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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

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

[Project One 15](#_Toc52464070)

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

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

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

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

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

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

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

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validating input data involves inspecting and verifying the correctness and integrity of data provided from sources outside of the program or system. The process usually includes checking data type, boundary conditions, SQL injections, and buffer overflow attacks. |
| 1. Heed Compiler Warnings | Compiler warnings are messages generated by the compiler when it encounters potentially problematic or non-standard code constructs during the compilation process. Ignoring these warnings can lead to subtle bugs, compatibility issues, or even security vulnerabilities in the resulting software. |
| 1. Architect and Design for Security Policies | By making security policies at the architectural level, you ensure that security is systematically integrated into the software’s structure and functionality. Being proactive about this helps to minimize vulnerabilities and ultimately reinforces the software’s overall security. |
| 1. Keep It Simple | Keeping it simple encourages developers to avoid unnecessary abstractions and overengineering. Instead, it promotes clean, maintainable, and reliable code. |
| 1. Default Deny | By default, access to resources or actions should be denied unless explicitly allowed. It is a proactive security measure that assumes any action or access is a potential security risk. This helps prevent things like unauthorized access and data breaches. |
| 1. Adhere to the Principle of Least Privilege | By default, users are given the minimum amount of access necessary to do their job. This is to help minimize the potential damage that can result from accidental or deliberate misuse of privileges. |
| 1. Sanitize Data Sent to Other Systems | Sanitization usually involves carefully validating and cleaning data before transmitting it to external systems, services, or APIs. Sanitizing data sent to other systems helps prevent security vulnerabilities, data breaches, and potential issues with interoperability. |
| 1. Practice Defense in Depth | Defense in depth is a strategy that involves the implementation of multiple layers of security controls and measures to protect a system or network. This approach aims to provide redundancy and resilience against security threats and vulnerabilities by ensuring that if one security layer fails or is breached, there are additional layers to safeguard the system. |
| 1. Use Effective Quality Assurance Techniques | Using effective Quality Assurance techniques helps identify and resolve issues early in the development process, reducing the cost and impact of defects in later stages. It also enhances the overall quality, reliability, and user satisfaction of the software product. |
| 1. Adopt a Secure Coding Standard | This standard focuses on writing code with security considerations in mind from the very beginning of the development process. By adopting security from the beginning, teams can significantly reduce the risk of security breaches, protect sensitive data, and build software that is resilient to a wide range of security threats and attacks. |

### 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-GEN | Data Type Coding Standards aim to ensure consistent and appropriate usage of data types throughout the codebase. Defining guidelines for choosing data types and naming conventions helps maintain code readability, portability, and reliability, and help avoid unnecessary type conversions. |

| **Noncompliant Code** |
| --- |
| In this example, int and float data types are mixed without proper type conversion, potentially leading to unexpected behavior or precision loss. |
| int x = 5;  float y = 3.14;  double result = x + y; |

| **Compliant Code** |
| --- |
| To fix the noncompliant code example, here we explicitly casts variables to the desired data type, avoiding implicit type conversions and ensuring proper data type usage. |
| int x = 5;  float y = 3.14f;  double result = static\_cast<double>(x) + static\_cast<double>(y); |

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

| **Principles(s):** DCL-031-CPP: Declarations and Initializing. This rule covers declaring your variables which involves deciding what data type would best represent the variable’s purpose. Specifically I think the sub rule “Declare identifiers before using them” ties in nicely to picking and using data types for your variables. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | Type-specifier  Function-return-type  Implicit-function-declaration  Undeclared-parameter | Fully checked |
| TrustInSoft Analyzer | 1.38 | Type specifier missing | Partially verified (exhaustively detects undefined behavior) |
| RuleChecker | 23.04 | Type-specifier  Function-return-type  Implicit-function-declaration  Undeclared-parameter | Fully checked |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-GEN | Data Value Coding Standards are designed to ensure that data values are represented, initialized, and used consistently and safely throughout the codebase. They define guidelines for data value assignments, literals, and initialization. |

| **Noncompliant Code** |
| --- |
| In this example, a character array is manually terminated with a null character, which can be error-prone and lead to buffer overflows if not done correctly. |
| char name[20];  name[0] = 'J';  name[1] = 'o';  name[2] = 'h';  name[3] = 'n';  name[4] = '\0'; |

| **Compliant Code** |
| --- |
| In this code snippet, the std::strcpy function from the C++ Standard Library is used for string initialization, which is a safer and more reliable way to handle strings. |
| char name[20];  std::strcpy(name, "John"); |

