**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 | This coding standard emphasizes the importance of validating and sanitizing user input data to prevent vulnerabilities such as injection attacks or buffer overflows. By verifying the integrity and format of incoming data, developers can ensure that it meets the expected criteria and is safe to process. |
| 1. Heed Compiler Warnings | This standard advises developers to pay attention to warnings generated by the compiler during the code compilation process. Compiler warnings can indicate potential issues or programming errors that might lead to bugs or vulnerabilities. By addressing these warnings, developers can improve the overall code quality and reduce the risk of introducing unexpected behavior. |
| 1. Architect and Design for Security Policies | This standard encourages developers to consider security requirements during the architectural and design phases of software development. By proactively integrating security measures into the system's design, developers can create a robust and secure foundation. This approach involves identifying potential security threats, defining security policies, and implementing appropriate security controls. |
| 1. Keep It Simple | The "Keep It Simple" standard promotes writing clean and straightforward code. Simplicity in coding minimizes the chances of introducing bugs or vulnerabilities due to overly complex logic. By favoring simplicity, developers can enhance code readability, maintainability, and reduce the overall complexity of the system. |
| 1. Default Deny | This standard advocates for adopting a security mindset where access is denied by default unless explicitly allowed. It involves implementing security measures that restrict access to sensitive resources or functionalities, ensuring that only authorized entities can access them. By employing the "Default Deny" principle, developers can reduce the attack surface and mitigate potential security risks. |
| 1. Adhere to the Principle of Least Privilege | This standard follows the principle of granting the minimum required privileges to perform a specific task. Developers should assign access rights and permissions based on The principle of least privilege, ensuring that each entity, whether it's a user or a system component, has only the necessary privileges to fulfill its intended function. This minimizes the potential impact of security breaches or unauthorized actions. |
| 1. Sanitize Data Sent to Other Systems | This coding standard emphasizes the importance of sanitizing data before transmitting it to external systems or interfaces. By cleaning and validating the data being sent, developers can prevent injection attacks, data corruption, or other security vulnerabilities that could be exploited by malicious actors. |
| 1. Practice Defense in Depth | Defense in Depth is a security strategy that involves layering multiple security measures to provide comprehensive protection. This standard encourages developers to incorporate multiple layers of security controls, such as firewalls, encryption, access controls, and intrusion detection systems, to defend against potential attacks. By implementing a multi-layered defense approach, developers can improve the overall security posture of the system. |
| 1. Use Effective Quality Assurance Techniques | This standard highlights the importance of employing effective quality assurance techniques throughout the software development lifecycle. It involves utilizing techniques like code reviews, automated testing, static analysis, and dynamic analysis to identify and eliminate coding errors, vulnerabilities, or performance issues. By prioritizing quality assurance, developers can ensure that the software meets the desired security and functionality standards. |
| 1. Adopt a Secure Coding Standard | This coding standard suggests following a set of established guidelines or best practices for secure coding. Secure coding standards provide a comprehensive set of rules and recommendations to address common vulnerabilities and ensure that developers follow secure coding practices consistently. By adopting a secure coding standard, developers can reduce the likelihood of introducing security flaws and enhance the overall security of the software. |

### 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] | The "Data Type" standard emphasizes the importance of using appropriate data types when declaring and manipulating variables in code. By selecting the correct data type for each variable, developers can ensure that the data is stored efficiently and accurately, and that the operations performed on the data are valid and optimized. |

| **Noncompliant Code** |
| --- |
| Assigning wrong variable type |
| #include <iostream>  #include <string>  int main() {  std::string num = "10"; // Assigning a string value to a variable intended for numerical operations  std::string result = num \* 2; // Performing multiplication with a string value    // Rest of the code...    return 0;  } |

| **Compliant Code** |
| --- |
| Assigning correct variable type |
| int main() {  int num = 10; // Assigning an integer value to a variable intended for numerical operations  int result = num \* 2; // Performing multiplication with an integer value    // Rest of the code...    return 0;  } |

