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

Malik Spruill



# 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 principle involves validating user input to prevent data entries from causing or creating security vulnerabilities. Vulnerabilities from not implementing this principle could involve SQL injection, buffer overflow attacks, and more. Character whitelisting, input length restrictions, these are a few ways to validate data. |
| 1. Heed Compiler Warnings | This principle involves ensuring compiler warnings aren’t left unnoticed, considering potential security flaws may be exploited. |
| 1. Architect and Design for Security Policies | This principle indicates a system should be designed with security in mind, which helps better enforce proper authentication, encryption, and access. |
| 1. Keep It Simple | This principle involves ensuring systems aren’t complexity-favored. Meaning, being minimalistic with dependencies and simplifying code can reduce areas of attack. |
| 1. Default Deny | This principle involves ensuring access being granted to authorized users only, reducing unauthorized access. |
| 1. Adhere to the Principle of Least Privilege | This principle involves only having users and applications be assigned the minimum level of access, reducing chances of security breaches. |
| 1. Sanitize Data Sent to Other Systems | This principle ensures data is sanitized before being transmitted to other systems, preventing injection attacks. This involves techniques such as encoding output, escaping special characters, and more to ensure data integrity and security. |
| 1. Practice Defense in Depth | This principle involves having multiple layers of security, which could be a combination of firewalls, multi-factor authentication, and other measures that increase protection. If one of the layers become compromised, the other layers will prevent an attacker from gaining full access to critical systems. |
| 1. Use Effective Quality Assurance Techniques | This principle involves using QA techniques like common testing methods, such as penetration testing, as well as code reviews, to detect and fix vulnerabilities before deployment. |
| 1. Adopt a Secure Coding Standard | This principle involves following industry-recognized secure coding standards, like CERT guidelines, to help write secure 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-002-ABC] | Ensure Proper Array Bounds Checking – To access arrays without bounds checking, this can lead to buffer overflows. Proper validation prevents memory corruption. |

| **Noncompliant Code** |
| --- |
| The code below will access the array without checking bounds, leading to out-of-bounds memory access |
| #include <iostream>  void printEl (int arr[], int size, int index) {  std::cout << "Element: " << arr[index] << std::endl;  }  int main() {  int numbers[] = {10, 20, 30, 40};  printEl(numbers, 4, 5);  } |

| **Compliant Code** |
| --- |
| The code below will ensure the index is within valid range before accessing the array. This will prevent buffer overflows. |
| #include <iostream>  #include <stdexcept>  void printEl(int arr[], int size, int index) {  if (index < 0 || index >= size) {  throw std::out\_of\_range("Index is out of bounds.");  }  std::cout << "Element: " << arr[index] << std::endl;  }  int main() {  int numbers[] = {10, 20, 30, 40};  try { printEl(numbers, 4, 2);  } catch (const std::exception& e) {  std::cerr << "Error: " << e.what() << std::endl; }  return 0;  } |

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

| **Principles(s):**  Principle 1 – Applies because it ensures that only properly formatted and expected data types are used. This prevents type mismatches and vulnerabilities like buffer overflows.  Principle 2 – Applies by helping catch type-related issues, such as conversions or unsafe typecasts. This would lead to security flaws if left ignored.  Principle 10 – Applies by promoting consistent, safe use of data types, reducing errors like integer overflows and memory corruption. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | cppcoreguidelines | Clang-Tidy is a C++ “linter” and static analysis tool. The tool would help conduct a variety for code to adhere to the C++ core guidelines, identifying mismatches and other issues.  Link: https://clang.llvm.org/extra/clang-tidy/ |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-004-DIV] | Prevent Division by Zero Errors – Division by zero is an undefined operation, which can lead to crashes and other unpredictable behavior. With proper validation, this will ensure the denominator is never 0, improving security. |

| **Noncompliant Code** |
| --- |
| This code will not check if the denominator is zero before performing a division, leading to potential errors. |
| #include <iostream>  double divide(int numerator, int denominator) {  return numerator / denominator.  }  int main() {  int a = 10, b = 0;  std::cout << "Result: " << divide(a, b) << std::endl;  } |

| **Compliant Code** |
| --- |
| This code will properly validate the denominator before performing division, ensuring safe execution. |
| #include <iostream>  #include <stdexcept>  double divide(int numerator, int denominator) {  if (denominator == 0) { throw std::invalid\_argument("Error: Division by zero is not allowed.");  }  return static\_cast<double>(numerator) / denominator;  }  int main() {  try {  int a = 10, b = 0; std::cout << "Result: " << divide(a, b) << std::endl;  } catch (const std::exception& e) {  std::cerr << "Exception: " << e.what() << std::endl;  }  return 0;  } |