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

| **Principles(s):** DCL-031-CPP: Declarations and Initializing. In order to keep an eye on your data values, you need to make sure they are declared and initialized properly and according to your design. Specifically I think the sub rule “Declare identifiers before using them” ties in nicely to picking and using data values for your variables. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | Cert-DCL31 | Fully implemented |
| Clang | 3.9 | -Wimplicit-int |  |
| Coverity | 2017.07 | MISRA C 2012 Rule 8.1 | Implemented |
| Parasoft C/C++ test | 2023.1 | CERT\_C-DCL31-a | All functions shall be declared before use |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-GEN | String Correctness Coding Standards are to ensure that string-related operations are performed correctly and safely within the codebase. They aim to define guidelines for string handling, memory allocation, and input validation. |

| **Noncompliant Code** |
| --- |
| This code reads user input without validating it or handling potential input errors, leaving the program vulnerable to unexpected input. |
| std::string userInput;  std::cin >> userInput; |

| **Compliant Code** |
| --- |
| To fix the noncompliant code, the user input is instead read and validated using std::getline, which allows for safer handling of user input, including error checking and validation. |
| std::string userInput;  std::cout << "Enter your name: ";  std::getline(std::cin, userInput); |

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

| **Principles(s):** STR-031-CPP: Characters and Strings. This rule is all about how to handle characters and strings, which fits perfectly into the String Correctness standard. Specifically, the sub rule “Guarantee that storage for strings has sufficient space for character data and the null terminator” maps nicely to this standard. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | LANG.MEM.BO | Buffer overrun  Type overrun  No space for null terminator  A collection of warning classes that report uses of library functions prone to internal buffer overflows |
| Polyspace Bug Finder | R2023b | CERT C: Rule STR31-C | Checks for:   * Use of dangerous standard function * Missing null in string array * Buffer overflow from incorrect string format specifier * Destination buffer overflow in string manipulation * Insufficient destination buffer size   Rule partially covered. |
| TrustInSoft | 1.38 | Mem\_access | Exhaustively verified |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-GEN | SQL Injection Coding Standards are to prevent vulnerabilities related to SQL injection attacks in database interactions within the codebase. They define guidelines for safe SQL query construction, parameterization, and input validation. |

| **Noncompliant Code** |
| --- |
| In this code, user input is directly concatenated into a SQL query string, making the code vulnerable to SQL injection attacks. An attacker could manipulate the userInput variable to execute arbitrary SQL commands. |
| sqlite3\* db;  std::string userInput = "'; DROP TABLE users --";  std::string query = "SELECT \* FROM users WHERE username = '" + userInput + "';";  int rc = sqlite3\_exec(db, query.c\_str(), callback, 0, 0); |

| **Compliant Code** |
| --- |
| In this example, we fixed the vulnerability by parameterizing the SQL query using placeholders (?) and prepared statements to safely insert user input. This approach mitigates SQL injection vulnerabilities. |
| sqlite3\* db;  std::string userInput = "'; DROP TABLE users --";  std::string query = "SELECT \* FROM users WHERE username = ?";  sqlite3\_stmt\* stmt;  int rc = sqlite3\_prepare\_v2(db, query.c\_str(), -1, &stmt, 0);  if (rc == SQLITE\_OK) {  sqlite3\_bind\_text(stmt, 1, userInput.c\_str(), -1, SQLITE\_STATIC);  rc = sqlite3\_step(stmt);  sqlite3\_finalize(stmt);  } |