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

| **Principles(s):** Principle of Data Integrity and Principle of Efficiency  These principles support the "Data Type" standard by guiding developers to use suitable data types that ensure data integrity and promote efficient code execution. By following these principles, codebases can benefit from accurate data representation, reduced risk of errors or data corruption, and improved performance through optimized resource utilization. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.21 | V560 | PVS-Studio is a static code analysis tool that can detect issues related to incorrect variable types. The V560 checker specifically looks for potential errors when assigning values of one type to variables of another type. |
| CLANG Tidy | 17.0.0 | bugprone | **clang-tidy** is a clang-based C++ “linter” tool. Its purpose is to provide an extensible framework for diagnosing and fixing typical programming errors, like style violations, interface misuse, or bugs that can be deduced via static analysis. **clang-tidy** is modular and provides a convenient interface for writing new checks. |
| Coverity | 2022.12 | INTEGER\_OVERFLOW | Coverity Scan is a free service for static code analysis of Open-Source projects. It is based on Coverity's commercial product and can analyze C, C++ and Java code. Coverity's static code analysis doesn't run the code. |
| SonarQube | 10.0 | S1067 | SonarQube is a popular static code analysis platform. The S1067 rule in SonarQube specifically focuses on identifying potential type-related issues, including incorrect assignments. |

**Coding Standard 2**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | The "Data Value" standard emphasizes the importance of using appropriate and secure data values in code. Noncompliant code may contain weak or easily guessable values, such as using simple passwords or hardcoding sensitive information. Compliant code follows best practices and ensures that data values are strong, secure, and appropriate for their intended use. |

| **Noncompliant Code** |
| --- |
| Weak Password |
| #include <string>  int main() {  std::string password = "password123"; // Using a weak and easily guessable password    // Rest of the code...    return 0;  } |

| **Compliant Code** |
| --- |
| Strong Password |
| #include <string>  int main() {  std::string password = "9^&$HG3l9k#@"; // Using a strong and secure password    // Rest of the code...    return 0;  } |

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

| **Principles(s):** Principle of Data Confidentiality and Principle of Least Privilege  By considering the principle of data confidentiality, the "Data Value" standard promotes the use of strong and secure data values to protect sensitive information. Additionally, the principle of least privilege emphasizes the importance of avoiding weak values and ensuring that only authorized entities have access to sensitive data. By adhering to these principles, codebases can enhance data security, protect against unauthorized access, and maintain the confidentiality of sensitive information. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OWASP Dependency Check | 3.2.0 | InsecureRandom | OWASP Dependency Check is a security scanner that can detect insecure random number generation functions and highlight weak data values, such as hard-coded passwords or sensitive information. |
| SonarQube | 10.0 | S2068 | SonarQube is a static code analysis platform that includes a rule called S2068, which detects hard-coded passwords. It can identify instances where sensitive data values are directly embedded in the code. |
| Fortify Static Code Analyzer | Latest | HardcodedCredential | Fortify Static Code Analyzer is a comprehensive tool for finding security vulnerabilities in code. The HardcodedCredential checker specifically targets hard-coded credentials and sensitive data values, helping to identify potential risks. |
| Bandit | Latest | B105 | Bandit is a security linter for Python code. The B105 check focuses on identifying hard-coded passwords and secret keys in the codebase, promoting the use of secure data values. |

#### 

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | The "String Correctness" standard focuses on ensuring the correctness and safety of string values used in code. Noncompliant code may include hard-coded strings without proper validation or sanitization, which can lead to security vulnerabilities or unexpected behavior. Compliant code follows best practices by using validated and sanitized string inputs, such as taking user input and applying appropriate sanitization methods to ensure the integrity and safety of the strings used in the code. |

| **Noncompliant Code** |
| --- |
| Didn’t Sanitize code |
| // Noncompliant code example  std::string message = "Hello, World!"; // Using hard-coded strings without proper validation or sanitization |

| **Compliant Code** |
| --- |
| Sanitized code |
| // Compliant code example  std::string userInput;  std::cout << "Enter a message: ";  std::getline(std::cin, userInput); // Taking user input and storing it in a variable  std::string message = sanitize(userInput); // Sanitizing the input before using it in the code |

