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



# Green Pace Secure Development Policy

## Contents

[Green Pace Secure Development Policy 1](#_Toc141114030)

[Contents 1](#_Toc338079405)

[Overview 2](#_Toc336030590)

[Purpose 3](#_Toc1937706942)

[Scope 3](#_Toc1222606289)

[Module Three Milestone 3](#_Toc850950876)

[Ten Core Security Principles 3](#_Toc1023568255)

[C/C++ Ten Coding Standards 4](#_Toc271599446)

[Coding Standard 1 4](#_Toc270483958)

[Coding Standard 2 6](#_Toc376831951)

[Coding Standard 3 8](#_Toc1052400352)

[Coding Standard 4 10](#_Toc1874375587)

[Coding Standard 5 13](#_Toc31856905)

[Coding Standard 6 15](#_Toc964384038)

[Coding Standard 7 17](#_Toc1050421516)

[Coding Standard 8 19](#_Toc600136068)

[Coding Standard 9 20](#_Toc906412291)

[Coding Standard 10 21](#_Toc264283487)

[Defense-in-Depth Illustration 22](#_Toc1532670753)

[Project One 23](#_Toc1197168986)

[Revise the C/C++ Standards 23](#_Toc1129304035)

[Risk Assessment 23](#_Toc122210549)

[Automated Detection 23](#_Toc620894048)

[Automation 23](#_Toc1274163917)

[Summary of Risk Assessments 24](#_Toc1243705234)

[Create Policies for Encryption and Triple A 24](#_Toc1424773527)

[Map the Principles 25](#_Toc1278125265)

[Audit Controls and Management 26](#_Toc410785568)

[Enforcement 26](#_Toc1378427306)

[Exceptions Process 26](#_Toc2084141471)

[Distribution 26](#_Toc378904852)

[Policy Change Control 27](#_Toc130076943)

[Policy Version History 27](#_Toc678212756)

[Appendix A Lookups 27](#_Toc2050818641)

[Approved C/C++ Language Acronyms 27](#_Toc1106612066)

## 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 | All input data must be validated to prevent security vulnerabilities such as buffer overflows, SQL injection, and cross-site scripting (XSS). |
| 1. Heed Compiler Warnings | Compiler warnings must be treated as errors and resolved before deploying the code to production. |
| 1. Architect and Design for Security Policies | Systems should be designed with security principles from the outset to ensure robustness and resilience. |
| 1. Keep It Simple | Complex security mechanisms often introduce vulnerabilities. Simplicity should be prioritized in security design. |
| 1. Default Deny | By default, access to all resources should be denied unless explicitly permitted. |
| 1. Adhere to the Principle of Least Privilege | Users and systems should only have the necessary privileges required to perform their tasks. |
| 1. Sanitize Data Sent to Other Systems | Data sent to external systems must be sanitized to prevent security breaches and vulnerabilities. |
| 1. Practice Defense in Depth | Multiple layers of security controls should be implemented to protect critical systems. |
| 1. Use Effective Quality Assurance Techniques | Security testing and static code analysis should be employed to identify vulnerabilities. |
| 1. Adopt a Secure Coding Standard | Following industry-approved secure coding standards ensures software integrity and reliability. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Using incorrect or mismatched data types can lead to security vulnerabilities such as buffer overflows, unintended type conversions, and memory corruption. Ensuring proper data type usage enhances code robustness, prevents undefined behavior, and reduces security risks in C++ programs. |

| **Noncompliant Code** |
| --- |
| The following code demonstrates an implicit conversion issue where an int is mistakenly used as a char, leading to unpredictable behavior. |
| void processData(char value) {  std::cout << "Processing value: " << value << std::endl;  }    int main() {  int input = 256; // Exceeds char range  processData(input); // Implicit conversion to char  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant version, explicit type checking is used, and the proper data type is ensured before passing the value. |
| void processData(char value) {  std::cout << "Processing value: " << static\_cast<int>(value) << std::endl;  }  int main() {  int input = 256;  if (input >= std::numeric\_limits::min() && input <= std::numeric\_limits::max()) {  processData(static\_cast<char>(input)); // Explicit conversion with validation  } else {  std::cerr << "Error: Input value out of char range." << std::endl;  }  return 0;  } |

