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



# Green Pace Secure Development Policy

# Vincent Yanez

## 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 | Always check and sanitize user input to prevent attacks like SQL injection and buffer overflows. Failing to do so can allow attackers to inject malicious code and compromise your system. |
| 1. Heed Compiler Warnings | Pay attention to compiler warnings as they often highlight security vulnerabilities in your code. Ignoring them can lead to serious bugs that attackers may exploit. |
| 1. Architect and Design for Security Policies | Build systems with security in mind from the start, ensuring they follow access control**,** encryption, and authentication best practices. A strong architecture minimizes security risks and reduces vulnerabilities. |
| 1. Keep It Simple | Complex systems have more vulnerabilities, design code and security measures to be as simple and clear as possible. Simplicity makes it easier to identify, fix, and prevent security flaws. |
| 1. Default Deny | Block all access by default and only allow what is explicitly permitted to reduce security risks. This approach ensures that unauthorized users or processes cannot access sensitive data. |
| 1. Adhere to the Principle of Least Privilege | Give users and programs only the permissions they need to function, preventing unnecessary access to sensitive data. This limits potential damage if an account or system is compromised. |
| 1. Sanitize Data Sent to Other Systems | Clean and validate all data before passing it to databases, APIs, or other systems to prevent injections and exploits. Untrusted data can lead to breaches, data corruption, or unauthorized access. |
| 1. Practice Defense in Depth | Layer multiple security measures (e.g., firewalls, encryption, access controls) to reduce the risk of a single point of failure. If one layer is compromised, others still provide protection. |
| 1. Use Effective Quality Assurance Techniques | Regularly test code with security reviews, automated tools, and penetration testing to catch vulnerabilities early. A strong QA process helps detect security flaws before they reach production. |
| 1. Adopt a Secure Coding Standard | Follow industry best practices for writing secure code to prevent common weaknesses and ensure consistency. Standardized coding reduces errors and makes security audits easier. |

### 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-DT] | |  |  | | --- | --- | |  | Use Fixed-Width Integer Types | |

| **Noncompliant Code** |
| --- |
| Using standard integer types can lead to platform-specific issues. In this example, int may have a different size on different systems, which can cause problems when storing large numbers. For instance, on a 32-bit system, int may not be large enough to store big values correctly. |
| #include <iostream>  int main() {  int x = 1000000000; // int size may vary depending on platform (e.g., 32-bit or 64-bit)  std::cout << "Value of x: " << x << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| Using fixed-width integer types ensures predictable behavior. Here, int32\_t is used, which always has a size of 32 bits, no matter the system. This ensures the code behaves the same across all platforms, avoiding issues with varying integer sizes. |
| #include <iostream>  #include <cstdint> // For fixed-width integer types  int main() {  int32\_t x = 1000000000; // int32\_t guarantees 32-bit size across all platforms  std::cout << "Value of x: " << x << std::endl;  return 0;  } |

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

| **Principles(s):** Consistency and Precision in Data Representation-Using fixed-width integers (like int32\_t) ensures that the data behaves the same way on all systems, preventing errors like overflow. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | latest | |  |  |  | | --- | --- | --- | | |  | | --- | | cppcoreguidelines-pro-type-fixed-width-  int |  |  | | --- | |  | |  |  | | --- | |  | | Checks for safe usage of fixed-width integers |
| |  | | --- | | **Coverity** |  |  | | --- | |  | | latest | |  | | --- | | C++-INT-FIXED |  |  | | --- | |  | | |  | | --- | | Identifies potential integer overflow issues | |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-DV] | |  | | --- | |  |  |  | | --- | | Do Not Read Uninitialized Memory | |

| **Noncompliant Code** |
| --- |
| The variable x is used without being initialized, leading to undefined behavior. In this example, the variable x is declared but not initialized. When we use x in the cout statement, it may contain random data, leading to unpredictable output. |
| #include <iostream>  int main() {  int x; // Uninitialized variable  std::cout << "Value of x: " << x << std::endl; // Using x before initializing it  return 0;  } |

| **Compliant Code** |
| --- |
| The variable x is initialized before use, ensuring predictable behavior. Here, x is initialized to 0 before being used, ensuring that it has a known value and avoiding undefined behavior. |
| #include <iostream>  int main() {  int x = 0; // Variable initialized before use  std::cout << "Value of x: " << x << std::endl; // Safe to use  return 0;  } |

