**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 | Input validation ensures that all data entering a system is verified for correctness, format, and integrity before processing. This prevents injection attacks, buffer overflows, and other vulnerabilities caused by malicious or unexpected input. |
| 1. Heed Compiler Warnings | Compiler warnings often highlight potential issues such as deprecated functions, insecure practices, or undefined behavior in code. Addressing these warnings proactively can prevent vulnerabilities from being introduced during development. |
| 1. Architect and Design for Security Policies | Security must be embedded into the application architecture and design. This involves identifying potential attack surfaces, applying secure design principles (e.g., secure defaults), and aligning system design with established security policies. |
| 1. Keep It Simple | Complex systems are harder to secure due to increased attack surfaces and difficulty in understanding interdependencies. Following the KISS (Keep It Simple, Stupid) principle ensures that security mechanisms are straightforward and less error-prone. |
| 1. Default Deny | Systems and applications should deny access by default and only allow explicitly authorized actions. This principle reduces exposure to unauthorized access and minimizes the risk of exploitation. |
| 1. Adhere to the Principle of Least Privilege | Users, processes, and applications should be granted only the permissions necessary to perform their functions. This reduces the potential damage caused by compromised accounts or processes. |
| 1. Sanitize Data Sent to Other Systems | Data leaving an application should be sanitized to ensure it does not carry harmful payloads or unintended information. This prevents vulnerabilities such as cross-site scripting (XSS) or SQL injection in connected systems. |
| 1. Practice Defense in Depth | Implement multiple layers of security to provide redundancy in case one mechanism fails. This strategy ensures that even if one layer is compromised, others remain in place to protect critical assets. |
| 1. Use Effective Quality Assurance Techniques | Regular code reviews, automated testing, static analysis, and dynamic testing help identify vulnerabilities early in the development lifecycle. This ensures software quality and reduces the risk of security flaws reaching production. |
| 1. Adopt a Secure Coding Standard | Follow industry-recognized secure coding standards, such as CERT or MISRA for C/C++, to reduce vulnerabilities inherent in unsafe coding practices. Secure standards guide developers in writing robust, maintainable, and 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** | **Avoid Implicit Conversions Between Data Types** |
| --- | --- | --- |
| **Data Type** | [STD-001-DAT] | Implicit conversions between data types can lead to data loss, unexpected results, or undefined behavior. Explicit conversions ensure clarity and avoid unintentional logic errors. |

| **Noncompliant Code** |
| --- |
| Implicit conversions occur when a value of one data type is automatically converted to another, potentially leading to issues such as data truncation or unintended rounding. |
| int calculateTotal(double value) {  return value; // Implicit conversion from double to int  } |

| **Compliant Code** |
| --- |
| Explicitly converting the data type using static\_cast or equivalent ensures program correctness and minimizes errors. |
| int calculateTotal(double value) {  return static\_cast<int>(value); // Explicit conversion with static\_cast  } |

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

| **Principles(s):**  **Type Safety** – Ensuring explicit type conversions improves code reliability by preventing unintended data loss or logic errors caused by implicit conversions.  **Code Readability & Maintainability** – Explicit conversions make it clear when data types are being changed, reducing debugging time and increasing maintainability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | Latest | cppcoreguidelines-narrowing-conversions | Detects implicit narrowing conversions |
| GCC Compiler | 10+ | -Wconversion | Warns about implicit conversions that may alter a value |
| CppCheck | Latest | inconclusive | Identifies potential type conversion issues |
| PVS-Studio | Latest | V1065 | Detects implicit conversions that may cause precision loss |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Use Constants for Fixed Data Values** |
| --- | --- | --- |
| **Data Value** | STD-002-DAT] | Using constants for fixed data values, such as configuration values, thresholds, and magic numbers, makes the code more maintainable. It ensures that any change to the value only requires an update in one place, reducing the risk of errors and improving clarity. Constants also provide context for the value, improving code readability. |

| **Noncompliant Code** |
| --- |
| Hardcoding fixed values throughout the code can lead to maintenance challenges and bugs when values need to be updated. |
| void calculateDiscount() {  int discount = 5; // Hardcoded discount value  totalAmount -= discount;  } |

| **Compliant Code** |
| --- |
| Using a constant for the fixed value makes the code easier to maintain and understand. |
| const int DISCOUNT\_RATE = 5; // Constant for discount value  void calculateDiscount() {  totalAmount -= DISCOUNT\_RATE;  } |