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

| **Principles(s):** Validate Input Data, Use Effective Quality Assurance Techniques, Adopt a Secure Coding Standard |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10+ | typeMismatch | Detects implicit conversions between incompatible data types that may cause truncation, overflow, or logic errors. |
| Clang-Tidy | 17+ | Cert-dc150-cpp | Flags improper declarations that may result in unintended type usage or memory issues. |
| SonarQube | latest | Cpp:S1192 | Flags improper declarations that may result in unintended type usage or memory issues. |
| Visual Studio | 2022+ | C++ core check | Built-in static analyzer that identifies type errors and incorrect conversions early in the build process. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Using uninitialized or improperly validated data can lead to undefined behavior, security vulnerabilities, or logic errors. Ensuring proper initialization and validation of variables prevents unintended outcomes, crashes, and potential exploits. |

| **Noncompliant Code** |
| --- |
| The following code demonstrates an issue where a variable is used without being initialized, leading to undefined behavior. |
| void printValue() {  int value; // Uninitialized variable  std::cout << "Value: " << value << std::endl; // Undefined behavior  }    int main() {  printValue();  return 0;  } |

| **Compliant Code** |
| --- |
| The compliant version ensures that variables are always initialized before use. |
| void printValue() {  int value = 0; // Properly initialized variable  std::cout << "Value: " << value << std::endl;  }    int main() {  printValue();  return 0;  } |

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

| **Principles(s):**  **Validate Input Data**: Ensures that all values used in the program are checked before they are processed. Prevents undefined behavior from uninitialized or out-of-bound values.  **Use Effective Quality Assurance Techniques**: Reinforces the importance of static and dynamic code analysis to catch uninitialized variables or improper data usage.  **Adopt a Secure Coding Standard**: Aligns with SEI CERT C++ rules like EXP53-CPP and EXP54-CPP, which recommend initializing all variables and respecting object lifetimes. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10+ | Uninitvar, unusedFunction | Detects uninitialized memory reads and improper object lifetime access. |
| Clang-Tidy | 17+ | Cert-exp53-cpp, cert-exp54-cpp | Flags variables used before initialization and unused logic that may be flawed. |
| SonarQube | latest | Cpp:S1481, cpp:S1172 | Detects uninitialized, unused, or redundant variables that may indicate logic errors. |
| Visual Studio | 2022+ | Code Analysis - Ruleset Security | Built-in analyzer to catch uninitialized or improperly scoped variables. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Improper handling of strings in C++ can lead to buffer overflows, memory corruption, and security vulnerabilities such as null-termination issues. Ensuring safe string handling by using standard C++ string libraries rather than raw C-style strings mitigates these risks. |

| **Noncompliant Code** |
| --- |
| The following code demonstrates unsafe string handling using C-style character arrays, leading to a potential buffer overflow. |
| #include <iostream>  #include <cstring>    void unsafeFunction(const char\* input) {  char buffer[10]; // Fixed-size buffer  std::strcpy(buffer, input); // No bounds checking (buffer overflow risk)  std::cout << "Buffer content: " << buffer << std::endl;  }    int main() {  const char\* largeInput = "This is a very long input string!";  unsafeFunction(largeInput); // Overflow occurs here  return 0;  } |

| **Compliant Code** |
| --- |
| The compliant version uses std::string to ensure safe string handling and avoid buffer overflows. |
| #include <iostream>  #include <string>    void safeFunction(const std::string& input) {  std::string buffer = input.substr(0, 9); // Safe substring extraction  std::cout << "Buffer content: " << buffer << std::endl;  }    int main() {  std::string largeInput = "This is a very long input string!";  safeFunction(largeInput); // Proper handling of long strings  return 0;  } |

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

| **Principles(s):**  **Sanitize Data Sent to Other Systems**: Prevents buffer overflows and format string attacks by ensuring strings are validated and bounded.  **Use Effective Quality Assurance Techniques**: Encourages static analysis to catch unsafe C-style string operations and improper use of string functions.  **Adopt a Secure Coding Standard**: Reinforces the use of safe string handling via std::string and aligns with SEI CERT rules like STR50-CPP and STR51-CPP. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17+ | Cert-str50-cpp, Cert-str51-cpp | Detects unsafe uses of C-style strings and null pointer risks in std::string |
| Cppcheck | 2.10+ | bufferOverrun | Flags unsafe string copies and potential overflows in fixed-size buffers |
| SonarQube | latest | Cpp:S3514,cppS1751 | Highlights risky string manipulations, especially unsafe buffer copying. |
| VS | 2022+ | C++ code analysis | [Insert text.] Flags unsafe string functions like strcpy, strcat, and non-null-terminated issues |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | SQL injection is a critical security vulnerability that allows attackers to manipulate SQL queries, potentially exposing or modifying database information. Using parameterized queries or prepared statements prevents injection attacks by separating user input from the query structure. |

