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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

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

[Project One 15](#_Toc52464070)

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

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

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

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

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

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

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

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validating input data is essential for preventing malicious data from compromising an application. This principle involves checking that all user-supplied input meets expected formats and constraints before it’s processed. By rigorously validating data, developers can mitigate risks like SQL injection, buffer overflows, and cross-site scripting, ensuring that only safe, expected data is accepted. |
| 1. Heed Compiler Warnings | Compiler warnings are early indicators of potential issues in the code that could lead to security vulnerabilities. Ignoring these warnings can allow subtle bugs or unsafe practices to persist, which attackers might exploit later. By treating warnings as errors and addressing them promptly, developers help maintain a robust and secure codebase. |
| 1. Architect and Design for Security Policies | Building security into the architecture from the start is critical. This principle means designing systems with clear security policies and incorporating measures such as encryption, authentication, and access control from the initial planning phase. A well-architected system aligns with industry standards and regulations, reducing the risk of vulnerabilities introduced later in the development process. |
| 1. Keep It Simple | Simplicity in design and code reduces the likelihood of introducing errors and vulnerabilities. Complex systems are harder to test, maintain, and secure. By keeping designs simple, developers can create more understandable and auditable code, which makes it easier to identify and resolve security issues. |
| 1. Default Deny | The default deny principle means that all access should be restricted unless explicitly allowed. This conservative approach minimizes the attack surface by ensuring that any unrecognized or unauthorized requests are automatically blocked. By defaulting to denial, systems enforce strict access control, which helps prevent unauthorized data access or system modifications. |
| 1. Adhere to the Principle of Least Privilege |  This principle mandates that every user, process, or system component should operate with the minimum level of access necessary to perform its tasks. By limiting privileges, the potential impact of a breach is contained, as attackers will have access only to a limited set of resources, reducing the overall risk.   |
| 1. Sanitize Data Sent to Other Systems | Sanitization involves cleaning and validating data before it is sent to external systems. This practice is crucial in preventing injection attacks and ensuring interoperability between systems. By thoroughly sanitizing outgoing data, developers ensure that only properly formatted and safe information is shared, protecting both the source and destination systems. |
| 1. Practice Defense in Depth | Defense in depth is a strategy that layers multiple security measures throughout an application. Instead of relying on a single security control, several overlapping defenses (e.g., firewalls, intrusion detection systems, encryption, and access controls) are implemented. This approach ensures that if one layer is breached, others remain in place to protect the system, providing comprehensive security. |
| 1. Use Effective Quality Assurance Techniques | Effective quality assurance involves thorough testing, including unit tests, integration tests, static analysis, and penetration testing. These techniques help uncover vulnerabilities and coding errors before deployment. Regular code reviews and automated testing frameworks ensure that the software remains secure and functions as expected under a variety of conditions. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard establishes a consistent framework for developing software safely. These standards, often derived from industry best practices and regulatory guidelines, provide clear rules and procedures that all developers must follow. A unified standard reduces the risk of security flaws, ensures compliance, and promotes the overall quality of the 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** | **Do not cast to an out-of-range enumeration value** |
| --- | --- | --- |
| **Data Type** | INT-50-CPP | Casting an integer to an enumeration type without verifying that the integer corresponds to a defined enumerator can result in undefined behavior and potentially expose security vulnerabilities. Ensuring that every cast is within the valid range of the enumeration promotes predictable behavior and robust error handling. This standard is critical for maintaining type safety and preventing logic errors that could be exploited or cause runtime faults. |

| **Noncompliant Code** |
| --- |
| The following example casts an integer to an enumeration type directly without checking whether the value is within the defined range. This can result in an out-of-range enumeration value, leading to unpredictable behavior. |
| enum Color { RED = 1, GREEN = 2, BLUE = 3 };  void setColor(int value) {  // Noncompliant: Casting an integer to enum without validating the value.  Color color = static\_cast<Color>(value);  // Use 'color' directly without checking if it is valid.  } |

| **Compliant Code** |
| --- |
| This example checks that the integer value falls within the defined range of the enumeration before performing the cast. If the value is out of range, it throws an exception to prevent undefined behavior. |
| #include <stdexcept>  enum Color { RED = 1, GREEN = 2, BLUE = 3 };  void setColor(int value) {  if (value < RED || value > BLUE) {  throw std::out\_of\_range("Value is out of range for Color enum");  }  Color color = static\_cast<Color>(value);  // 'color' is now guaranteed to be valid.  } |