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

| **Principles(s):** Clear Intent and Predictable Behavior- Initializing variables before use ensures predictable behavior and helps prevent undefined or erratic outcomes. This improves code reliability, readability, and debugging efficiency. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| |  | | --- | | **clang-tidy** |  |  | | --- | |  | | latest | |  | | --- | |  |  |  | | --- | | cppcoreguidelines-pro-type-uninitialized | | Detects use of uninitialized variables |
| Coverity | latest | C++-UNINITIALIZED | |  | | --- | |  |  |  | | --- | | Identifies places where variables are used before being initialized, which can lead to undefined behavior | |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-SC] | Ensure Correctness of String Handling |

| **Noncompliant Code** |
| --- |
| The code improperly uses a string without ensuring enough space is allocated, which can lead to a buffer overflow. In this example, str is allocated only 5 bytes, but the string "Hello, world!" is much longer. This causes a buffer overflow, which can corrupt data and lead to crashes. |
| #include <iostream>  #include <cstring> // For strcpy()  int main() {  char str[5];  strcpy(str, "Hello, world!"); // Buffer overflow - str is too small to hold the string  std::cout << str << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| The string is properly handled by ensuring there is enough space in the buffer, preventing buffer overflow. Here, we ensure that str has enough space for the string "Hello, world!" by using strncpy(). We also ensure null-termination by explicitly adding a '\0' at the end, avoiding buffer overflows. |
| #include <iostream>  #include <cstring> // For strncpy()  int main() {  char str[20];  strncpy(str, "Hello, world!", sizeof(str) - 1); // Proper bounds checking  str[sizeof(str) - 1] = '\0'; // Ensuring null-termination  std::cout << str << std::endl;  return 0;  } |

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

| **Principles(s):** Prevent Buffer Overflow- Make sure there's enough space for a string. If not, it can cause a **buffer overflow**, leading to data corruption or crashes. This standard uses safer functions like strncpy() to prevent that. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| **clang-tidy** | latest | cppcoreguidelines-strf-string | Detects unsafe string functions that could potentially lead to buffer overflow issues. |
| **Coverity** | latest | C++-STR-BOUNDS | Detects out-of-bounds access in strings and other unsafe string manipulations. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-SI] | Prevent SQL Injection |

| **Noncompliant Code** |
| --- |
| The code constructs an SQL query directly with user input, making it vulnerable to SQL injection attacks. This code inserts user input directly into the SQL query, which can be exploited by attackers to manipulate the query, causing security risks. |
| #include <iostream>  #include <string>  #include <mysql/mysql.h>  int main() {  std::string username = "user";  std::string password = "password";    MYSQL \*conn;  conn = mysql\_init(0);  mysql\_real\_connect(conn, "localhost", "root", "root", "database", 3306, NULL, 0);  // Vulnerable to SQL Injection  std::string query = "SELECT \* FROM users WHERE username = '" + username + "' AND password = '" + password + "'";  mysql\_query(conn, query.c\_str());  mysql\_close(conn);  return 0;  } |