| **Noncompliant Code** |
| --- |
| The following code constructs an SQL query using string concatenation with user input, making it vulnerable to SQL injection. |
| void executeQuery(sqlite3\* db, const std::string& userInput) {  std::string query = "SELECT \* FROM users WHERE username = '" + userInput + "';";  char\* errorMessage = nullptr;  if (sqlite3\_exec(db, query.c\_str(), nullptr, nullptr, &errorMessage) != SQLITE\_OK) {  std::cerr << "SQL error: " << errorMessage << std::endl;  sqlite3\_free(errorMessage);  }  }  int main() {  sqlite3\* db;  sqlite3\_open("example.db", &db);    std::string userInput;  std::cout << "Enter username: ";  std::getline(std::cin, userInput);    executeQuery(db, userInput);  sqlite3\_close(db);  return 0;  } |

| **Compliant Code** |
| --- |
| The compliant version uses prepared statements to safely handle user input and prevent SQL injection. |
| void executeQuery(sqlite3\* db, const std::string& userInput) {  const char\* query = "SELECT \* FROM users WHERE username = ?;";  sqlite3\_stmt\* stmt;  if (sqlite3\_prepare\_v2(db, query, -1, &stmt, nullptr) == SQLITE\_OK){  sqlite3\_bind\_text(stmt, 1, userInput.c\_str(), -1, SQLITE\_STATIC);   while (sqlite3\_step(stmt) == SQLITE\_ROW) {  std::cout << "User found: " << sqlite3\_column\_text(stmt, 0) << std::endl;  }  sqlite3\_finalize(stmt);  } else {  std::cerr << "Failed to prepare statement." << std::endl;  }  }  int main() {  sqlite3\* db;  sqlite3\_open("example.db", &db);    std::string userInput;  std::cout << "Enter username: ";  std::getline(std::cin, userInput);  executeQuery(db, userInput);  sqlite3\_close(db);  return 0;  } |

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

| **Principles(s):**   * **Validate Input Data**: Ensures all user-supplied data is checked before it is included in SQL queries. * **Sanitize Data Sent to Other Systems**: Prevents malicious commands from being interpreted as part of the SQL logic. * **Architect and Design for Security Policies**: Enforces use of prepared statements and parameterized queries in application design to isolate data from code logic. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | Latest | Cpp:S3649 | Detects dynamic SQL queries and highlights unsafe string concatenation patterns. |
| [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-005- CPP] | Improper memory management can lead to security vulnerabilities such as buffer overflows, use-after-free, dangling pointers, and memory leaks. Ensuring proper allocation, deallocation, and bounds checking prevents security risks and enhances program stability. |

| **Noncompliant Code** |
| --- |
| The following code demonstrates a memory leak where dynamically allocated memory is not freed. |
| void allocateMemory() {  int\* ptr = new int[10]; // Dynamically allocated array  ptr[0] = 42;  std::cout << "First element: " << ptr[0] << std::endl;  // Memory leak: No delete[] call  }    int main() {  allocateMemory();  return 0;  } |

| **Compliant Code** |
| --- |
| The compliant version ensures that memory is properly managed using smart pointers (std::unique\_ptr and std::vector). |
| void allocateMemory() {  std::vector<int> vec(10); // Automatic memory management  vec[0] = 42;  std::cout << "First element: " << vec[0] << std::endl;  }    int main() {  allocateMemory();  return 0;  } |

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

| **Principles(s):** Practice Defense in Depth: Protects against memory-related vulnerabilities such as buffer overflows, dangling pointers, and use-after-free bugs. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17+ | Cert-mem50-cpp | Detects unsafe access to freed memory and improper deallocations. |
| Valgrind | Latest | memcheck | Performs runtime detection of memory leaks, invalid accesses, and overflows. |
| [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-006- CPP] | Assertions are used to validate assumptions in code during development and debugging. However, using assertions for **runtime error handling** is dangerous because assertions are removed in release builds when compiled with NDEBUG. Proper error handling mechanisms, such as exceptions or error codes, should be used instead. |

| **Noncompliant Code** |
| --- |
| The following code incorrectly uses an assertion to validate user input, which may be disabled in release mode. |
| #include <iostream>  #include <cassert>    void processInput(int value) {  assert(value >= 0 && value <= 100); // Unsafe: Assertion removed in release mode  std::cout << "Processing value: " << value << std::endl;  }    int main() {  int input;  std::cout << "Enter a value between 0 and 100: ";  std::cin >> input;    processInput(input); // No actual error handling  return 0;  } |