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

| **Principles(s):**  Principle of Input Validation  Principle of Defense in Depth  By adhering to the principle of input validation, the "String Correctness" standard promotes the practice of validating user input to maintain data integrity and prevent unexpected behavior. Additionally, the principle of defense in depth reinforces the importance of implementing multiple layers of protection, such as input sanitization, to mitigate the risk of security vulnerabilities and maintain the correctness and safety of string values used in the code. |
| --- |
|  |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.0 | SonarC++ (C/C++ analyzer provided by SonarSource) | SonarQube with the SonarC++ analyzer can perform static code analysis and detect issues related to string correctness. It provides various built-in rules that can identify instances of hard-coded strings without proper validation or sanitization. |
| ESLint with C++ Plugin | Latest | C++ Plugin for ESLint | ESLint is a popular linting tool widely used for JavaScript code analysis. With the C++ plugin, ESLint can be extended to analyze C++ code as well. The C++ plugin provides rules that can help identify noncompliant string usage, including hard-coded strings without proper validation or sanitization. |
| Clang-Tidy | 17.0 | Clang-Tidy | Clang-Tidy is a powerful tool that performs static analysis on C++ code using the Clang compiler infrastructure. It offers various checks and diagnostics to identify potential issues and noncompliant code patterns. Clang-Tidy can be configured to include checks specific to string correctness, allowing you to detect instances of hard-coded strings without proper validation or sanitization in your codebase. |
| Cppcheck | Latest | Cppcheck | Cppcheck is an open-source static analysis tool for C++ code. It can detect a wide range of coding errors and potential vulnerabilities. Cppcheck includes checks that can help identify noncompliant string usage, such as hard-coded strings without proper validation or sanitization. It provides command-line interface options for integrating with build systems and automation workflows. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | The "SQL Injection" standard aims to prevent SQL injection vulnerabilities by properly handling user input when constructing SQL queries. Noncompliant code may use string concatenation to construct queries directly from user input, which can lead to SQL injection attacks. Compliant code follows best practices by using parameterized queries or prepared statements, which ensure that user input is properly sanitized and separated from the query structure, mitigating the risk of SQL injection vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Incorrect parameters |
| // Noncompliant code example  std::string username = getUserInput();  // Assuming getUserInput() retrieves user input without proper sanitization  std::string query = "SELECT \* FROM Users WHERE username = '" + username + "';";  // Constructing SQL query using string concatenation  executeQuery(query);  // Executing the SQL query without proper parameterization or prepared statements |

| **Compliant Code** |
| --- |
| Correct parametes |
| // Compliant code example  std::string username = getUserInput(); // Assuming getUserInput() retrieves user input without proper sanitization  // Using parameterized queries or prepared statements to prevent SQL injection  std::string query = "SELECT \* FROM Users WHERE username = ?";  executePreparedStatement(query, username); // Executing the prepared statement with the sanitized user input |

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

| **Principles(s):** Principle of Input Validation, Principle of Defense in Depth  The "SQL Injection" standard aligns with the Principle of Input Validation by emphasizing the need to properly validate and sanitize user input when constructing SQL queries. By using parameterized queries or prepared statements, the standard promotes the separation of user input from the query structure, ensuring that any malicious SQL code injected by an attacker is treated as data and not executed. This mitigates the risk of SQL injection vulnerabilities. Additionally, the standard aligns with the Principle of Defense in Depth by adding an extra layer of security to the application's data access layer, reducing the likelihood of successful SQL injection attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.0 | Rule - "Query parameters should be used in SQL statements" | SonarQube's C/C++ analyzer provides a rule to detect instances where string concatenation is used to construct SQL queries without proper parameterization. This rule identifies potential SQL injection vulnerabilities and encourages the use of parameterized queries or prepared statements. By integrating SonarQube into the CI/CD pipeline, the organization can automatically detect noncompliant code and enforce the SQL Injection coding standard. |
| Fortify Static Code Analyzer | 21.1 | SQL Injection vulnerability check | Fortify Static Code Analyzer is a comprehensive tool that includes checks specifically designed to detect SQL injection vulnerabilities. It analyzes the codebase, identifies potential injection points, and provides guidance on remediation. By incorporating Fortify into the development process, organizations can automatically scan code for SQL injection issues and ensure compliance with the SQL Injection coding standard |
| Checkmarx CxSAST | 9.0 | SQL Injection vulnerability detection | Checkmarx CxSAST is a powerful static application security testing (SAST) tool that includes dedicated checks for SQL injection vulnerabilities. It scans the codebase, identifies vulnerable code patterns, and provides actionable insights for developers. By integrating Checkmarx CxSAST into the CI/CD pipeline, organizations can automatically identify and mitigate SQL injection vulnerabilities, aligning with the SQL Injection coding standard. |
| Veracode Static Analysis | 20.4 | SQL Injection security flaw detection | Veracode Static Analysis is a leading application security testing tool that includes checks for SQL injection security flaws. It performs a comprehensive analysis of the codebase, identifies potential vulnerabilities, and provides developers with actionable recommendations. By incorporating Veracode Static Analysis into the development workflow, organizations can automatically detect and address SQL injection issues, ensuring compliance with the SQL Injection coding standard. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | The "Memory Protection" standard focuses on ensuring proper memory protection and avoiding buffer overflows or other memory-related vulnerabilities. Noncompliant code may use fixed-size buffers without proper boundary checks, which can lead to buffer overflows and memory corruption. Compliant code follows best practices by using dynamic memory allocation to ensure appropriate memory size based on input length, and performs proper memory management, such as deallocating the memory when it's no longer needed. This approach helps protect against memory-related vulnerabilities and ensures proper memory utilization. |