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

| **Principles(s):**  Principle 1 – Applies by ensuring values fall within expected ranges, preventing issues like integer overflows or invalid inputs  Principle 6 – Applies by limiting access to critical data values, ensuring only authorized users or processes can access or utilize sensitive info.  Principle 7 – Applies by ensuring data values are filtered and correctly formatted before transmitting to other parts of the system. This prevents injection attacks, for example |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | Built-in data value analysis | Cppcheck is a static analysis tool designed for C++ code. This tool would help detect real errors, minimizing false positives.  Link: https://cppcheck.sourceforge.io/ |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-005-STR] | Ensure Proper String Handling and Safety – Improper handling of strings, this will cause memory corruption and open room for security vulnerabilities. Safe use of string operations will ensure higher chance of code stability and security |

| **Noncompliant Code** |
| --- |
| This example demonstrates an unsafe use of C-style strings, which can lead to buffer overflows and undefined behavior. |
| #include <iostream>  #include <cstring>  void unsafeCopy(const char\* input) {  char buffer[10]; // Fixed-size buffer  std::strcpy(buffer, input); // Unsafe: No bounds checking!  std::cout << "Copied string: " << buffer << std::endl;  }  int main() {  const char\* userInput = "ThisStringIsTooLong"; // Exceeds buffer size  unsafeCopy(userInput); // Potential buffer overflow  return 0;  } |

| **Compliant Code** |
| --- |
| This example correctly uses std::string, which provides automatic memory management and prevents buffer overflows. |
| #include <iostream>  #include <string>  void safeCopy(const std::string& input) {  std::string buffer = input.substr(0, 10); // Ensures no overflow  std::cout << "Copied string: " << buffer << std::endl;  }  int main() {  std::string userInput = "ThisStringIsTooLong"; // Longer than 10 characters  safeCopy(userInput); // Safe copy with length restriction  return 0;  } |

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

| **Principles(s):**  Principle 1 – Applies by ensuring strings adhere to formatting restrictions, preventing injection attacks, overflows, and more.  Principle 5 – Applies by enforcing strict string operations, rejection formats that are incorrect. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OCLint | 21.10 | C++ code quality and bug detection | OCLint is an open-source static analysis tool that helps improve C++ code quality. It does so by detecting potential problems, including string operation issues like buffer overflows.  Link: https://github.com/oclint/oclint |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-006-SQL] | Use Parameterized Queries to Prevent SQL Injection - SQL injection occurs when user input is directly added to SQL queries, counting as a part of the query itself versus a string. Parameterized queries or prepared statements will best prevent injection attacks. |

| **Noncompliant Code** |
| --- |
| This example directly concatenates user input into an SQL query string, making it susceptible to SQL injection attacks. |
| #include <iostream>  #include <sqlite3.h>  void unsafeQuery(sqlite3\* db, const std::string& userInput) {  std::string query = "SELECT \* FROM users WHERE username = '" + userInput + "';"; // Unsafe concatenation  char\* errMsg = nullptr;    if (sqlite3\_exec(db, query.c\_str(), nullptr, nullptr, &errMsg) != SQLITE\_OK) {  std::cerr << "SQL error: " << errMsg << std::endl;  sqlite3\_free(errMsg);  } else {  std::cout << "Query executed: " << query << std::endl;  }  }  int main() {  sqlite3\* db;  sqlite3\_open(":memory:", &db); // Open in-memory database for demonstration  std::string userInput = "'; DROP TABLE users; --"; // Malicious input  unsafeQuery(db, userInput); // SQL Injection possible!  sqlite3\_close(db);  return 0;  } |

| **Compliant Code** |
| --- |
| This example correctly uses **parameterized queries** to prevent SQL injection. |
| #include <iostream>  #include <sqlite3.h>  void safeQuery(sqlite3\* db, const std::string& userInput) {  const char\* sql = "SELECT \* FROM users WHERE username = ?;"; // Parameterized query  sqlite3\_stmt\* stmt;  if (sqlite3\_prepare\_v2(db, sql, -1, &stmt, nullptr) == SQLITE\_OK) {  sqlite3\_bind\_text(stmt, 1, userInput.c\_str(), -1, SQLITE\_STATIC); // Safely bind user input    while (sqlite3\_step(stmt) == SQLITE\_ROW) {  std::cout << "User found: " << sqlite3\_column\_text(stmt, 0) << std::endl;  }    sqlite3\_finalize(stmt); // Properly free statement  } else {  std::cerr << "Failed to prepare statement." << std::endl;  }  }  int main() {  sqlite3\* db;  sqlite3\_open(":memory:", &db); // Open in-memory database for demonstration  std::string userInput = "'; DROP TABLE users; --"; // Attempted SQL injection  safeQuery(db, userInput); // Injection prevented!  sqlite3\_close(db);  return 0;  } |