| **Compliant Code** |
| --- |
| This code uses a safe way to handle user input, preventing SQL injection by using prepared statements. In this compliant code, we use a prepared statement, which ensures that user input is safely inserted into the query. This prevents any malicious manipulation of the SQL query. |
| #include <iostream>  #include <string>  #include <mysql/mysql.h>  int main() {  // Define username and password  std::string username = "user";  std::string password = "password";    // Initialize MySQL connection  MYSQL \*conn = mysql\_init(0);  mysql\_real\_connect(conn, "localhost", "root", "root", "database", 3306, NULL, 0);  // Prepare and execute a SQL query safely using a prepared statement  MYSQL\_STMT \*stmt = mysql\_stmt\_init(conn);  std::string query = "SELECT \* FROM users WHERE username = ? AND password = ?";  mysql\_stmt\_prepare(stmt, query.c\_str(), query.length());  // Bind parameters for the query  MYSQL\_BIND bind[2];  bind[0].buffer\_type = MYSQL\_TYPE\_STRING;  bind[0].buffer = (char\*)username.c\_str();  bind[0].buffer\_length = username.length();  bind[1].buffer\_type = MYSQL\_TYPE\_STRING;  bind[1].buffer = (char\*)password.c\_str();  bind[1].buffer\_length = password.length();  // Bind parameters and execute the statement  mysql\_stmt\_bind\_param(stmt, bind);  mysql\_stmt\_execute(stmt);  // Close the statement and connection  mysql\_stmt\_close(stmt);  mysql\_close(conn);  return 0;  } |

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

| **Principles(s):** Input Validation and Secure Output Encoding-  Ensures that input values are treated as data, not executable code, by using prepared statements. Prevents attackers from injecting SQL commands. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | latest | cppcoreguidelines-sql-injection | Detects unsafe SQL practices and potential vulnerabilities. |
| Coverity | latest | C++-SQL-INJECTION | Identifies SQL injection vulnerabilities and recommends using secure practices such as prepared statements. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-MP] | Avoid Memory Leaks by Using Smart Pointers |

| **Noncompliant Code** |
| --- |
| In this code, memory is allocated with new, but it's never freed with delete, causing a memory leak. The memory is still being used but can't be accessed, wasting resources. |
| #include <iostream>  int main() {  int\* ptr = new int(10); // Memory allocated  std::cout << \*ptr << std::endl;  // Memory not freed (leak)  return 0;  } |

| **Compliant Code** |
| --- |
| This code uses std::unique\_ptr, which automatically takes care of memory. When the pointer goes out of scope, it automatically frees the memory, preventing memory leaks. You don’t need to manually delete memory with std::unique\_ptr. |
| #include <iostream>  #include <memory> // For smart pointers  int main() {  std::unique\_ptr<int> ptr = std::make\_unique<int>(10); // Memory automatically managed  std::cout << \*ptr << std::endl;  // No need to manually delete; memory will be freed when ptr goes out of scope  return 0;  } |

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

| **Principles(s):** Use smart pointers to manage memory automatically-Smart pointers (like std::unique\_ptr and std::shared\_ptr) automatically free memory when it's no longer needed, which helps prevent memory leaks and makes your code safer and easier to maintain. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| |  | | --- | | clang-tidy |  |  | | --- | |  | | latest | cppcoreguidelines-owning-memory | Ensures memory ownership is clear and safely managed, reducing the risk of leaks or dangling pointers. |
| Coverity | latest | |  | | --- | |  | | C++-MEMORY-LEAK | |  |  | | --- | |  | | Detects cases where allocated memory is not properly released, causing potential memory leaks. |

|  |
| --- |
|  |

#### 

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-AS] | Use Assertions for Error Checking |

| **Noncompliant Code** |
| --- |
| This code doesn't check if the value is valid, potentially causing undefined behavior if the value is invalid. The code does not check if value is valid, which could cause issues if the value is not what is expected like, negative values when only positive values are valid. It may lead to unexpected results or errors in more complex scenarios. |
| #include <iostream>  int main() {  int value = -1;  std::cout << "Value: " << value << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| This code uses assert to check if the value is valid before proceeding, ensuring that only valid values are used. Here, the assert checks if value is greater than or equal to zero. If it's not, the program will terminate with an error message, preventing invalid behavior or unintended results. |
| #include <iostream>  #include <cassert> // For assert  int main() {  int value = -1;  assert(value >= 0); // Assert that value is non-negative  std::cout << "Value: " << value << std::endl;  return 0;  } |