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

| * **Validate Input Data – Ensuring that every external or internal integer is checked against the valid enumeration range before use directly maps to the principle of validating all input data to prevent misuse or unexpected values.** * **Keep It Simple – By performing an explicit, straightforward range check prior to casting, the code remains easy to understand and audit, reducing hidden complexity that can introduce security holes.** |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.4 | enumConstOverflow | Flags static\_cast<Enum>(int) without preceding range check |
| Clang-Tidy | 12.0 | Misc-enum-range-check | Warns when enum conversion isn’t guarded bu a range check |
| SonarQube | 8.9 | S3502 | Detects unchecked casts to enum types |
| Visual Studio Analysis | 2019 | Code Analysis: C6011, C6020 | Can be configured to spot potential invalid enum casts |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Ensure that integer conversions do not result in lost or misinterpreted data** |
| --- | --- | --- |
| **Data Value** | INT-31-C | Integer conversion operations can lead to data loss, truncation, or misinterpretation when values exceed the range of the target type. This standard ensures that every conversion is checked for safety, preserving data integrity and preventing subtle bugs that could lead to security vulnerabilities. By enforcing safe conversions, developers can prevent unexpected behavior, especially when converting between different integer sizes or between signed and unsigned types. |

| **Noncompliant Code** |
| --- |
| The following example converts a 64-bit integer to a 32-bit integer without checking whether the value fits within the 32-bit range. This can result in data truncation if the input value is too large. |
| #include <cstdint>  int32\_t convertValue(int64\_t value) {  // No range check: data loss may occur if 'value' exceeds the 32-bit limit.  return static\_cast<int32\_t>(value);  } |

| **Compliant Code** |
| --- |
| This example verifies that the 64-bit integer is within the valid range of a 32-bit integer before performing the conversion. If the value is out of range, it throws an exception to prevent data loss. |
| #include <cstdint>  #include <stdexcept>  #include <limits>  int32\_t convertValue(int64\_t value) {  if (value < std::numeric\_limits<int32\_t>::min() || value > std::numeric\_limits<int32\_t>::max()) {  throw std::out\_of\_range("Value cannot be represented as a 32-bit integer");  }  return static\_cast<int32\_t>(value);  } |

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

| **\* Validate Input Data – Every integer conversion is treated as input to a potentially dangerous operation and must be validated to ensure it falls within the allowable range.**  **\* Default Deny – Conversions that cannot be proven safe are refused (by throwing), rather than allowed to proceed and risk data corruption.** |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.4 | implicitIntegerConversion | Detects casts that may lose integer precision or range |
| Clang-Tidy | 12.0 | Bugprone-narrowing-conversions | Warns on implicit narrowing without explicit range validation |
| SonarQube | 8.9 | Cpp:S2131 | Flags potential data loss in integer narrowing conversions |
| Visual Studio Code Analysis | 2019 | Warning C4244 | Alerts on conversion from larger to smaller integer types |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Guarantee Sufficient Storage for Strings** |
| --- | --- | --- |
| **String Correctness** | STR-50-CPP | When using C-style strings, it is critical to allocate enough space to hold all characters plus the terminating null character ('\0'). Failing to reserve this extra space can result in buffer overflows, data corruption, and security vulnerabilities. This standard ensures that string buffers are sized appropriately to maintain program integrity and prevent unexpected behavior. |

| **Noncompliant Code** |
| --- |
| The following example allocates a character buffer that does not account for the null terminator. If the input string exactly fills the intended character count, there will be no space for the null terminator, leading to undefined behavior when the string is processed. |
| #include <cstring>  void copyString(const char\* input) {  // Allocates 5 bytes but does not reserve space for the null terminator.  char buffer[5];  // If input is "hello" (5 characters), buffer overflow will occur.  strcpy(buffer, input);  } |

| **Compliant Code** |
| --- |
| This example properly allocates a buffer with an extra byte for the null terminator, ensuring that the string is safely copied without overflowing the buffer. |
| #include <cstring>  void copyString(const char\* input) {  // Allocates 6 bytes: 5 for the characters and 1 for the null terminator.  char buffer[6];  strcpy(buffer, input);  } |