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

| **Principles(s):**  Principle 1 – Applies by insuring user inputs are checked for formatting and prohibited if containing malicious SQL code. This helps prevent injection attacks.  Principle 8 – Applies by reinforcing security by implementing layers of protection which could consist of input validation and least privilege for database access. This helps prevent injection attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| RATS | 2.4 | SQL injection vulnerability detection | RATS is an open-source static analysis tool that scans C and C++ source code, identifying security vulnerabilities like SQL injection flaws.  Link: https://code.google.com/archive/p/rough-auditing-tool-for-security/ |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-007-MEM] | Ensure Proper Memory Management to Prevent Leaks and Dangling Pointers – improper memory management techniques could lead to memory leaks and an array of other behaviors. Smart pointers and proper deallocation techniques will improve stability and security. |

| **Noncompliant Code** |
| --- |
| This example manually allocates memory but forgets to free it, leading to a **memory leak**. |
| #include <iostream>  void unsafeMemoryUsage() {  int\* ptr = new int[10]; // Dynamically allocated array  for (int i = 0; i < 10; i++) {  ptr[i] = i \* 2;  }  std::cout << "First element: " << ptr[0] << std::endl;  // Memory leak: 'delete[] ptr;' is missing!  }  int main() {  unsafeMemoryUsage();  return 0; // Memory is never freed, causing a leak  } |

| **Compliant Code** |
| --- |
| This version uses **std::unique\_ptr**, which ensures memory is automatically freed when it goes out of scope. |
| #include <iostream>  #include <memory>  void safeMemoryUsage() {  auto ptr = std::make\_unique<int[]>(10); // Smart pointer manages memory automatically  for (int i = 0; i < 10; i++) {  ptr[i] = i \* 2;  }  std::cout << "First element: " << ptr[0] << std::endl;  // No need to manually delete the array; unique\_ptr handles it automatically  }  int main() {  safeMemoryUsage();  return 0; // Memory is safely managed, preventing leaks  } |

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

| **Principles(s):**  Principle 3 – Applies by ensuring memory management strategies like access control are integrated into the system’s design to prevent exploitation.  Principle 9 – Applies by applying rigorous testing methods like fuzz testing and static analysis to detect and resolve memory safety issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.20.0 | Memcheck | Valgrind is an instrumentation framework for building dynamic analysis tools. Its Memcheck tool helps detect memory-related errors, assisting greatly with debugging.  Link: https://valgrind.org/ |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-008-ASSERT] | Use Assertions to Catch Logic Errors in Debug Mode – Assertions will help detect logic errors while developing. However, it is bad-practice to rely on assertions in productions, considering they’re removed in release builds often. |

| **Noncompliant Code** |
| --- |
| This example uses assertions to check user input, which should instead be handled with proper error handling. |
| #include <iostream>  #include <cassert>  void processInput(int value) {  assert(value >= 0 && value <= 100); // Unsafe: Assertions should not be used for input validation!  std::cout << "Processing value: " << value << std::endl;  }  int main() {  int userInput;  std::cout << "Enter a number (0-100): ";  std::cin >> userInput;  processInput(userInput); // If user enters -5, assertion fails and program crashes  return 0;  } |

| **Compliant Code** |
| --- |
| This example uses assertions to check internal logic and **explicit error handling for user input**. |
| #include <iostream>  #include <cassert>  void processData(int value) {  assert(value >= 0 && value <= 100); // Valid: Ensures internal logic correctness in debug mode  std::cout << "Processing value: " << value << std::endl;  }  int main() {  int userInput;  std::cout << "Enter a number (0-100): ";  std::cin >> userInput;  if (userInput < 0 || userInput > 100) {  std::cerr << "Error: Invalid input. Please enter a number between 0 and 100." << std::endl;  return 1; // Proper error handling instead of an assertion failure  }  processData(userInput); // Now, assert is used only for internal consistency checks  return 0;  } |