| **Noncompliant Code** |
| --- |
| Improper boundary checks |
| // Noncompliant code example  char buffer[100];  readInput(buffer, 200); // Reading input into a fixed-size buffer without proper boundary checks |

| **Compliant Code** |
| --- |
| Using proper buffers and boundary checks |
| // Compliant code example  std::string input = getUserInput(); // Assuming getUserInput() retrieves user input safely  std::string sanitizedInput = sanitizeInput(input); // Sanitizing the input  // Using dynamic memory allocation to ensure appropriate memory size based on input length  char\* buffer = new char[sanitizedInput.length() + 1];  std::strcpy(buffer, sanitizedInput.c\_str());  // Use the buffer...  // Don't forget to deallocate the memory when no longer needed  delete[] buffer; |

**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 | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| AddressSanitizer | Latest | AddressSanitizer | AddressSanitizer is a dynamic memory error detector tool provided by the LLVM project. It can automatically detect various memory-related vulnerabilities such as buffer overflows, use-after-free, and memory leaks. By integrating AddressSanitizer into the build and testing process, it can help identify and mitigate memory protection issues in the code, ensuring compliance with the Memory Protection coding standard. |
| Valgrind | Latest | Memcheck | Valgrind is a popular dynamic analysis framework that includes Memcheck, a tool for detecting memory-related errors such as memory leaks, buffer overflows, and uninitialized memory usage. By running the code through Valgrind's Memcheck, developers can automatically identify memory protection issues and ensure compliance with the Memory Protection coding standard. |
| Microsoft C/C++ Compiler (MSVC) | Latest | /analyze option | MSVC includes a static code analysis tool that can detect potential memory errors, including buffer overflows and use-after-free. By enabling the /analyze option during the compilation process, developers can automatically analyze the code for memory protection issues and ensure compliance with the Memory Protection coding standard. |
| Clang Static Analyzer | Latest | Memory Check | Clang Static Analyzer is a powerful static code analysis tool that can detect various issues, including memory-related vulnerabilities. It offers a range of checkers focused on memory safety, such as buffer overflows, null pointer dereferences, and resource leaks. By integrating Clang Static Analyzer into the build process, developers can automatically identify and address memory protection issues, ensuring compliance with the Memory Protection coding standard. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | The "Assertions" standard promotes the use of assertions to validate assumptions and ensure program correctness. Noncompliant code may rely on manual error handling and conditional checks to validate assumptions, which can be error-prone and cumbersome. Compliant code uses assertions to explicitly state assumptions and halt program execution if those assumptions are violated. This helps catch programming errors early in the development process and ensures that the program operates with correct data and assumptions. Note that assertions are typically used during development and testing and may be disabled in production for performance reasons. |

| **Noncompliant Code** |
| --- |
| No assertion to validate positive number |
| // Noncompliant code example  int x = getPositiveNumber(); // Assuming getPositiveNumber() returns a positive number  if (x <= 0) {  // Handle the error...  } |

| **Compliant Code** |
| --- |
| With assertion to properly validate if the number is positve |
| // Compliant code example  int x = getPositiveNumber(); // Assuming getPositiveNumber() returns a positive number  assert(x > 0); // Assertion to validate that x is positive  // Continue with code execution... |