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

| **\* Validate Input Data – Even when buffer sizes are correct, inputs must be checked (e.g., via strncpy or length checks) to ensure they fit within the allocated space.**  **\* Default Deny – If input exceeds the maximum allowed length, reject it rather than truncating or allowing overflow.** |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.4 | bufferOverflow | Detects potential writes beyond array bounds when using string copy functions. |
| Clang-Tidy | 12.0 | Clang-analyzer-security.insecureAPI.strcpy | Warns on use of strcpy without explicit bounds checking. |
| SonarQube | 8.9 | Cpp:S3499 | Flags unsafe use of buffer copy functions that can overflow fixed-size arrays. |
| Visual Studio CA | 2019 | Warning C4996 | Marks deprecated/unbounded string functions such as strcpy as unsafe. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevent SQL Injection** |
| --- | --- | --- |
| **SQL Injection** |  | SQL injection is a critical vulnerability that occurs when untrusted user input is concatenated directly into SQL queries. This practice can allow attackers to modify query logic, access sensitive data, or even alter the database. To mitigate these risks, it is essential to use parameterized queries or prepared statements to ensure that user input is treated solely as data rather than executable SQL code. |

| **Noncompliant Code** |
| --- |
| The following example is noncompliant because it directly concatenates user input into a SQL query string, making the application vulnerable to SQL injection attacks. |
| #include <iostream>  #include <sstream>  #include <sqlite3.h>  void insecureQuery(sqlite3\* db, const std::string& username) {  std::stringstream sql;  sql << "SELECT \* FROM Users WHERE username = '" << username << "';";  char\* errorMessage = nullptr;  if (sqlite3\_exec(db, sql.str().c\_str(), nullptr, nullptr, &errorMessage) != SQLITE\_OK) {  std::cerr << "SQL error: " << errorMessage << std::endl;  sqlite3\_free(errorMessage);  }  } |

| **Compliant Code** |
| --- |
| This compliant example uses a prepared statement with parameterized queries. By binding user input as a parameter, the database treats the input strictly as data, which prevents SQL injection vulnerabilities. |
| #include <iostream>  #include <sqlite3.h>  void secureQuery(sqlite3\* db, const std::string& username) {  const char\* sql = "SELECT \* FROM Users WHERE username = ?;";  sqlite3\_stmt\* stmt;  if (sqlite3\_prepare\_v2(db, sql, -1, &stmt, nullptr) == SQLITE\_OK) {  // Bind the username safely to the prepared statement.  sqlite3\_bind\_text(stmt, 1, username.c\_str(), -1, SQLITE\_TRANSIENT);  while (sqlite3\_step(stmt) == SQLITE\_ROW) {  // Process each row as needed.  }  } else {  std::cerr << "Failed to prepare statement." << std::endl;  }  sqlite3\_finalize(stmt);  } |

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

| **\* Validate Input Data – Enforce that all external inputs be treated only as data; reject or escape anything unexpected.**  **\* Default Deny – Disallow any SQL execution paths that depend on unchecked string concatenation.** |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.4 | Sql-injection | Detects string concatenation into SQL APIs and flags unparameterized queries. |
| SonarQube | 8.0 | Cpp:S3649 | Enforces use of prepared statements and flags instances of dynamic SQL string construction. |
| Coverity | 2023.03 | SQL\_INJECTION | Identifies SQL call sites where arguments originate from external input without sanitization. |
| Fortify SCA | 23.1 | SQL-Injection | Scans C/C++ code for SQL query assembly patterns vulnerable to injection. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Do Not Access Free Memory** |
| --- | --- | --- |
| **Memory Protection** | MEM-50-CPP | Accessing memory after it has been freed can lead to undefined behavior, program crashes, or exploitable vulnerabilities (use-after-free attacks). Ensuring that freed memory is never accessed is critical for maintaining memory safety and application stability. This standard requires that once a memory block is deallocated, all pointers to that block are either set to nullptr or are no longer used, thereby preventing accidental or malicious use of freed memory. |