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

| **Principles(s):**  FIO-030-CPP: Input Output. This rule is all about how to read in from different sources and closing files when they are not currently in use. Specifically, the sub rule “Exclude user input from format strings” maps nicely to watching out for code injections. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | 1.4 | CERT C: Rule FIO30-C | Checks for tainted string format(rule partially covered) |
| Coverity | 2017.07 | TAINTED\_STRING | Implemented |
| CodeSonar | 7.4p0 | IO.INJ.FMT  MISC.FMT | Format string injection  Format string |
| Axivion Bauhaus Suite | 7.2.0 | CertC0-FIO30 | Partially implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-GEN | Memory Protection Coding Standards are established to prevent memory-related vulnerabilities, such as buffer overflows and memory leaks, within the codebase. They define guidelines for safe memory allocation, deallocation, and usage. |

| **Noncompliant Code** |
| --- |
| In this example, a buffer overflow occurs when attempting to access an element beyond the array's bounds. Additionally, memory allocated with new is not deallocated, leading to a memory leak. |
| int main() {  int\* arr = new int[1000];  arr[1001] = 42;  delete[] arr;  return 0;  } |

| **Compliant Code** |
| --- |
| This correct code example initializes the array with default values, accesses the array within its bounds, and deallocates memory properly, avoiding buffer overflows and memory leaks. |
| int main() {  int\* arr = new int[1000]();  arr[999] = 42;  delete[] arr;  return 0;  } |

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

| **Principles(s):** MEM-030-CPP: Input Output. This rule deals with how to handle memory allocation and clean-up properly. All these things map directly to memory protection. Specifically, the sub rule “Do not access freed memory” is a simple but direct rule for protecting memory. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | Dangling\_pointer\_use | Supported |
| CodeSonar | 7.4p0 | ALLOC.UAF | Use after free |
| LDRA tool suite | 9.7.1 | 51 D, 484 S, 112 D | Partially implemented |
| Parasoft C/C++ test | 2023.1 | CERT\_C-MEM30-a | Do not use resources that have been freed |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-GEN | Assertions Coding Standards are to improve code reliability and debugging by incorporating assertions to validate assumptions and detect unexpected conditions within the codebase. They aim to enhance code quality, identify issues early in development, and facilitate debugging. |

| **Noncompliant Code** |
| --- |
| This noncompliant code performs division without validating the denominator, resulting in a division by zero error. There are no assertions to catch this issue. |
| int divide(int numerator, int denominator) {  return numerator / denominator;  }  int main() {  int result = divide(10, 0);  std::cout << "Result: " << result << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| The compliant code includes an assertion to validate the denominator, ensuring that division by zero is detected early during development. |
| int divide(int numerator, int denominator) {  assert(denominator != 0);  return numerator / denominator;  }  int main() {  int result = divide(10, 0);  std::cout << "Result: " << result << std::endl;  return 0;  } |

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

| **Principles(s):** ERR-033-CPP: Err Handling. Assertions are used for catching unexpected conditions and behavior within a program, which is everything this rule covers. Specifically, the sub rule “Detect and handle standard library error” maps perfectly with this standard. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | Error-information-unused  Error-information-unused-computed | Partially checked |
| LDRA | 9.7.1 | 80 D | Partially implemented |
| PC-lint Plus | 1.4 | 534 | Partially supported |
| RuleChecker | 23.04 | Error-information-unused | Partially checked |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-GEN | Exceptions Handling Coding Standards are to promote robust error handling and recovery mechanisms within the codebase. They aim to define guidelines for consistent and effective use of exceptions. |

| **Noncompliant Code** |
| --- |
| In this example, error handling is performed by printing an error message to standard error, but there is no use of exceptions for structured error handling. This can lead to inconsistent error reporting and limited error recovery. |
| void divide(int numerator, int denominator) {  if (denominator == 0) {  std::cerr << "Division by zero!" << std::endl;  return;  }  int result = numerator / denominator;  std::cout << "Result: " << result << std::endl;  }  int main() {  divide(10, 0);  return 0;  } |

| **Compliant Code** |
| --- |
| Here we properly use exceptions (in this case, std::runtime\_error) for structured error handling. When a division by zero error occurs, it throws an exception with a meaningful error message, allowing for consistent error reporting and potential error recovery. |
| int divide(int numerator, int denominator) {  if (denominator == 0) {  throw std::runtime\_error("Division by zero!");  }  return numerator / denominator;  }  int main() {  try {  int result = divide(10, 0);  std::cout << "Result: " << result << std::endl;  } catch (const std::exception& ex) {  std::cerr << "Error: " << ex.what() << std::endl;  }  return 0;  } |