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

| **Principles(s):**  **Maintainability** – Using constants prevents hardcoded values from being scattered across the codebase, making future changes easier.  **Readability** – Constants provide meaningful names, making the code self-explanatory. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | Latest | readability-magic-numbers | Identifies hardcoded magic numbers |
| CppCheck | Latest | performance-magicnumbers | Detects magic numbers that should be replaced with constants |
| SonarQube | Latest | S109 | Warns about the presence of hardcoded values |
| PVS-Studio | Latest | V112 | Detects numerical literals that should be replaced with named constants |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Proper Use of String Manipulations** |
| --- | --- | --- |
| **String Correctness** | [STD-003-STR] | Correct string handling is essential to avoid issues such as data corruption, errors, and security vulnerabilities like buffer overflows or injection attacks. Ensuring that string manipulations follow best practices improves both code reliability and performance. Additionally, handling edge cases like empty strings or null values is critical for the robustness of the program. |

| **Noncompliant Code** |
| --- |
| Using improper string operations, such as direct concatenation without validation, can lead to issues like buffer overflows or incorrect string formation. |
| char message[10];  strcpy(message, "Hello");  strcat(message, "World!"); // This causes a buffer overflow, as the message size exceeds the buffer |

| **Compliant Code** |
| --- |
| Properly handling string lengths and using safer functions helps prevent buffer overflows and other string-related issues. |
| #define MAX\_MESSAGE\_LENGTH 20  char message[MAX\_MESSAGE\_LENGTH];  strncpy(message, "Hello", MAX\_MESSAGE\_LENGTH);  strncat(message, " World!", MAX\_MESSAGE\_LENGTH - strlen(message) - 1); // Safe concatenation |

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

| **Principles(s):**  **Defense in Depth**   * This principle relates to using multiple layers of protection to prevent failures. By validating the size of strings and checking buffer limits, the code avoids potential issues such as buffer overflows or memory corruption, thus enhancing the security and stability of the program.   **Fail Safe Defaults**   * This principle emphasizes building systems that default to safe behavior when faced with errors. By using safe string manipulation functions, the code ensures that if an error occurs (e.g., insufficient buffer space), the system defaults to a secure behavior without risking crashes or data loss. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | Latest | inconclusive | Detects unsafe buffer handling and string manipulation issues. |
| Clang-Tidy | Latest | bugprone-buffer-overflow | Detects unsafe buffer manipulations, especially buffer overflows. |
| SonarQube | Latest | bugprone-string-literal | Detects risky string operations that could lead to buffer overflows. |
| PVS-Studio | Latest | V6090 | Warns about unsafe string manipulations that could lead to vulnerabilities. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Protection Against SQL Injection** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-SQL] | SQL injection is a critical vulnerability that allows an attacker to manipulate SQL queries by inserting malicious code into them. This can lead to unauthorized access, data modification, and even data destruction. To prevent SQL injection, always use prepared statements, parameterized queries, and ORM tools that handle query composition securely. Avoid concatenating user inputs directly into SQL statements. |

| **Noncompliant Code** |
| --- |
| Directly embedding user input into SQL queries without validation exposes the system to SQL injection attacks. |
| <?php  $user = $\_GET['user'];  $password = $\_GET['password'];  $query = "SELECT \* FROM users WHERE username = '$user' AND password = '$password'"; // Vulnerable to SQL Injection  $result = mysqli\_query($conn, $query);  ?> |

| **Compliant Code** |
| --- |
| Using parameterized queries or prepared statements eliminates the risk of SQL injection by ensuring that user inputs are treated as data, not executable code. |
| <?php  $user = $\_GET['user'];  $password = $\_GET['password'];  $query = "SELECT \* FROM users WHERE username = ? AND password = ?";  $stmt = mysqli\_prepare($conn, $query);  mysqli\_stmt\_bind\_param($stmt, 'ss', $user, $password); // Parameters are bound and input is treated as data  mysqli\_stmt\_execute($stmt);  $result = mysqli\_stmt\_get\_result($stmt);  ?> |