| **Noncompliant Code** |
| --- |
| The following example demonstrates noncompliant behavior by accessing a pointer after the memory has been freed. This can lead to unpredictable behavior and potential security vulnerabilities. |
| #include <cstdlib>  #include <iostream>  void useAfterFree() {  int\* data = new int[10];  // ... code that uses 'data' ...  delete[] data;  // Noncompliant: Accessing freed memory.  std::cout << "First element: " << data[0] << std::endl;  } |

| **Compliant Code** |
| --- |
| This example avoids accessing freed memory by setting the pointer to nullptr immediately after deletion. This ensures that any subsequent attempts to dereference the pointer can be safely checked against nullptr, preventing accidental use of freed memory. |
| #include <cstdlib>  #include <iostream>  void safeMemoryAccess() {  int\* data = new int[10];  // ... code that uses 'data' ...  delete[] data;  data = nullptr; // Safely reset the pointer  if (data) {  std::cout << "First element: " << data[0] << std::endl;  } else {  std::cout << "Memory has been freed and pointer is now null." << std::endl;  }  } |

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

| **\* Practice Defense in Depth – Memory hygiene is one layer of defense against exploitation; nulling pointers prevents accidental re‑use.**  **\* Use Effective Quality Assurance Techniques – Static analysis and code reviews catch use‑after‑free before deployment.** |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Possible | Low | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.4 | Memleak/nullPointer | Flags pointers used after deletion and null pointer dereferences. |
| Clang Static Analyzer | 12.0+ | Use-after-free | Detects code paths where freed memory is dereferenced. |
| Coverity | 2023.03 | USE\_AFTER\_FREE | Identifies instances of deallocation followed by pointer dereference. |
| SonarQube | 8.0 | Cpp:S675 | Ensures pointers are reset to nullptr post-free and guards all dereferences. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Do Not Abruptly Terminate the Program** |
| --- | --- | --- |
| **Assertions** | ERR-50-CPP | Abruptly terminating a program—such as when using assertions that call abort() in production—can lead to resource leaks, loss of data, and an unpredictable user experience. While assertions are useful for catching errors during development, relying on them in production code can prevent proper error handling and recovery. Instead, errors should be handled gracefully, allowing for resource cleanup and clear error reporting. This approach increases the robustness and security of the software by ensuring that the system remains in a controlled state even when unexpected conditions arise. |

| **Noncompliant Code** |
| --- |
| The following code uses the standard assert() macro, which, if triggered, abruptly terminates the program. This abrupt termination is undesirable in production environments because it does not allow for proper cleanup or error reporting. |
| #include <cassert>  #include <iostream>  void processData(int\* data, size\_t size) {  // Noncompliant: Using assert() for error handling leads to abrupt termination.  assert(data != nullptr);  // Further processing...  } |

| **Compliant Code** |
| --- |
| This example replaces the abrupt termination mechanism with a runtime check that handles the error gracefully. Instead of calling assert(), it logs an error message and safely returns from the function, ensuring that resources can be cleaned up properly. |
| #include <iostream>  void processData(int\* data, size\_t size) {  if (data == nullptr) {  std::cerr << "Error: Data pointer is null. Aborting processing gracefully." << std::endl;  return; // Handle error gracefully without abrupt termination.  }  // Further processing...  } |

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

| **\* Practice Defense in Depth – Layered error handling ensures that a single failure does not collapse the entire application.**  **\* Use Effective Quality Assurance Techniques – Replacing assert() with structured checks allows static analysis to verify correct error handling and prevents production crashes.** |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.4 | assertDanger | Detects use of assert() in code paths and warns against abrupt termination. |
| Clang Static Analyzer | 12.0+ | Debug.Assert | Flags calls to the assert() macro in non-debug builds. |
| Coverity | 2023.03 | ASSERT\_USED | Identifies use-after-assert() patterns that could abord program execution. |
| SonarQube | 8.0 | Cpp:S1871 | Disallows use of assert() in production code, enforcing explicit error handling instead. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle All Exceptions** |
| --- | --- | --- |
| **Exceptions** | ERR-51-CPP | Proper exception handling is essential for creating robust and secure applications. When exceptions are not caught, they can propagate and cause unexpected termination or resource leaks, leading to unpredictable behavior and security vulnerabilities. This standard requires that all exceptions thrown within a try block are caught and managed appropriately, either by logging the error, performing necessary cleanup, or rethrowing in a controlled manner. Handling exceptions gracefully ensures that the application remains stable even in error conditions. |