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

| **Principles(s):** ERR-033-CPP: Exception Handling. Exceptions are used for catching unexpected conditions and behavior within a program, which is everything this rule covers. Specifically, the sub rule “Detect and handle standard library error” maps perfectly with this standard. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | LANG.FUNCS.IRV  LANG.ERRCODE.NOTEST  LANG.ERRCODE.NZ | Ignored return value  Missing test of error code  Non-zero error code |
| Coverity | 2017.07 | MISRA C 2012 Rule 22.8  MISRA C 2012 Rule 22.9  MISRA C 2012 Rule 22.10 | Implemented |
| LDRA tool suite | 9.7.1 | 80 D | Partially implemented |
| PC-lint Plus | 1.4 | 534 | Partially supported |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Formatting and Indentation | STD-008-GEN | Formatting and Indentation Coding Standards are to ensure code readability, consistency, and maintainability by defining guidelines for code formatting, indentation, and whitespace usage. Proper formatting helps developers understand and navigate the codebase more effectively, reducing the likelihood of errors and making the code more accessible to team members. |

| **Noncompliant Code** |
| --- |
| Even though this code is valid, there is no consistent formatting, and the code lacks proper indentation, making it difficult to read and maintain. |
| int main(){int x=5;std::cout<<"Value of x is:"<<x<<std::endl;return 0;} |

| **Compliant Code** |
| --- |
| Here is the same code but with a consistent formatting style with proper indentation and spaces between operators and operands, making the code more readable and maintainable. |
| int main() {  int x = 5;  std::cout << "Value of x is: " << x << std::endl;  return 0;  } |

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

| **Principles(s):** MSC-012-CPP: Miscellaneous. This recommendation covers everything from semicolon placement to using comments consistently and in a readable fashion. Specifically, the sub rule “Detect and Remove code that has no effect or is never executed”, although it is more inline with optimizing your code, fits with formatting in that it aims for the readability and maintainability of your code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| GCC | 3.0 | -Wunused-value  -Wunused-parameter | Options detect unused local variables, nonconstant static variables and unused function parameters, or unreachable code respectively |
| ÉCLAIR | 1.2 | CC2.MSC12 | Partially implemented |
| Splint | 3.1.1 | -standard | The default mode checks for unreachable code |
| Polyspace Bug Finder | R2023b | CERT C: Rec. MSC12-C | Checks for |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Commenting and Documentation | STD-009-GEN | Commenting and Documentation Coding Standards define guidelines for adding comments and documentation to the code. Well-documented code aids developers in understanding code functionality, usage, and rationale, facilitating code reviews and future maintenance. |

| **Noncompliant Code** |
| --- |
| In this code, there are minimal comments, and the existing comments are not informative. The code lacks documentation, making it difficult for other developers to understand the purpose of the functions and how to use them. |
| int add(int a, int b) {  // Function to add two numbers  return a + b;  }  int main() {  int result = add(5, 3); // Call the add function  std::cout << "Result: " << result << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| This fixed code example includes informative comments. These comments provide details about the function's purpose, parameters, and return value, enhancing code understanding and maintainability. |
| /\*\*  brief Function to add two numbers.  a is the first number.  b is the second number.  Returns the sum of a and b.  \*/  int add(int a, int b) {  return a + b;  }  int main() {  int result = add(5, 3); // Call the add function to add 5 and 3  std::cout << "Result: " << result << std::endl;  return 0;  } |

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

| **Principles(s):** MSC-004-CPP: Miscellaneous. This recommendation covers everything from semicolon placement to using comments consistently and in a readable fashion, especially the sub rule “Use comments consistently and in a readable fashion”. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Medium | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ÉCLAIR | 1.2 | CC2.MSC04 | Fully implemented |
| PC-lint Plus | 1.4 | 1, 427, 602, 689, 853, 9059, 9060, 9066, 9259 | Fully supported |
| Polyspace Bug Finder | R2023b | CERT C: Rec. MSC04-C | Checks for use of /\* and // within a comment (rule partially covered) |
| RuleChecker | 23.04 | Mmline-comment  Sline-comment  Sline-spicing  Smline-comment | Partially checked |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Version Control | STD-010-GEN | Version Control Coding Standards are established to ensure efficient and effective use of version control systems in software development projects. It defines guidelines for consistent version control practices, commit messages, branching strategies, and code review workflows. Proper version control helps track changes, collaborate with team members, and maintain code quality. |