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

| **Principles(s):**  **Security** – Preventing SQL injection is crucial for safeguarding databases from unauthorized access and data manipulation.<br>**Maintainability** – Using prepared statements and parameterized queries improves maintainability by avoiding ad-hoc query construction.<br>**Robustness** – Proper query protection ensures that user inputs are safely handled, making the system more resilient to malicious attempts. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | Latest | S1098 | Warns about SQL injection vulnerabilities in query construction. |
| Checkmarx | Latest | CxSQLInjection | Detects SQL injection vulnerabilities and insecure query practices. |
| Fortify | Latest | SQLInjection | Identifies SQL injection vulnerabilities in queries. |
| Brakeman | Latest | SQLInjection | Scans for SQL injection vulnerabilities in Ruby on Rails applications |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Memory Protection** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-MEM] | Memory protection is essential to prevent unauthorized access, modification, or corruption of memory. It helps ensure that each program or user only has access to their own memory space, reducing the risk of unintended or malicious behavior that can lead to vulnerabilities like buffer overflows, data corruption, and crashes. In critical systems, enforcing memory protection can protect against exploits, ensuring system stability and security. |

| **Noncompliant Code** |
| --- |
| The following example uses unsafe memory access, potentially leading to buffer overflows and memory corruption by writing beyond the array's bounds. This is a vulnerability that can be exploited by attackers to gain access to sensitive memory regions. |
| #include <stdio.h>  void unsafe\_memory\_access() {  char buffer[10];  printf("Enter some text: ");  gets(buffer); // Unsafe: gets does not check buffer bounds  printf("You entered: %s\n", buffer);  } |

| **Compliant Code** |
| --- |
| This example uses fgets() to ensure that input does not exceed the buffer size, thus preventing buffer overflows. Proper memory bounds are enforced, improving security. |
| #include <stdio.h>  void safe\_memory\_access() {  char buffer[10];  printf("Enter some text: ");  fgets(buffer, sizeof(buffer), stdin); // Safe: fgets checks buffer bounds  printf("You entered: %s\n", buffer);  } |

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

| **Principles(s):**   **Security** – Prevents unauthorized access and exploitation of memory-related vulnerabilities.   **Reliability** – Ensures system stability by avoiding crashes due to invalid memory access.   **Robustness** – Safe memory handling minimizes unexpected failures. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | Latest | memory-protection | Detects unsafe memory operations and out-of-bounds accesses. |
| SonarQube | Latest | cpp:S1076 | Detects memory safety violations like buffer overflows. |
| PVS-Studio | Latest | V3035 | Finds dangerous memory manipulations. |
| Coverity | Latest | Memory corruption detection | Identifies risks like buffer overflows and invalid memory accesses. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Assertions** |
| --- | --- | --- |
| **Assertions** | [STD-006-ASR] | Assertions are used to validate assumptions made during program execution. They help in detecting bugs early in development by verifying that certain conditions hold true. However, they should not replace proper error handling in production code. Assertions are useful for catching programming errors, especially in the development and debugging stages, but they must be disabled in release builds to avoid unnecessary overhead and to prevent security issues. |

| **Noncompliant Code** |
| --- |
| The following example uses assertions incorrectly in a production environment, which could introduce unwanted performance overhead or expose sensitive program details. Additionally, the assertion is misused for error handling, which should be done through proper exception handling instead. |
| #include <assert.h>  void process\_data(int value) {  assert(value >= 0); // Noncompliant: Assertions shouldn't be used for runtime checks in production  printf("Processing value: %d\n", value);  } |

| **Compliant Code** |
| --- |
| In this compliant version, we use assertions in the development environment to validate assumptions, and the actual error handling is done with a proper conditional check. Assertions are disabled in production builds by defining NDEBUG. |
| #include <assert.h>  #include <stdio.h>  void process\_data(int value) {  assert(value >= 0); // Compliant: Used for debugging purposes only, not in production  if (value < 0) {  printf("Error: Negative value not allowed\n");  return;  }  printf("Processing value: %d\n", value);  } |