| **Compliant Code** |
| --- |
| The compliant version uses explicit error handling to properly handle invalid input. |
| void processInput(int value) {  if (value < 0 || value > 100) {  throw std::out\_of\_range("Error: Value must be between 0 and 100.");  }  std::cout << "Processing value: " << value << std::endl;  }  int main() {  int input;  std::cout << "Enter a value between 0 and 100: ";  std::cin >> input;  try {  processInput(input); // Proper error handling  } catch (const std::exception& e) {  std::cerr << e.what() << std::endl;  }   return 0;  } |

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

| **Principles(s):**  Use Effective Quality Assurance Techniques: Highlights the importance of using proper runtime error handling rather than relying on development-only assertions.  Keep It Simple: Replaces overused or misused assert() logic with clear, maintainable error handling structures. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.10+ | assertWithSideEffects | Detects misuse of assertions, especially when they include critical logic |
| Clang-Tidy | 17+ | Misc-static-assert | Ensures proper usage of static\_assert and identifies bad runtime assertion use. |
| [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-007-CPP] | Exceptions are a critical part of modern C++ error handling, allowing for safe and structured error propagation. However, improper use, such as throwing raw pointers, failing to catch exceptions correctly, or catching them by value, can lead to resource leaks, undefined behavior, or poor program stability. |

| **Noncompliant Code** |
| --- |
| The following code demonstrates poor exception handling by catching exceptions by value and not rethrowing correctly. |
| void riskyOperation() {  throw std::runtime\_error("An error occurred!"); // Throws an exception  }    int main() {  try {  riskyOperation();  } catch (std::runtime\_error e) { // INCORRECT: Exception caught by value (slicing issue)  std::cerr << "Error: " << e.what() << std::endl;  }  return 0;  } |

| **Compliant Code** |
| --- |
| The compliant version correctly catches exceptions by reference, ensuring no slicing occurs. |
| void riskyOperation() {  throw std::runtime\_error("An error occurred!"); // Proper exception throwing  }    int main() {  try {  riskyOperation();  } catch (const std::runtime\_error& e) { // CORRECT: Catch by reference  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):**   * **Architect and Design for Security Policies**: Promotes structured exception handling as part of resilient application design. * **Use Effective Quality Assurance Techniques**: Ensures all thrown exceptions are caught properly and that exception safety is maintained. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17+ | Cert-err51-cpp | Detects unhandled exceptions and improper catch types |
| Cppcheck | 2.10+ | catchByValue | Flags incorrect exception handling practices, such as not catching base exception classes |
| [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** |
| --- | --- | --- |
| Input Validation | [STD-008-CPP] | Improper input validation can lead to buffer overflows, SQL injection, command injection, and undefined behavior. Ensuring that all user inputs are properly validated prevents security risks and ensures application stability. |

| **Noncompliant Code** |
| --- |
| The following code does not validate user input, which could cause buffer overflow or allow malicious inputs. |
| void processInput(char\* input) {  char buffer[10];  strcpy(buffer, input); // No bounds checking (buffer overflow risk)  std::cout << "You entered: " << buffer << std::endl;  }    int main() {  char userInput[50];  std::cout << "Enter text: ";  std::cin >> userInput; // No input validation  processInput(userInput);  return 0;  } |

| **Compliant Code** |
| --- |
| The compliant version ensures safe input handling by using std::string and validating input length. |
| void processInput(const std::string& input) {  if (input.length() > 10) {  std::cerr << "Error: Input too long!" << std::endl;  return;  }  std::cout << "You entered: " << input << std::endl;  }    int main() {  std::string userInput;  std::cout << "Enter text: ";  std::cin >> userInput; // Properly handling input with std::string  processInput(userInput);  return 0;  } |