| **Noncompliant Code** |
| --- |
| This Git branching strategy does not follow a clear naming convention or provide context for the branches. It can lead to confusion and challenges in tracking feature development. |
| # Noncompliant Git branching strategy  feature-1/  feature-2/ |

| **Compliant Code** |
| --- |
| This compliant Git branching strategy uses a clear and organized naming convention, providing context for each branch's purpose. It helps streamline feature development and maintain a clean branch hierarchy. |
| # Compliant Git branching strategy  main/  develop/  feature/feature-1/  feature/feature-2/ |

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

| **Principles(s):** MSC-000 & 001-CPP: Miscellaneous. The best fit I could find for this standard is recommendation 49. This recommendation covers everything from semicolon placement to using comments consistently and in a readable fashion. Specifically, I think the sub rules “Compile cleanly at high warning levels” and “Strive for logical completeness” can be applied to this standard of version control. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | P8 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | BUILD.WALL | Not all warnings are enabled |
| PVS-Studio | 7.26 | V665 | [Insert text.] |
| SonarQube C/C++ Plugin | 3.11 | S1762  S973 | Warns when the default warning specifier is used with #pragma warning  Requires documentation of #pragma uses |

### Defense-in-Depth Illustration

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



## Project One

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

### Revise the C/C++ Standards

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

### Risk Assessment

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

### Automated Detection

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

### Automation

Provide a written explanation using the image provided.



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

Green Pace will enhance its existing DevOps process by integrating automation tools to enforce security standards. Static code analysis tools like SonarQube will scan code repositories for vulnerabilities and coding standards violations. Automated testing with tools like OWASP ZAP and Nessus will be integrated into the pipeline to ensure security testing during every build. Infrastructure code will be checked for security compliance using tools like Terraform and AWS Config. Continuous monitoring and log analysis tools will be employed to detect and respond to security incidents in real-time. Automation will also generate compliance reports, express security policies as code, and trigger remediation actions when issues are identified. This integrated DevSecOps approach ensures security is an integral part of the development lifecycle.

Furthermore, Green Pace will utilize container security scanning tools and implement security training and awareness programs through automation. By integrating these tools and practices, Green Pace aims to create a streamlined DevSecOps pipeline that enforces security standards consistently across all stages of development, from code creation to deployment. This approach enhances security, accelerates development, and reduces the risk of security vulnerabilities in software products.

### 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 | Low | Unlikely | Low | P3 | L3 |
| STD-002-CPP | Low | Unlikely | Low | P3 | L3 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | High | Likely | Medium | P18 | L1 |
| STD-007-CPP | High | Likely | Medium | P18 | L1 |
| STD-008-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-009-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-010-CPP | Medium | Probable | Medium | P8 | L2 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption in rest is used to safeguard data while it is in storage from unauthorized users. The encryption is performed once the data is sent to its storage location. The policy mandates the use of in rest encryption for sensitive data when it is stored on servers, databases, or cloud storage to prevent data breaches in case of unauthorized access to physical or digital storage devices. |
| Encryption at flight | Encryption in flight is used to secure data as it is transmitted over a network to ensure it remains confidential and protected from eavesdroppers. It is applied when data is being sent from one point to another. The policy mandates the use of in-flight encryption whenever sensitive data is transmitted over a network, both internal and external networks, to help protect data privacy during transit. |
| Encryption in use | Encryption in use protects data when it is actively being processed by applications or services and ensures the data remains confidential during computation. The policy mandates the use of in use encryption when sensitive data needs to be processed by applications and services to maintain its integrity during computations and manipulations. |

| 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 users or systems attempting to access data or systems. Authentication methods such as passwords, biometrics, and multi-factor authentication are implemented to confirm the identity of users before granting them access. The policy requires strong authentication mechanisms for users attempting to access sensitive resources to protect against unauthorized access and data breaches. |
| Authorization | Authorization determines what actions or resources authenticated users or systems are allowed to access. A common authorization framework is role-based access control, which enforces access control policies based on roles and permissions. The policy requires authenticated users have authorization to access only the resources they need to do their job and nothing more, to minimize the risk of unauthorized access and data misuse. |
| Accounting | Accounting involves tracking and logging all activities related to data access and system operations. Logging and monitoring systems are implemented to record and analyze user activities, enabling the detection of suspicious or unauthorized actions. The policy requires accounting all data access and system activities to help with identifying security incidents and ensuring compliance with security policies and regulations. |