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

| **Principles(s):**   **Fail Safe Defaults** – Assertions ensure that critical assumptions are met before proceeding.   **Debugging** – Helps catch programming errors before deployment.   **Performance Optimization** – Avoids unnecessary performance overhead by disabling assertions in production. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | Latest | assert-checking | Identifies improper use of assertions in production. |
| SonarQube | Latest | cpp:S1071 | Warns about assertions left in production code. |
| CppCheck | Latest | performance-assert | Detects unnecessary assertions in non-debug builds. |
| Coverity | Latest | Assertion misuse | Finds assertions used incorrectly in error handling. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-EXC] | Exceptions are a powerful mechanism for handling runtime errors, but they should be used judiciously. They allow for centralized error handling and can help keep the normal code flow clean. However, exceptions should not be used for regular control flow or expected conditions. Exceptions should only be thrown for truly exceptional or unexpected events. Additionally, catching general exceptions or swallowing exceptions without proper handling can lead to hidden issues that make debugging difficult. Properly classifying and handling different types of exceptions ensures that the program can respond appropriately to errors without compromising readability or stability. |

| **Noncompliant Code** |
| --- |
| The following code uses exceptions improperly by throwing them for expected conditions or using a general exception type without specific error handling, which can cause performance overhead and poor error management. |
| def divide(a, b):  try:  result = a / b  except Exception as e: # Noncompliant: Catching a general exception  print("An error occurred:", e)  return None  return result |

| **Compliant Code** |
| --- |
| In this compliant version, we handle the specific error (e.g., division by zero) rather than using a general exception. The exception is only raised for truly exceptional cases, and meaningful error handling is provided. |
| def divide(a, b):  if b == 0:  print("Error: Division by zero.")  return None  return a / b |

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

| **Principles(s):**   **Error Management** – Ensures exceptions are used correctly for handling critical failures.   **Maintainability** – Helps create structured error-handling workflows.   **Performance** – Avoids unnecessary overhead from improper exception usage. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | Latest | exception-escape | Identifies functions that throw exceptions unexpectedly. |
| SonarQube | Latest | exception:S1071 | Detects unhandled exceptions and missing try-catch blocks. |
| CppCheck | Latest | exception-usage | Warns about misuse of exception handling. |
| Coverity | Latest | Exception handling violations | Ensures proper error-handling techniques are followed. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Code Documentation** |
| --- | --- | --- |
| Documentation | [STD-008-DOC] | Code documentation is crucial for maintaining and scaling software systems. Well-documented code allows others (and your future self) to easily understand the purpose, usage, and behavior of your code. This includes writing clear comments for complex or non-obvious logic and providing docstrings for functions, classes, and modules. Code should not rely on the assumption that others will figure out what the code does—proper documentation ensures maintainability, collaboration, and debugging efficiency. Additionally, keeping documentation up to date as the code evolves is equally important. |

| **Noncompliant Code** |
| --- |
| The following code lacks sufficient documentation. It is unclear what the function does, what parameters it expects, or what it returns. This could lead to confusion and difficulty for other developers trying to work with this code in the future. |
| #include <iostream>  int process(int a, int b) {  int c = a + b;  return c;  }  int main() {  int result = process(5, 10);  std::cout << "Result: " << result << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant version, we've added a docstring that describes the function's purpose, parameters, and return value. This improves readability and ensures that the function is properly understood by others. |
| #include <iostream>  /\*\*  \* Adds two numbers together and returns the result.  \*  \* @param a The first number to be added.  \* @param b The second number to be added.  \* @return The sum of a and b.  \*/  int process(int a, int b) {  int c = a + b;  return c;  }  int main() {  int result = process(5, 10);  std::cout << "Result: " << result << std::endl;  return 0;  } """  c = a + b  return c |

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

| **Principles(s):**   **Maintainability** – Ensures that the code remains understandable for future developers.   **Readability** – Improves comprehension by clearly explaining logic and function purpose.   **Collaboration** – Facilitates teamwork by providing well-documented reference points. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Doxygen | Latest | Comment checking | Ensures proper documentation structure. |
| SonarQube | Latest | documentation:S1103 | Identifies missing or incomplete comments. |
| Pylint | Latest | docstring-checker | Ensures functions have proper docstrings. |
| Clang-Tidy | Latest | readability-docs | Warns about undocumented functions. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Code Readability and Consistency** |
| --- | --- | --- |
| Code Quality | [STD-009-UNC] | Unused code (such as variables, functions, or imports that are never used) adds unnecessary complexity, can lead to confusion, and increases the size of the codebase without providing any functionality. Removing unused code helps keep the code clean, efficient, and maintainable. It also reduces the potential for bugs and helps other developers focus on the essential parts of the code. |