| **Noncompliant Code** |
| --- |
| The following example fails to handle exceptions because it omits any catch block. As a result, if an exception is thrown, it will propagate unhandled, potentially terminating the program unexpectedly. |
| #include <stdexcept>  #include <iostream>  void processData() {  try {  // Code that might throw an exception.  throw std::runtime\_error("Error occurred");  }  // No catch block provided – exception is not handled.  } |

| **Compliant Code** |
| --- |
| This example demonstrates proper exception handling by catching exceptions derived from std::exception. It logs the error message, allowing for graceful recovery or cleanup without abruptly terminating the program. |
| #include <stdexcept>  #include <iostream>  void processData() {  try {  // Code that might throw an exception.  throw std::runtime\_error("Error occurred");  }  catch (const std::exception& ex) {  std::cerr << "Exception caught: " << ex.what() << std::endl;  // Handle exception: perform cleanup, logging, or rethrow as needed.  }  } |

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

| **\* Practice Defense in Depth – Layered exception handling ensures failures are contained without cascading into higher layers.**  **\* Use Effective Quality Assurance Techniques – Enforcing full try/catch coverage can be verified via static analysis to prevent unhandled exceptions.** |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.4 | missingExceptionHandler | Flags try statements without any catch clause. |
| Clang Static Analyzer | 12.0 | Core.C++MissingException | Detects throw inside try blocks with no corresponding catch. |
| Coverity | 2023.03 | ERR\_EXC\_UNCAUGHT | Reports potential uncaught exceptions that may propagate past function boundaries. |
| SonarQube | 8.0 | Cpp:S112 | Ensures try blocks include catch clauses and discourages catching generic … only |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Do Not Modify the Standard Namespaces** |
| --- | --- | --- |
| Declarations | DCL-58-CPP | Modifying the standard namespaces (e.g., std) can lead to undefined behavior, conflicts with future standard library updates, and unexpected program behavior. The C++ Standard reserves the std namespace for the standard library, and any alterations to it can break portability, reduce maintainability, and introduce subtle bugs. Adhering to this standard ensures that developers do not inadvertently interfere with the well-defined behavior of the standard library. |

| **Noncompliant Code** |
| --- |
| The following example demonstrates noncompliant behavior by adding a new function to the std namespace. This modification can conflict with future standard definitions and is considered undefined behavior. |
| #include <iostream>  namespace std {  // Noncompliant: Modifying the standard namespace by adding a custom function.  void customPrint() {  std::cout << "Hello from customPrint" << std::endl;  }  } |

| **Compliant Code** |
| --- |
| This example avoids modifying the std namespace by placing custom functionality in a separate, user-defined namespace. This ensures that the standard library remains unaltered and that custom code does not conflict with future library updates. |
| #include <iostream>  namespace myutils {  // Compliant: Custom function is defined in a separate namespace.  void customPrint() {  std::cout << "Hello from customPrint" << std::endl;  }  } |

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

| **\* Adopt a Secure Coding Standard – Following recognized rules (e.g., SEI CERT C++) prevents misuse of reserved namespaces.**  **\* Architect and Design for Security Policies – Establishing clear boundaries (no changes to std) enforces maintainable and secure designs.** |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.4 | Dc158-cpp | Detects definitions inside the std namespace |
| Clang-Tidy | 12.0 | Misc-std-namespace | Flags any namespace std { … } blocks that introduce user code. |
| SonarQube | 8.0 | cppLS2416 | Prevents custom definitions in the std namespace per CERT C++ DCL58. |
| Pvs-Studio | 7.0 | V501 | Warns on modifications to standard library namespaces. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do Not Depend on the Order of Evaluation for Side Effects** |
| --- | --- | --- |
| Expressions | EXP-50-CPP | In C++, the order in which subexpressions are evaluated is often unspecified. Relying on a particular order for expressions that produce side effects (such as modifying a variable) can lead to undefined behavior. This standard mandates that developers write code that does not depend on the evaluation order to ensure predictable and portable behavior across different compilers and optimization settings. |