**\***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.

Data Type Standard: Principles 1 and 3.

Validating input data (Principle 1) is crucial for ensuring that the data type of incoming information is correct and matches the expected format. Designing with security policies in mind (Principle 3) involves considering data types as part of the security architecture, ensuring that the data is used and validated correctly to prevent type-related vulnerabilities.

Data Value Standard: Principles 1 and 6.

Validating input data (Principle 1) includes checking not only data type but also the specific values. This ensures that data values conform to expected ranges or patterns. The default deny principle (Principle 6) relates to rejecting data values that do not meet predefined criteria, which is essential for data value validation.

String Correctness Standard: Principles 1 and 9.

Validating input data (Principle 1) includes validating strings to ensure they are correctly formatted and free from malicious content. Effective quality assurance techniques (Principle 9) encompass thorough testing of string inputs to verify their correctness, including boundary testing, input validation, and regular expression checks.

SQL Injection Standard: Principles 1, 7, and 10.

Validating input data (Principle 1) is critical to preventing SQL injection attacks, as it involves checking and sanitizing user inputs before they are used in SQL queries. Sanitizing data sent to other systems (Principle 7) ensures that data being passed to databases or other systems is free from malicious SQL injection code. Adopting a secure coding standard (Principle 10) sets guidelines and best practices for preventing SQL injection vulnerabilities.

Memory Protection Standard: Principles 2, 4, and 8.

Heeding compiler warnings (Principle 2) is essential for identifying potential memory-related issues at compile time, reducing the risk of memory vulnerabilities. Keeping code simple (Principle 4) can help minimize complex memory management operations, reducing the likelihood of memory-related bugs. Practicing defense in depth (Principle 8) includes implementing multiple layers of security, which can include memory protection mechanisms to safeguard against exploitation.

Assertions Standard: Principles 2 and 9.

Heeding compiler warnings (Principle 2) includes addressing issues identified by the compiler, which may involve using assertions to validate assumptions and catch errors. Effective quality assurance techniques (Principle 9) encompass the use of assertions as a tool for testing and ensuring code correctness.

Exceptions Standard: Principles 2 and 8.

Heeding compiler warnings (Principle 2) may involve addressing exception-related warnings, ensuring proper exception handling. Practice defense in depth (Principle 8) includes considering exception handling as a part of overall system defense, ensuring that exceptions do not lead to security vulnerabilities.

Formatting and Indentation Standard: Principles 4 and 9.

Keeping code simple (Principle 4) includes maintaining consistent code formatting and indentation practices, which enhances code readability and reduces the likelihood of introducing errors. Effective quality assurance techniques (Principle 9) involve code reviews that may include formatting and indentation checks to ensure adherence to coding standards.

Commenting and Documentation Standard: Principles 3 and 8.

Architecting and designing for security policies (Principle 3) may involve documenting security-related design decisions and rationale. Practicing defense in depth (Principle 8) includes documenting security measures and considerations to ensure that security is a well-documented aspect of the system.

Version Control Standard: Principles 6 and 8.

Default deny (Principle 6) can be reinforced by version control practices that ensure only authorized changes are accepted into the codebase. Practice defense in depth (Principle 8) may involve version control for tracking security-related changes and managing code revisions in a secure manner.

**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 |  |
| 2.0 | 11/13/2023 | Ten Core Security Principles  Coding Standards | Carmen Kingery |  |
| 3.0 | 12/09/2023 | Revised Standards  Risk Assessment  Automation Detection  Automation Reasoning  Summary of Risk Assessment  Encryption Policies | Carmen Kingery |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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