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

| **Principles(s):** Principle of Defensive Programming |
| --- |
| The "Assertions" standard aligns with the Principle of Defensive Programming by advocating for the use of assertions to validate assumptions and ensure program correctness. By incorporating assertions into the code, developers explicitly state their assumptions about the program's state and data. If those assumptions are violated during runtime, the program halts, providing an opportunity to catch programming errors early. This proactive approach helps improve the overall quality and reliability of the codebase. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.0 | Assertions should be used to validate assumptions | SonarQube's C/C++ analyzer includes a rule that detects instances where assertions are not used to validate assumptions. This rule helps identify noncompliant code that relies on manual error handling instead of assertions. By integrating SonarQube into the CI/CD pipeline, organizations can automatically detect such code and enforce the "Assertions" coding standard. |
| Cppcheck | 2.6 | Possible null pointer dereferences | Cppcheck is a static code analysis tool that checks for potential null pointer dereference. While not specifically tailored to assertions, this rule can help identify scenarios where assertions could be used to validate assumptions involving pointer values. By integrating Cppcheck into the development process, organizations can automatically detect code that could benefit from the use of assertions. |
| PVS Studio | 7.14 | V638 | PVS-Studio is a static analysis tool that includes checks for potential infinite loops. Although not directly focused on assertions, this rule can identify situations where assertions can be used to validate loop conditions and prevent infinite iterations. By integrating PVS-Studio into the development workflow, organizations can automatically identify code patterns where assertions can enhance program correctness. |
| CodeSonar | 5.4 | Proper Use of assertions | CodeSonar is a static analysis tool that includes a checker for ensuring the proper use of assertions. It can detect cases where assertions are missing or not used effectively to validate assumptions. By incorporating CodeSonar into the development process, organizations can automatically analyze the codebase for compliance with the "Assertions" coding standard and ensure the correct usage of assertions. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | The "Exceptions" standard promotes the use of exceptions for error handling and exceptional situations. Noncompliant code may rely on error codes, flags, or manual error handling to handle exceptional conditions, which can lead to code duplication and make error handling more complex. Compliant code uses exceptions to explicitly indicate exceptional conditions and handle them at an appropriate level in the code hierarchy. This allows for cleaner and more maintainable code by separating the normal control flow from error handling and providing a consistent mechanism for propagating and handling errors. |

| **Noncompliant Code** |
| --- |
| No use of exceptions |
| // Noncompliant code example  int divide(int dividend, int divisor) {  if (divisor == 0) {  // Handle the error condition...  }  return dividend / divisor;  } |

| **Compliant Code** |
| --- |
| Proper use of exceptions |
| // Compliant code example  int divide(int dividend, int divisor) {  if (divisor == 0) {  throw std::runtime\_error("Divide by zero error"); // Throwing an exception for the error condition  }  return dividend / divisor;  } |

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

| **Principles(s):** Principle of Error Handling  The "Exceptions" standard aligns with the Principle of Error Handling by promoting the use of exceptions to handle exceptional conditions. Exceptions provide a structured and consistent approach to handle errors, allowing for clean separation between normal control flow and error handling logic. By using exceptions, developers can provide meaningful error messages, propagate errors to appropriate levels in the code hierarchy, and ensure consistent error handling practices throughout the codebase. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.0 | Exceptions should be used for exceptional conditions | SonarQube's C/C++ analyzer includes a rule that detects instances where exceptions are not used to handle exceptional conditions. This rule helps identify noncompliant code that relies on error codes or manual error handling instead of exceptions. By integrating SonarQube into the CI/CD pipeline, organizations can automatically detect such code and enforce the "Exceptions" coding standard. |
| Cppcheck | 2.6 | Exception thrown by functions is not caught or documented | Cppcheck is a static code analysis tool that includes checks for uncaught exceptions. This rule can help identify instances where exceptions are thrown but not properly caught or documented. By integrating Cppcheck into the development process, organizations can automatically detect code that does not conform to the "Exceptions" coding standard and ensure proper exception handling. |
| PVS-Studio | 7.14 | V549 | The 'foo' function throws an exception and then another exception is thrown in the destructor of the object." PVS-Studio is a static analysis tool that includes checks for potential issues related to exception handling. This rule can help identify cases where exceptions thrown in a function conflict with exceptions thrown in the destructor of an object, potentially leading to unexpected behavior. By incorporating PVS-Studio into the development workflow, organizations can automatically identify code patterns that violate the "Exceptions" coding standard. |
| CodeSonar | 5.4 | Exception Handling | CodeSonar is a static analysis tool that includes a checker for analyzing exception handling. It can detect violations of exception handling best practices, such as not catching exceptions, catching exceptions by value instead of reference, or rethrowing exceptions without proper documentation. By incorporating CodeSonar into the development process, organizations can automatically analyze the codebase for compliance with the "Exceptions" coding standard and ensure proper exception handling. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Validation | [STD-008-CPP] | The "Input Validation" standard emphasizes the importance of validating and sanitizing input data to prevent security vulnerabilities and unexpected behavior. Noncompliant code may assume that user input is valid without performing any validation or sanitization. Compliant code incorporates input validation by applying appropriate validation techniques (e.g., checking input length, format, and range) and sanitizing the input to remove any potentially harmful or unwanted characters. This helps ensure that the data being processed is within expected boundaries and reduces the risk of security vulnerabilities or errors caused by invalid input. |