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

| **Principles(s):** **Error Checking with Assertions-**Assertions check for impossible or unexpected conditions in the code during development. They help catch errors early but are usually disabled in production for better performance. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| |  | | --- | | clang-tidy |  |  | | --- | |  | | latest | |  | | --- | |  |  |  | | --- | | cppcoreguidelines-assert | | |  | | --- | |  |  |  | | --- | | Detects improper use or absence of assertions in the code. | |
| Coverity | latest | |  | | --- | | C++-ASSERTION |  |  | | --- | |  | | Detects unsafe or missing assertions in C++ code. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-EX] | Use Exceptions for Error Handling |

| **Noncompliant Code** |
| --- |
| Using return codes to signal errors can be unclear and difficult to manage, especially in larger applications. This code uses error codes (like -1 for division by zero), but it’s easy to forget to handle the error properly, which can lead to unhandlederrors in larger programs. |
| #include <iostream>  int divide(int a, int b) {  if (b == 0) return -1; // Error code for division by zero  return a / b;  }  int main() {  int result = divide(10, 0);  if (result == -1) {  std::cout << "Error: Division by zero!" << std::endl;  } else {  std::cout << "Result: " << result << std::endl;  }  return 0;  } |

| **Compliant Code** |
| --- |
| This code uses exceptions to handle errors, making error handling clearer and more reliable. If an error occurs, it’s thrown and caught in a catch block, preventing unhandled errors and improving code safety. |
| #include <iostream>  #include <stdexcept> // For exceptions  int divide(int a, int b) {  if (b == 0) throw std::invalid\_argument("Division by zero!"); // Throw exception  return a / b;  }  int main() {  try {  std::cout << "Result: " << divide(10, 0) << std::endl;  } catch (const std::exception& e) {  std::cout << "Error: " << e.what() << std::endl; // Handle exception  }  return 0;  } |

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

| **Principles(s):** Robust Error Management-Using exceptions separates error-handling code from regular code, making programs easier to maintain and less error-prone. It improves clarity and allows for consistent handling of unexpected conditions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| |  | | --- | | Clang-Tidy |  |  | | --- | |  | | latest | cppcoreguidelines-avoid-exceptions | Flags improper or discouraged use of exceptions. |
| Coverity | latest | |  | | --- | | C++-EXCEPTION-HANDLING |  |  | | --- | |  | | Detects improper exception usage or missing handlers |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Code Readability** | [STD-008-CR] | Use Meaningful Variable and Function Names |

| **Noncompliant Code** |
| --- |
| In this code, the variable and function names are unclear and don’t describe their purpose, making the code harder to understand. The function name a and variable names b and c don’t convey any meaning about their purpose. Anyone reading the code will struggle to understand what the code is doing. |
| #include <iostream>  int a(int b, int c) {  return b + c; // Unclear function name 'a' and variables 'b', 'c'  }  int main() {  int x = 5, y = 10;  std::cout << a(x, y) << std::endl; // Hard to understand what 'a' does  return 0;  } |

| **Compliant Code** |
| --- |
| In this code, the function and variable names are descriptive, making it easy to understand what the code does. The function addNumbers and the variables firstNumber and secondNumber clearly describe their purpose, making the code easier to understand and maintain. |
| #include <iostream>  int addNumbers(int firstNumber, int secondNumber) {  return firstNumber + secondNumber; // Descriptive function and variable names  }  int main() {  int number1 = 5, number2 = 10;  std::cout << addNumbers(number1, number2) << std::endl; // Clear function call  return 0;  } |

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

| **Principles(s):** Improve Code Clarity and Maintainability- Using descriptive names makes code easier to understand, helping developers detect bugs and make updates more easily. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | latest | readability-identifier-naming | Flags poorly named or inconsistent identifiers |
| Coverity | latest | C++-NAMING-CONVENTION *(if available)* | Detects unclear or inconsistent naming patterns |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Comment Code** | [STD-009-CC] | Comment Code to Explain Complex Logic the standard |

| **Noncompliant Code** |
| --- |
| Without comments, the purpose of the calculateSum function and the specific calculation a + b \* c is unclear. Someone else reading the code might not immediately understand the intention behind this calculation. |
| #include <iostream>  int calculateSum(int a, int b, int c) {  return a + b \* c; // What does this calculation mean?  }  int main() {  int x = 5, y = 10, z = 3;  std::cout << calculateSum(x, y, z) << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| In this version, the function calculateSum is clearly described with a comment, explaining that it adds a to the product of b and c. This makes the code easier to understand and maintain. |
| #include <iostream>  // Function to calculate the sum of a and the product of b and c  int calculateSum(int a, int b, int c) {  return a + b \* c; // Adding 'a' to the product of 'b' and 'c'  }  int main() {  int x = 5, y = 10, z = 3;  std::cout << calculateSum(x, y, z) << std::endl;  return 0;  } |