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

| **Principles(s):**  Principle 4 – Applies by promoting clear, meaningful assertions to verify code. This makes it easier to debug |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | Assertion usage analysis | Cppcheck is a static analysis tool that focuses on detecting undefined behavior. It can also detect improper use of assertions, ensuring they’re caught early in development.  Link: https://cppcheck.sourceforge.io/ |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-009-EXC] | Use Exceptions for Error Handling, Not Return Codes – When using exceptions for error handling, errors can be handled at a higher level, avoiding nested error logic. |

| **Noncompliant Code** |
| --- |
| This example **catches all exceptions generically (catch (...))** without handling them properly, which can lead to **silent failures** or **difficulty in debugging**. |
| #include <iostream>  void riskyFunction() {  throw std::runtime\_error("Something went wrong!"); // Throws an exception  }  int main() {  try {  riskyFunction();  } catch (...) { // Bad practice: Catches all exceptions without handling them  std::cout << "An error occurred." << std::endl; // No specific handling  }    return 0; // Execution continues without knowing what went wrong  } |

| **Compliant Code** |
| --- |
| This version properly catches **specific exceptions** and provides meaningful error handling. |
| #include <iostream>  #include <stdexcept>  void riskyFunction() {  throw std::runtime\_error("Something went wrong!"); // Throws an exception  }  int main() {  try {  riskyFunction();  } catch (const std::runtime\_error& e) { // Good practice: Catch specific exceptions  std::cerr << "Runtime error: " << e.what() << std::endl;  return 1; // Exit with a failure code  } catch (const std::exception& e) { // Catch other standard exceptions  std::cerr << "Exception: " << e.what() << std::endl;  return 1;  }  return 0; // Normal execution  } |

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

| **Principles(s):**  Principle 4 – Applies by promoting exception handling that is clear and understandable, avoiding complex logic that could introduce vulnerabilities.  Principle 9 – Applies by introducing exception-handling techniques in testing to ensure units fail knowingly. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | Exception safety checking | Cppcheck is a static analysis tool that focuses on detecting undefined behavior. It can also detect exception safety, such as proper usage of destructors to identify exception handling issues  https://cppcheck.sourceforge.io/ |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Cod Documentation and Comments | STD-011-DOC | Document Code with Clear and Concise Comments – When code is well-documented, this helps ensure developers are understanding of the system, function, class, and etc. Comments should often explain why to provide insight on design, logic, and behavior. |

| **Noncompliant Code** |
| --- |
| This example lacks proper documentation, has misleading comments, and includes unnecessary or redundant comments. |
| #include <iostream>  // This function does something  void doSomething(int x) {  x \*= 2; // Multiply x by 2  std::cout << x << std::endl; // Print x  }  int main() {  int num = 5; // Declare a variable  doSomething(num); // Call function  return 0; // Return 0  } |

| **Compliant Code** |
| --- |
| This version follows best practices with meaningful comments and **documentation** for maintainability. |
| #include <iostream>  /\*\*  \* @brief Doubles the given integer and prints the result.  \*  \* @param x The integer to be doubled.  \* @note This function does not modify the original input.  \*/  void doubleAndPrint(int x) {  int result = x \* 2;  std::cout << "Doubled value: " << result << std::endl;  }  int main() {  int num = 5; // Example input value  doubleAndPrint(num); // Print doubled value  return 0;  } |

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

| **Principles(s):**  Principle 10 – Applies by promoting consistent, well-documented coding practices. This makes it easier for developers to follow security guidelines and maintain secure software. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Doxygen | 1.13.2 | Automated documentation generator | Doxygen is a free, open-source tool that parses C++ source code and extracts documentation from comments to produce reference manuals. This will help alleviate the task of having to craft documentation from code for developers. Link: https://www.doxygen.nl/ |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Hardcoded Values | STD-012-HCV | Avoid Hardcoding Values and Use Constants or Configuration Files – When hardcoding values, the code is difficult to maintain. Therefore, the use of constants and other variables allow updates to be more seamless and flexible. |

| **Noncompliant Code** |
| --- |
| This example contains **magic numbers**, making the code **difficult to maintain and understand**. |
| #include <iostream>  void calculateCircleArea(double radius) {  double area = 3.14159 \* radius \* radius; // Hardcoded value for Pi  std::cout << "Circle area: " << area << std::endl;  }  int main() {  calculateCircleArea(5.0); // Hardcoded radius value  return 0;  } |

| **Compliant Code** |
| --- |
| This version defines **named constants** to make the code **more maintainable and readable**. |
| #include <iostream>  #include <cmath> // Use built-in math constants (C++20)  constexpr double PI = M\_PI; // Use predefined Pi constant  /\*\*  \* @brief Calculates the area of a circle.  \* @param radius The radius of the circle.  \*/  void calculateCircleArea(double radius) {  double area = PI \* radius \* radius;  std::cout << "Circle area: " << area << std::endl;  }  int main() {  constexpr double DEFAULT\_RADIUS = 5.0; // Named constant for default radius  calculateCircleArea(DEFAULT\_RADIUS);  return 0;  } |