| **Noncompliant Code** |
| --- |
| No user input validation |
| // Noncompliant code example  std::string userInput = getUserInput(); // Assuming getUserInput() retrieves user input without validation  processData(userInput); // Processing the user input without validation |

| **Compliant Code** |
| --- |
| Proper user Input validation |
| // Compliant code example  std::string userInput = getUserInput(); // Assuming getUserInput() retrieves user input without validation  // Validate and sanitize the user input before processing  if (isValidInput(userInput)) {  std::string sanitizedInput = sanitizeInput(userInput);  processData(sanitizedInput);  } else {  // Handle invalid input...  } |

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

| **Principles(s):** Principle of Input Validation, Principle of Least Privilege  The "Input Validation" standard aligns with the Principle of Input Validation by emphasizing the need to validate and sanitize user input to prevent security vulnerabilities. By validating input length, format, and range, and sanitizing input to remove potentially harmful characters, the standard ensures that the data being processed is within expected boundaries. This reduces the risk of injection attacks, buffer overflows, and other security vulnerabilities arising from invalid input. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.0 | Input Validation should be applied to user-supplied data | SonarQube's C/C++ analyzer provides a rule to detect instances where user input is not properly validated. This rule identifies potential security vulnerabilities and encourages the use of input validation techniques. By integrating SonarQube into the CI/CD pipeline, the organization can automatically detect noncompliant code and enforce the Input Validation coding standard. |
| Coverity | 2022.12 | INCOMPLETE\_TO\_INVALID | Coverity is a static analysis tool that offers a wide range of checkers for identifying coding issues. The "INCOMPLETE\_TO\_INVALID" checker helps detect cases where incomplete input validation may lead to security vulnerabilities. By leveraging Coverity, the organization can enhance code security and adherence to the Input Validation coding standard. |
| Fortify | 21.1 | Input Validation | Fortify is a comprehensive application security tool that includes rules specifically designed for input validation. The "Input Validation" rule helps identify potential weaknesses in input handling and provides guidance on proper validation techniques. Integrating Fortify into the development process enables automated detection of noncompliant code and ensures adherence to the Input Validation coding standard. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Error Handling and Logging | [STD-009-CPP] | The "Error Handling and Logging" standard emphasizes proper error handling and logging practices. Noncompliant code may rely on conditional checks or return statements to handle errors, which can make error handling inconsistent and may not provide enough information for debugging or auditing purposes. Compliant code uses exceptions to explicitly indicate error conditions and throws an exception when an error is encountered. This allows for centralized error handling, cleaner code structure, and the ability to include meaningful error messages. Additionally, proper logging is essential to capture relevant information about errors, warnings, and critical events to aid in troubleshooting and auditing. |

| **Noncompliant Code** |
| --- |
| No error handling |
| // Noncompliant code example  void processData(int value) {  if (value < 0) {  // Error: Invalid value  return;  }  // Process the value...  } |

| **Compliant Code** |
| --- |
| Proper error handling |
| // Compliant code example  void processData(int value) {  if (value < 0) {  throw std::runtime\_error("Invalid value"); // Throwing an exception to indicate the error condition  }  // Process the value...  } |