| **Noncompliant Code** |
| --- |
| The following code includes an unused variable and an unused import statement that clutter the codebase. |
| #include <iostream>  #include <cmath> // Unused import  void calculate\_area(double radius) {  double area = 3.14 \* radius \* radius;  int unused\_var = 10; // Unused variable  std::cout << "Area: " << area << std::endl;  }  int main() {  calculate\_area(5);  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant version, the unused import and variable are removed, making the code cleaner and more efficient. |
| #include <iostream>  void calculate\_area(double radius) {  double area = 3.14 \* radius \* radius; // Calculate the area of a circle  std::cout << "Area: " << area << std::endl;  }  int main() {  calculate\_area(5);  return 0;  } |

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

| **Principles(s):**   **Code Quality** – Clean, consistent code is easier to debug and maintain.   **Error Reduction** – Eliminating unused variables and improving formatting helps prevent hidden bugs.   **Performance Optimization** – Removing unnecessary code reduces processing overhead. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Format | Latest | formatting-checker | Ensures consistent code formatting. |
| SonarQube | Latest | readability:S2006 | Identifies readability issues. |
| ESLint | Latest | unused-vars | Flags unused variables and functions. |
| CppCheck | Latest | style-check | Detects inconsistent coding styles. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Variable Declaration and Initialization** |
| --- | --- | --- |
| Variable Management | [STD-010-VDI] | Variable declaration and initialization should be clear and concise to prevent unexpected behavior and confusion. Variables should be declared close to their point of use, and initialized properly to avoid errors related to undefined or null values. Consistent and clear variable naming conventions are essential for readability and maintainability. |

| **Noncompliant Code** |
| --- |
| The following code contains poorly declared and initialized variables, leading to potential issues. |
| #include <iostream>  double calculate\_discount(double price, double discount\_percentage) {  double disc = discount\_percentage / 100; // Discount percentage as a decimal  double discount\_price = price - (price \* disc);  double p = price; // Unclear variable name  return discount\_price;  }  int main() {  double discounted\_price = calculate\_discount(200, 15);  std::cout << "Discounted Price: " << discounted\_price << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant version, the variables are more clearly declared and initialized with meaningful names. |
| #include <iostream>  double calculate\_discount(double price, double discount\_percentage) {  double discount\_as\_decimal = discount\_percentage / 100; // Convert discount percentage to decimal  double discounted\_price = price - (price \* discount\_as\_decimal); // Calculate the final discounted price  return discounted\_price;  }  int main() {  double discounted\_price = calculate\_discount(200, 15);  std::cout << "Discounted Price: " << discounted\_price << std::endl;  return 0;  } |

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

| **Principles(s):**   **Code Clarity** – Ensures variables are meaningful and easy to understand.   **Error Prevention** – Prevents usage of uninitialized or undefined values.   **Maintainability** – Helps reduce confusion and makes code easier to modify. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | Latest | bugprone-use-after-move | Detects usage of uninitialized variables. |
| SonarQube | Latest | cpp:S5433 | Ensures proper variable naming conventions. |
| CppCheck | Latest | style-variable | Flags variables declared without initialization. |
| PVS-Studio | Latest | V606 | Identifies redundant or improperly declared variables. |

### 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.

Using the DevSecOps diagram as context, security automation should be embedded into the following stages:

1. **Plan:** Automated security risk assessments should be conducted before development begins. This includes setting coding standards, defining secure configurations, and integrating security best practices.
2. **Develop:** Developers should use automated code analysis tools such as SonarQube, Clang-Tidy, and CppCheck to identify security vulnerabilities early. Pre-commit hooks should enforce compliance with coding standards.
3. **Build:** Automated security scanning tools should be integrated into CI/CD pipelines. Tools like Coverity and PVS-Studio should scan for code vulnerabilities before builds are deployed.
4. **Test:** Static and dynamic application security testing (SAST/DAST) should be automated to detect security weaknesses. Automated unit tests should validate compliance with secure coding practices.
5. **Release:** Security policies should enforce automated vulnerability scanning before release. Infrastructure-as-code (IaC) templates should be checked for misconfigurations.
6. **Deploy:** Automated runtime security checks and container security scanning (e.g., using Aqua Security or Sysdig Secure) should be integrated to prevent deploying insecure configurations.
7. **Operate & Monitor:** Automated monitoring solutions should track security events, log anomalies, and detect unauthorized access attempts. Security information and event management (SIEM) tools should provide real-time threat detection.
8. **Respond:** Automated incident response workflows should be triggered when security breaches occur. Automated rollback and remediation mechanisms should minimize damage.