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

| **Principles(s):**   * **Validate Input Data**: Prevents malicious or malformed input from entering the system and causing logic or security issues. * **Sanitize Data Sent to Other Systems**: Ensures that only well-formed and validated input reaches downstream systems like databases or files. * **Default Deny**: Rejects all input by default unless it meets strict criteria. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.10+ | strcpy | Detects unsafe input handling through unchecked buffer copies and raw input usage |
| [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** |
| --- | --- | --- |
| Secure File Handling | [STD-009-CPP] | Improper file handling can lead to race conditions, data corruption, privilege escalation, and unauthorized access. Following secure file access practices ensures data integrity and prevents security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The following code opens a file without checking for errors and does not handle privileges properly. |
| void readFile(const std::string& filename) {  std::ifstream file(filename); // No error handling  std::string line;    while (std::getline(file, line)) {  std::cout << line << std::endl;  }  }    int main() {  std::string filename;  std::cout << "Enter filename: ";  std::cin >> filename;    readFile(filename); // Unsafe file access  return 0;  } |

| **Compliant Code** |
| --- |
| The compliant version verifies file access, limits input scope, and handles errors properly. |
| bool isValidFilename(const std::string& filename) {  // Prevent directory traversal attacks by allowing only alphanumeric filenames  for (char ch : filename) {  if (!std::isalnum(ch) && ch != '.' && ch != '\_') {  return false;  }  }  return true;  }    void readFile(const std::string& filename) {  if (!isValidFilename(filename)) {  std::cerr << "Error: Invalid filename!" << std::endl;  return;  }    std::ifstream file(filename);  if (!file) {  std::cerr << "Error: Cannot open file!" << std::endl;  return;  }    std::string line;  while (std::getline(file, line)) {  std::cout << line << std::endl;  }  }    int main() {  std::string filename;  std::cout << "Enter filename: ";  std::cin >> filename;    readFile(filename);  return 0;  } |

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

| **Principles(s):**   * Adhere to the Principle of Least Privilege: Ensures file operations are only allowed where necessary and with minimal access rights. * Sanitize Data Sent to Other Systems: Prevents malicious file paths or commands from being used to exploit system-level file access. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17+ | Cert-fio30-c | Warns against unsafe file operations and improper EOF handling |
| CppCheck | 2.10+ | fileHandling | Detects unsafe file access patterns and missing validation on user-supplied filenames |
| [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** |
| --- | --- | --- |
| Concurrency Safety | [STD-010-CPP] | Improper handling of concurrency can lead to race conditions, deadlocks, and data corruption. Using proper thread synchronization mechanisms ensures that concurrent programs execute correctly and safely. |

| **Noncompliant Code** |
| --- |
| The following code shares a global variable between multiple threads without synchronization, leading to data races. |
| int counter = 0; // Shared resource    void incrementCounter() {  for (int i = 0; i < 10000; ++i) {  counter++; // Data race: multiple threads modifying without synchronization  }  }    int main() {  std::thread t1(incrementCounter);  std::thread t2(incrementCounter);    t1.join();  t2.join();    std::cout << "Final counter value: " << counter << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| The compliant version uses a mutex (std::mutex) to ensure that only one thread modifies the shared resource at a time. |
| int counter = 0;  std::mutex counterMutex; // Mutex to synchronize access    void incrementCounter() {  for (int i = 0; i < 10000; ++i) {  std::lock\_guard<std::mutex> lock(counterMutex); // Ensures mutual exclusion  counter++;  }  }    int main() {  std::thread t1(incrementCounter);  std::thread t2(incrementCounter);    t1.join();  t2.join();    std::cout << "Final counter value: " << counter << std::endl;  return 0;  } |

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

| **Principles(s):** [   * **Practice Defense in Depth**: Encourages the use of synchronization mechanisms and data consistency checks to protect shared resources. * **Use Effective Quality Assurance Techniques**: Promotes use of tools to detect data races, deadlocks, and concurrency misuse early in development. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10+ | seadlockRisk | Detects common concurrency issues like unsynchronized access to shared variables. |
| [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.

Automation will be integrated into each stage of Green Pace’s DevSecOps pipeline to enforce compliance with secure coding standards. Based on the continuous loop shown in the DevSecOps diagram, security tools like Cppcheck, Clang-Tidy, and SonarQube will be embedded from planning through deployment. Static analysis will be used during development and build stages to catch vulnerabilities early, while CI/CD pipelines will run automated security checks before code is released. Post-deployment, monitoring tools will track runtime behavior and generate alerts if deviations occur. This continuous integration of automated checks ensures early detection, consistent enforcement, reduced human error, and improved system integrity across the development lifecycle.