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

| **Principles(s):** Principle of Fail Fast, Principle of Least Astonishment  The "Error Handling and Logging" standard aligns with the Principle of Fail Fast by promoting the use of exceptions to explicitly indicate error conditions. By throwing an exception when an error is encountered, the code immediately terminates the normal execution flow, allowing for centralized error handling and preventing the propagation of incorrect data. Additionally, the standard aligns with the Principle of Least Astonishment by encouraging proper error messages and consistent error handling practices, ensuring that developers and users are not surprised by unexpected behavior and can easily understand and address error conditions. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Medium | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.0 | Exceptions should not be ignored | SonarQube's C/C++ analyzer provides a rule that detects instances where exceptions are ignored or not properly handled. This rule helps enforce the Error Handling and Logging coding standard by identifying code that lacks proper error handling using exceptions. By integrating SonarQube into the CI/CD pipeline, the organization can automatically detect noncompliant code and ensure adherence to the standard. |
| Coverity | 2022.12 | EXCEPTIONS | Coverity offers a checker specifically designed to detect issues related to exception handling. The "EXCEPTIONS" checker identifies potential errors and weaknesses in exception handling, allowing developers to enhance error handling practices and comply with the Error Handling and Logging coding standard. Integrating Coverity into the development process enables automated detection of noncompliant code and helps improve code quality and reliability. |
| Parasoft C/C++ | 2021.1 | Static Analysis-ERR09-C | Parasoft C/C++test includes a static analysis rule, "ERR09-C," which focuses on proper error handling using exceptions. This rule detects instances where exceptions are not handled or are handled inadequately, providing guidance and suggestions for improvement. By utilizing Parasoft C/C++test, the organization can automate the detection of noncompliant code and enforce the Error Handling and Logging coding standard. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Code Documentation | [STD-010-CPP] | The "Code Documentation" standard emphasizes the importance of documenting code to improve maintainability and readability. Noncompliant code may lack proper comments and documentation, making it difficult for other developers (including the original author) to understand the purpose, behavior, and usage of the code. Compliant code includes descriptive comments and documentation that provide information about the purpose of functions, the meaning of parameters, and any assumptions or constraints associated with the code. This promotes easier comprehension, collaboration, and maintenance of the codebase, reducing the risk of misinterpretation or misunderstanding. |

| **Noncompliant Code** |
| --- |
| No code documentation or in-line comments |
| 0-int calculateSum(int a, int b) {    return a + b;  } |

| **Compliant Code** |
| --- |
| Proper code documentation/ in-line comments |
| // Compliant code example  /\*\*  \* Calculates the sum of two integers.  \*  \* @param a the first integer.  \* @param b the second integer.  \* @return the sum of the two integers.  \*/  int calculateSum(int a, int b) {  return a + b;  } |

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

| **Principles(s):** Principle of Self-Documenting Code, Principle of Readability  The "Code Documentation" standard aligns with the Principle of Self-Documenting Code by emphasizing the use of comments and documentation to make the code self-explanatory. By providing descriptive comments that describe the purpose, behavior, and usage of the code, the standard ensures that other developers can easily understand the code without requiring additional clarification. Additionally, the standard aligns with the Principle of Readability by promoting clear and concise documentation, enhancing the code's readability and reducing the time and effort required for maintenance and collaboration. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Doxygen | 1.91 | DocumentationCheck | Doxygen is a widely used documentation generation tool that can automatically extract code documentation from source files and generate documentation in various formats (e.g., HTML, PDF). By following specific documentation comment styles and configuring Doxygen to process the codebase, it can automatically detect missing or incomplete documentation, aiding in enforcing the Code Documentation standard. Integrating Doxygen into the development process enables automated generation and validation of code documentation. |
| Cppcheck | 2.5 | missingInclude | Cppcheck is a static analysis tool for C and C++ code that can detect various coding errors and provide recommendations for improvement. Although it is not specifically designed for code documentation, it includes a "missingInclude" style checker that can identify missing comments or documentation for functions, helping to enforce the Code Documentation standard. By integrating Cppcheck into the build process, the organization can automate the detection of noncompliant code documentation and improve code maintainability. |

### 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



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.