By embedding security into each phase, Green Pace can ensure continuous compliance and proactive risk mitigation.

### 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-DAT | Medium | High | Low | High | 3 |
| STD-003-STR | High | Moderate | Low | High | 5 |
| STD-004-SQL | High | High | Low | High | 5 |
| STD-005-MEM | High | High | Medium | Highest | 5 |
| STD-006-ASR | Medium | Medium | Low | Medium | 3 |
| STD-007-EXC | Medium | High | Medium | High | 4 |
| STD-008-DOC | Medium | Medium | Low | Medium | 3 |
| STD-009-UNC | Low | Medium | Low | Medium | 2 |
| STD-010-VDI | Medium | Medium | Low | Medium | 3 |
| STD-002-DAT | Medium | High | Low | High | 3 |
| STD-003-STR | High | Moderate | Low | High | 5 |
| STD-004-SQL | High | High | Low | High | 5 |
| STD-005-MEM | High | High | Medium | Highest | 5 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Data stored on disk, databases, and backup systems must be encrypted using AES-256 encryption. This prevents unauthorized access in case of physical theft or unauthorized system access. Applies to databases, log files, and configuration files. |
| Encryption in flight | Data transmitted over networks should be encrypted using TLS 1.2 or higher. This ensures confidentiality and integrity during transmission, preventing eavesdropping and man-in-the-middle attacks. Applies to API communications, web applications, and internal/external network traffic. |
| Encryption in use | |  |  | | --- | --- | |  | Sensitive data actively processed in memory should be encrypted when possible using hardware security modules (HSMs) or secure enclaves. This prevents exposure of sensitive data to unauthorized applications. Applies to financial transactions, authentication credentials, and cryptographic operations. | |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Ensures that users and systems verify their identities before accessing resources. Multi-factor authentication (MFA) must be enforced for all user logins, with password policies requiring strong credentials. Applies to login systems, APIs, and privileged accounts. |
| Authorization | |  |  | | --- | --- | |  | Defines user roles and permissions to restrict access based on the principle of least privilege. Users should only have access to the data and functions necessary for their roles. Applies to database access, file permissions, and administrative privileges. | |
| Accounting | Logs and monitors all access and modification activities to detect anomalies and ensure compliance. Audit logs must track logins, user activity, file access, and configuration changes. Applies to security monitoring, incident response, and compliance auditing. |

**\***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

|  |  |  |
| --- | --- | --- |
| Rule | Applicable Principles | Justification |
| STD-001-CPP | 1, 3, 5 | Ensures type safety, maintainability, and error prevention by requiring explicit type conversions. |
| STD-002-DAT | 2, 4, 6 | Enforces maintainability and readability by requiring constant use for fixed data values. |
| STD-003-STR | 1, 7, 8 | Prevents buffer overflows and ensures secure string manipulations. |
| STD-004-SQL | 3, 5, 9 | Protects against SQL injection and unauthorized query execution. |
| STD-005-MEM | 1, 4, 10 | Ensures secure memory allocation and prevents unauthorized access. |
| STD-006-ASR | 2, 5, 8 | Encourages effective debugging without affecting production performance. |
| STD-007-EXC | 3, 6, 9 | Promotes proper error handling while maintaining performance. |
| STD-008-DOC | 2, 6, 10 | Supports maintainability and clear documentation of system logic. |
| STD-009-UNC | 4, 7, 9 | Improves readability by removing unused code and enforcing consistency. |
| STD-010-VDI | 1, 3, 6 | Requires clear variable declarations for better maintainability and debugging. |

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 | 02/20/2025 | Completed Project 2 requirements | Caleb Leavell | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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