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

| **Principles(s):**  Principle 5 – Applies by ensuring sensitive values, such as passwords, are not hardcoded. This helps reduce risk of unauthorized access if the source code is found.  Principle 6 – Applies by preventing security vulnerabilities because of values. This is by ensuring sensitive values are protected by authorized access.  Principle 10 – Applies by enforcing best practices, such as use of environment variables or secure configuration management. This helps avoid risks with hardcoded values directly in source code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | Readability-magic-numbers | Clang-Tidy is a C++ “linter” and static analysis tool based on the clang compiler. It includes the readability-magic-numbers checker, identifying hardcoded numeric values, for instance.  Link: https://clang.llvm.org/extra/clang-tidy/ |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Loops | STD-013-LOOP | Use Efficient Loop Constructs and Avoid Unnecessary Loops – Loops can lead to performance bottlenecks with applications. Therefore, it is best to use efficient loop techniques to avoid increased time complexities and redundancies. |

| **Noncompliant Code** |
| --- |
| This example demonstrates bad practices such as **off-by-one errors**, **manual index management**, and **unnecessary while loops**. |
| #include <iostream>  void printNumbers() {  int i = 0;  while (i <= 5) { // Off-by-one error: prints 6 numbers instead of 5  std::cout << "Number: " << i << std::endl;  i++;  }  }  int main() {  printNumbers();  return 0;  } |

| **Compliant Code** |
| --- |
| This version follows modern C++ best practices, using **range-based loops** for clarity and correctness. |
| #include <iostream>  #include <vector>  /\*\*  \* @brief Prints numbers from a predefined list.  \*/  void printNumbers() {  std::vector<int> numbers = {0, 1, 2, 3, 4, 5}; // Predefined sequence  for (int num : numbers) { // Range-based loop for readability and safety  std::cout << "Number: " << num << std::endl;  }  }  int main() {  printNumbers();  return 0;  } |

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

| **Principles(s):**  Principle 4 – Applies by ensuring loops are written efficiently, reducing complexity which will reduce chances of logic errors.  Principle 9 – Applies by promoting qualitive testing such as boundary testing. This ensures loops function correctly and don’t introduce vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 15.0 | Loop Analysis | The Clang Static Analyzer is a source code analysis tool that finds bugs in C and C\_\_ programs, which includes loop-related issues.  Link: https://clang.llvm.org/extra/clang-tidy/ |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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.

For the assess and plan phase, thread modeling, risk assessment, and regulatory compliance checks can be implemented. We’d use the CI/CD pipeline integration to flag security risks in the backlog, especially for policies like least privilege. For production, during the monitor and detect level, we could update this part of the DevSecOps diagram by automating intrusion detection and integrating thread intelligence feeds to better ensure we’re ahead of threat indication. That way, we’d reduce the time it takes to remove malicious attackers from our systems.

### 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-ABC] | High | High | Medium | High | 5 |
| [STD-004-DIV] | High | High | Medium | High | 5 |
| [STD-005-STR] | High | High | Medium | High | 5 |
| [STD-006-SQL] | High | High | Medium | High | 5 |
| [STD-007-MEM] | High | High | High | High | 5 |
| [STD-008-ASSERT] | Medium | Medium | Low | Medium | 3 |
| [STD-009-EXC] | Medium | Medium | Medium | Medium | 3 |
| STD-011-DOC | Low | Medium | Low | Low | 2 |
| STD-012-HCV | High | High | Medium | High | 5 |
| STD-013-LOOP | Medium | Medium | Low | Medium | 3 |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### Create Policies for Efffffncryption anfd 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, hfffow 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 | This involves encryption of stored data and using strong algorithms to protect against unauthorized. This adheres to the principle of least privilege by ensuring only authorized users can decrypt sensitive data. |
| Encryption in flight | Secures data transmitted over networks using TLS/SSL to prevent interception. And sanitize data sent to other systems adhere to this. |
| Encryption in use | Protects data by actively being processed in memory using techniques. Defense in Depth applies to this by adding multiple levels of security |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | [Insert text.] |
| Authorization | [Insert text.] |
| Accounting | [Insert text.] |

**\***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 |  |
| 2.0 | 02/02/2025 | Coding Standards | Malik Spruill | Mimi Tam |
| 3.0 | 02/23/2025 | Security Policy | Malik Spruill | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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