To automate the enforcement of the coding standards defined in this policy, Green Pace can integrate SonarQube into their existing DevOps process. SonarQube can be added as a step in the CI/CD pipeline to perform static code analysis and automatically detect issues related to data types, data values, and string correctness. By running SonarQube as part of the code review process, developers can receive immediate feedback on noncompliant code, allowing them to address the issues early in the development lifecycle. SonarQube can be integrated with popular CI/CD tools such as Jenkins, Azure DevOps, or GitLab CI/CD, enabling seamless automation of code analysis and compliance checking. This integration helps ensure that the coding standards are consistently enforced and promotes the principle of automation in the DevSecOps pipeline.

It's important to configure SonarQube with the specific rules/checks mentioned for each coding standard to align with the organization's requirements and policies. Additionally, SonarQube provides flexibility in customizing the rule sets and allows for the creation of custom rules to address specific coding standards or security guidelines unique to Green Pace.

### 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 | Likely | High | High | 1 |
| STD-003-CPP | High | Likely | Medium | High | 2 |
| STD-004-CPP | High | Likely | Low | High | 3 |
| STD-005-CPP | High | Likely | Medium | High | 1 |
| STD-006-CPP | Medium | Medium | Low | Medium | 3 |
| STD-007-CPP | Medium | Medium | Low | Medium | 3 |
| STD-008-CPP | Medium | Likely | Medium | Medium | 2 |
| STD-009-CPP | Medium | Unlikely | Medium | High | 2 |
| STD-010-CPP | Low | Likely | Low | Medium | 3 |

Include all three types of encryptions (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 | **Policy**: All sensitive data stored in databases, file systems, or any other storage medium must be encrypted at rest.  **Explanation**: Encryption at rest involves encrypting data when it is stored or saved in storage systems. This policy applies to all sensitive data to prevent unauthorized access or data breaches in case the storage media are compromised physically or electronically. Implementation: Organizations should implement encryption mechanisms such as disk encryption, database encryption, or file-level encryption to protect sensitive data stored on servers, databases, or other storage devices. |
| Encryption at flight | **Policy**: All data transmitted over networks, including internal and external networks, must be encrypted in transit.  **Explanation**: Encryption in flight ensures that data is protected while being transmitted between systems or over networks. This policy applies to all communication channels, including internal networks, internet connections, and cloud-based services, to prevent eavesdropping or data interception during transmission. Implementation: Organizations should implement secure communication protocols such as SSL/TLS (HTTPS) for web traffic, IPsec for virtual private networks (VPNs), or S/MIME for email encryption to secure data in transit. |
| Encryption in use | **Policy**: Encryption must be implemented for sensitive data when it is being processed or used by applications or systems.  **Explanation**: Encryption in use protects sensitive data while it is being processed, accessed, or used by applications or systems. This policy applies to data that is temporarily decrypted for processing or computation purposes. Implementation: Organizations should use encryption techniques such as homomorphic encryption, secure enclaves, or trusted execution environments to protect sensitive data during processing or computation. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | **Policy**: All users must authenticate themselves using unique credentials before accessing any system or resource.  **Explanation**: Authentication verifies the identity of users and ensures that only authorized individuals can access the system or resources. This policy applies to all users, including employees, contractors, and external users, to prevent unauthorized access and protect sensitive information. Implementation: Organizations should implement strong authentication mechanisms such as passwords, multi-factor authentication (MFA), biometrics, or smart cards to authenticate users. |
| Authorization | **Policy**: User access privileges must be defined and enforced based on the principle of least privilege.  **Explanation**: Authorization determines the level of access and actions that users are allowed to perform once authenticated. This policy applies to all users and resources to ensure that each user has the appropriate level of access based on their roles and responsibilities. Implementation: Organizations should implement access control mechanisms, role-based access control (RBAC), or attribute-based access control (ABAC) to enforce proper authorization and limit user privileges. |
| Accounting | **Policy**: All user activities and system actions must be logged and audited for accountability and security monitoring purposes.  **Explanation**: Accounting involves capturing and recording user activities, system events, and resource usage for auditing, tracking, and investigating purposes. This policy applies to all systems, applications, and resources to maintain an audit trail and enable security incident response and compliance monitoring. Implementation: Organizations should implement logging and auditing mechanisms that capture user logins, database changes, user access permissions, file accesses, and other relevant activities. |

**\***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.

**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 Template | David Buksbaum |  |
| 1.1 | 5/28/2023 | Update | Tyler Morgan |  |
| 1.2 | 6/10/2023 | Update | Tyler Morgan |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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