**

| 1. **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | This standard ensures that data types are correctly validated to prevent type mismatches and other issues that could lead to undefined behavior or vulnerabilities. For example, improper type handling can cause buffer overflows, crashes, or corrupted data. |
| **SQL Injection Prevention** | STD-002-CPP | Input must be validated to ensure that no harmful characters or SQL statements can be passed through. This helps prevent malicious data injection attempts. |
| **Data Value** | STD-003-CPP | This standard ensures that all data values are validated to confirm they fall within acceptable ranges. It prevents errors caused by invalid or maliciously crafted data, which could lead to unexpected behavior or system failures. For instance, ensuring that numeric values do not exceed predefined boundaries is critical to maintaining system stability. |
| **String Correctness** | STD-004-CPP | This standard ensures proper handling of strings in C++ to prevent buffer overflows, memory corruption, and security vulnerabilities. Unsafe string handling can result in application crashes, corrupted data, or unauthorized access to sensitive information. |
| **SQL Injection** | STD-005-CPP | This standard ensures that all database queries are constructed securely using parameterized queries or prepared statements. SQL injection attacks can allow unauthorized access, data breaches, or corruption of critical information. Proper validation and sanitization of inputs are critical to preventing this vulnerability. |
| **Assertions** | STD-006-CPP | This standard ensures that assertions are used appropriately to verify assumptions and preconditions in the code, especially in debugging or development environments. Assertions can catch bugs early by ensuring that certain conditions hold true during execution. However, they should not be used for error handling in production code, as they may be disabled in optimized builds. |
| **Exceptions** | STD-007-CPP | This standard ensures that exceptions are used appropriately for error handling. Using exceptions allows for graceful handling of unexpected conditions, preventing crashes and ensuring that errors are caught and addressed without corrupting the program state. It’s critical to distinguish between recoverable errors and programming bugs, handling the former via exceptions and not relying on them for control flow. |
| **Input Validation** | STD-008-CPP | Input must be validated to ensure that it meets expected formats and does not contain malicious data. |
| Buffer Overflow Prevention | STD-009-CP | This standard ensures that memory buffers are properly managed to avoid buffer overflows, which can lead to data corruption, application crashes, and security vulnerabilities. Using safe memory handling functions and controlling buffer sizes is critical to maintaining program integrity. |
| Secure File Handling | STD-010-CPP | This standard ensures that file handling operations, including reading and writing to files, are performed securely. Insecure file handling can lead to vulnerabilities such as file path manipulation, privilege escalation, and unauthorized access to sensitive files. |

**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 with 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 follow 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 the 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 Template | David Buksbaum |  |
| 1.1 | 12/12/2024 | Revised Template | Kevin Thompson |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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