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

| **Principles(s):** Improve Maintainability and Collaboration- Commenting helps explain complex logic, making it easier for others (and your future self!) to understand, debug, or extend the code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | latest | readability-misleading-indentation | Ensures clarity around blocks that could be misread |
| |  | | --- | | Coverity |  |  | | --- | |  | | latest | C++-MISSING-COMMENT *(if available)* | |  | | --- | | Flags public methods lacking documentation | |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Avoid Magic Numbers | [STD-010-NF] | Use of Named Constants Instead of Magic Numbers |

| **Noncompliant Code** |
| --- |
| Using raw numerical values (magic numbers) directly in the code. The numbers 10 and 5 are hardcoded into the function, making it unclear what they represent. This can be confusing and error-prone if the same value needs to be changed in multiple places. |
| #include <iostream>  int calculateArea() {  int length = 10;  int width = 5;  return length \* width; // Using magic numbers for length and width  }  int main() {  std::cout << calculateArea() << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant code, LENGTH and WIDTH are defined as constants, which makes it clear what these values represent and makes the code more maintainable. By using named constants instead of magic numbers, you improve the clarity of the code and make it easier to modify values later. |
| #include <iostream>  const int LENGTH = 10; // Define constant for length  const int WIDTH = 5; // Define constant for width  int calculateArea() {  return LENGTH \* WIDTH; // Use named constants instead of magic numbers  }  int main() {  std::cout << calculateArea() << std::endl;  return 0;  } |

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

| **Principles(s):** Improve Readability and Maintainability- Named constants make code easier to understand, debug, and maintain. Replacing unexplained numbers with descriptive names clarifies the code’s intent and reduces errors from using incorrect or outdated values. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| |  | | --- | | Clang-Tidy |  |  | | --- | |  | | [Insert text.] | |  | | --- | | cppcoreguidelines-avoid-magic-numbers |  |  | | --- | |  | | |  | | --- | |  |  |  | | --- | | Flags hard-coded numeric literals that should be replaced by constants | |
| |  | | --- | | Coverity |  |  | | --- | |  | | [Insert text.] | |  | | --- | | C++-MAGIC-NUMBER *(if available)* |  |  | | --- | |  | | |  | | --- | | Detects magic numbers in code logic | |

### 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.

**Automation Explanation**

Automation will help enforce the C/C++ coding standards throughout the DevSecOps pipeline.

* **Pre-production (Left side of diagram):**  
  During planning, design, and build, tools like **Clang-Tidy** will check code for standard violations (e.g., memory safety, assertions, naming rules). These tools run automatically in the CI/CD pipeline.
* **Verify and Test:**  
  Automated tests and scans (static and dynamic) will catch errors before release, ensuring code meets security and quality standards.
* **Production (Right side of diagram):**  
  Monitoring tools will continue checking the software after deployment. If issues are found, automated alerts and responses will help teams fix them quickly.

This process keeps code secure and standards-compliant at every stage of development.