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | Medium | Likely | Low | Medium | 3 |
| STD-003-CPP | High | Likely | Medium | High | 4 |
| STD-004-CPP | High | Medium | Medium | High | 4 |
| STD-005-CPP | High | Likely | High | Critical | 5 |
| STD-006-CPP | Medium | Unlikely | Low | Low | 2 |
| STD-007-CPP | Medium | Medium | Medium | Medium | 3 |
| STD-008-CPP | High | Likely | Medium | High | 5 |
| STD-009-CPP | High | Medium | Medium | High | 4 |
| STD-010-CPP | High | Medium | High | High | 4 |

### 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 | Encryption at rest protects stored data on physical media, such as hard drives and databases. Green Pace systems will use AES-256 encryption for files and full-disk encryption on all devices where sensitive data is stored. This ensures that if physical devices are lost or stolen, the data remains unreadable to unauthorized parties. |
| Encryption in flight | Encryption in flight secures data during transmission over networks. Green Pace mandates the use of TLS 1.3 for all client-server communications, APIs, and remote access systems. This prevents man-in-the-middle attacks and protects credentials and confidential data in transit. |
| Encryption in use | Encryption in use refers to protecting data while it is being actively processed in memory. Although more advanced, this includes using secure enclaves (e.g., Intel SGX) and isolating memory processes when dealing with extremely sensitive operations. Green Pace applies this in financial or user-authenticated computations to minimize exposure during runtime. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies user identities before granting system access. Green Pace uses multi-factor authentication (MFA) for all employee logins and system access points. This policy applies at login, when accessing sensitive modules, and during role changes. It prevents unauthorized system access and strengthens identity verification. |
| Authorization | Authorization governs what authenticated users can access. Green Pace follows a role-based access control (RBAC) model to ensure users can only access data or functions required for their role. This policy applies when assigning permissions, onboarding new users, and when accessing critical files, APIs, or administrative features. |
| Accounting | Accounting refers to tracking and logging user activity to ensure traceability and support audits. Green Pace logs all user logins, permission changes, data access events, and system modifications. These logs are stored securely and reviewed regularly for compliance, anomaly detection, and breach 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

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Applicable Principles (by number)** | **Justification** |
| **STD-001-CPP** (Data Type) | 1, 9, 10 | (1) Validating input types avoids unintended behavior; (9) Quality tools can detect type mismatches; (10) Uses SEI CERT standards to enforce safe declarations. |
| **STD-002-CPP** (Data Value) | 1, 9, 10 | (1) Ensures input values are initialized and valid; (9) QA tools catch uninitialized or risky values; (10) Aligns with secure coding standards for variable safety. |
| **STD-003-CPP** (String Correctness) | 1, 7, 9, 10 | (1) Validates safe string input; (7) Sanitizing outbound data prevents overflows and injection; (9) Quality checks confirm string bounds; (10) Based on CERT string handling rules. |
| **STD-004-CPP** (SQL Injection) | 1, 3, 7, 10 | (1) Input validation stops dangerous injection strings; (3) Security must be planned in architecture; (7) Sanitizing user input ensures database safety; (10) Aligns with injection mitigation practices. |
| **STD-005-CPP** (Memory Protection) | 3, 8, 9, 10 | (3) Secure design avoids memory misuse; (8) Defense in depth applies checks across components; (9) QA tools detect leaks and overflows; (10) Follows CERT memory management rules. |
| **STD-006-CPP** (Assertions) | 4, 9, 10 | (4) Keeping logic simple avoids reliance on assertions for control; (9) QA can flag assertion misuse; (10) Supports best practice separation between dev and prod logic. |
| **STD-007-CPP** (Exceptions) | 3, 9, 10 | (3) Exception handling is part of secure design; (9) QA tools ensure safe exception paths; (10) Secure standard calls for proper exception propagation. |
| **STD-008-CPP** (Input Validation) | 1, 5, 7, 10 | (1) Core principle to reject invalid inputs; (5) Default-deny protects against unexpected behavior; (7) Sanitization avoids harm to other systems; (10) Enforced by policy and standards. |
| **STD-009-CPP** (Secure File Handling) | 5, 6, 7, 10 | (5) Limits file access by default; (6) Least privilege restricts file operations; (7) Sanitizing file names prevents attacks; (10) Rooted in CERT secure file operations. |
| **STD-010-CPP** (Concurrency Safety) | 3, 8, 9, 10 | (3) Concurrency requires deliberate architecture; (8) Defense in depth prevents race conditions and deadlocks; (9) Tools detect unsafe patterns; (10) Uses CERT rules for thread safety. |

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 | 03/22/2025 | Part 1 Done | Jacob Suich |  |
| 1.2 | 04/12/2025 | Finial Doc | Jacob Suich |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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