### 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-DT | High | Unlikely | Medium | High | 2 |
| STD- 002-DV | High | Likely | Medium | High | 4 |
| STD- 003-SC | Medium | Likely | Low | Medium | 3 |
| STD- 004-SI | High | Likely | Medium | High | 4 |
| STD- 005-MP | Medium | Possible | Low | Medium | 3 |
| STD- 006-AS | Medium | Possible | Low | Medium | 3 |
| STD- 007-EX | High | Possible | Medium | High | 4 |
| STD- 008-CR | Low | Likely | Low | Low | 2 |
| STD- 009-CC | Low | Likely | Low | Low | 2 |
| STD- 010-NF | Medium | Possible | Low | Medium | 3 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | **What it is**: Protecting data that is stored on disk, ensuring it is unreadable without proper decryption keys.  **How it should be applied**: Data stored on servers, databases, or backup media should be encrypted using strong algorithms for example: AES-256.  **Why it should be used**: Prevents unauthorized access to sensitive data in case of physical theft, system breaches, or storage device compromise. |
| Encryption in flight | **What it is**: Securing data as it is transmitted over a network, using HTTPS, TLS.  **How it should be applied**: Ensure all data transferred over networks is encrypted using protocols like TLS or SSL.  **Why it should be used**: Protects data from interception, man-in-the-middle attacks, or eavesdropping during transmission over public or private networks. |
| Encryption in use | **What it is**: Protecting data while it is being processed or actively used in memory.  **How it should be applied**: Use techniques like homomorphic encryption or secure enclaves to keep data encrypted during computation.  **Why it should be used**: Prevents unauthorized access to sensitive data while in use, reducing the risk of attacks on live data in memory. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | **What it is**: Verifying the identity of a user or system, typically through usernames, passwords, or multi-factor authentication.  **How it should be applied**: Implement secure login mechanisms and require multiple factors for sensitive systems.  **Why it should be used**: Ensures only legitimate users or systems gain access, protecting against unauthorized access. |
| Authorization | **What it is**: Determining what actions or resources a user or system is allowed to access after authentication.  **How it should be applied**: Assign roles and permissions based on user needs, enforcing the principle of least privilege.  **Why it should be used**: Limits access to sensitive data and actions, reducing the risk of misuse or accidental exposure. |
| Accounting | **What it is**: Tracking and logging user activity to monitor and audit actions performed in the system.  **How it should be applied**: Record and regularly review logs for all critical actions, including logins, data access, and changes to the system.  **Why it should be used**: Provides a trail for accountability, enabling detection of suspicious activities and aiding in investigations. |

**\***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

Standard Mapped Principles

STD-001-DT (Data Type Checks) 1, 4, 10

Validating input (Principle 1) prevents improper data types. Simplicity (4) makes data type handling more reliable. Secure coding standards (10) ensure consistent type validation.

STD-002-DV (Data Validation) 1, 7, 10

Input validation (1) and sanitizing data to other systems (7) are the foundation of secure data handling. Secure coding standards (10) promote consistent validation practices.

STD-003-SC (System Configuration) 3, 5, 6, 8

Designing secure systems (3), using default deny (5), enforcing least privilege (6), and layering defenses (8) are key to secure configurations.

STD-004-SI (System Interfaces) 1, 3, 7

Validating data to/from interfaces (1, 7) and securely designing those interfaces (3) minimizes risks of data leaks or injection.

STD-005-MP (Memory Protection) 2, 4, 9

Heeding compiler warnings (2) catches memory bugs early. Simplicity (4) helps manage memory safely. QA (9) ensures memory vulnerabilities are tested.

STD-006-AS (Application Security) 3, 6, 8, 10

Secure design (3), least privilege (6), layered defense (8), and secure coding practices (10) build a strong application defense

STD-007-EX (Exception Handling) 4, 9, 10

Keeping exception handling simple (4) avoids exposing sensitive data. QA (9) ensures it's tested, and secure coding (10) provides consistent practices.

STD-008-CR (Credential Rules) 1, 3, 6

Valid input (1), secure design (3), and least privilege (6) prevent unauthorized access and protect credentials.

STD-009-CC (Code Comments) 2, 4, 10

Avoiding revealing info in comments (2), keeping them simple (4), and following secure coding (10) prevents leakage of sensitive data.

STD-010-NF (Naming Functions) 4, 10

Simple, clear names (4) reduce confusion and potential misuse. Following standards (10) ensures consistency and security awareness.

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

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 | 04/11/2025 | Initial Template | Vincent Yanez | Aaron Demory |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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