| **Noncompliant Code** |
| --- |
| The following code depends on the unspecified order of evaluation. The expression modifies the variable i multiple times within the same statement, leading to undefined behavior. |
| #include <iostream>  int main() {  int i = 1;  // Noncompliant: The order of evaluation of i++ and ++i is unspecified.  i = i++ + ++i;  std::cout << "Result: " << i << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| This compliant example splits the operations into separate statements. By doing so, it avoids relying on the order of evaluation within a single expression, ensuring defined and predictable behavior. |
| #include <iostream>  int main() {  int i = 1;  int temp = i++; // i is incremented after assignment; temp holds the original value.  i = temp + (++i); // Now, i is incremented before use.  std::cout << "Result: " << i << std::endl;  return 0;  } |

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

| **\* Keep It Simple – By avoiding complex single‑statement modifications, code remains clear and free of undefined behaviors.**  **\* Adopt a Secure Coding Standard – Conforming to CERT C++ EXP‑50‑CPP ensures side effects are sequenced properly, eliminating this class of bugs.** |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Possible | Low | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.4 | Uninitvar/sequencepoint | Detects use and modification of the same variable in one expression. |
| Clang-Tidy | 12.0 | Clang-diagnostic-undefined-behavior | Emits warnings when expressions may invoke undefined behavior due to side-effect ordering. |
| SonarQube | 8.0 | Cpp:S4755 | Flags code relying on unspecified evaluation order for side effects. |
| PVS-Studio | 7.0 | V688 | Reports modifications and use of a variable without an intervening sequence point. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Close the Files When They Are No Longer Needed** |
| --- | --- | --- |
| Input/Output | FIO-51-CPP | Files that are opened in a program consume system resources, such as file descriptors and memory buffers. Failing to close files when they are no longer needed can lead to resource leaks, which may eventually exhaust available resources and degrade system performance or even cause application crashes. By ensuring that files are explicitly closed after use, developers maintain efficient resource utilization and minimize potential security risks associated with resource exhaustion. |

| **Noncompliant Code** |
| --- |
| The following code demonstrates noncompliant behavior by opening a file but failing to close it after operations are complete. This oversight can lead to resource leaks and, in long-running applications, exhaustion of file descriptors. |
| #include <cstdio>  void readFile(const char\* filename) {  FILE\* file = fopen(filename, "r");  if (file == nullptr) {  return;  }  // Process file without closing it  // File remains open, leading to a resource leak.  } |

| **Compliant Code** |
| --- |
| This compliant example correctly closes the file after processing. The call to fclose() ensures that system resources are released properly, preventing resource leaks. |
| #include <cstdio>  void readFile(const char\* filename) {  FILE\* file = fopen(filename, "r");  if (file == nullptr) {  return;  }  // Process file  // ...  // Properly close the file to release resources  fclose(file);  } |

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

| * **Keep It Simple – By following a straightforward open‑process‑close pattern, code remains clear and free of resource‑leak pitfalls.** * **Use Effective Quality Assurance Techniques – Enforcing file‑close checks via code reviews and static analysis prevents leaks.** * **Adopt a Secure Coding Standard – Conforming to CERT FIO‑51‑CPP embeds resource‑management discipline into the codebase.** |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.4 | resourceLeak | Detects file descriptors opened without corresponding fclose. |
| Clang-Tidy | 12.0 | Clang-analyzer-cplusplus.ResourceLeak | Warns when opened files are not closed on all paths. |
| SonarQube | 8.0 | cppLS2095 | Flags file handles that are not closed before function exit. |
| PVS-Studio | 7.0 | V513 | Reports opened files never closed, preventing resource leaks. |

### 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.

[Insert your written explanations here.]

### 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 | Unlikely | Low | Medium | 3 |
| STD-003-CPP | High | Likely | Low | High | 2 |
| STD-004-CPP | High | Likely | Low | High | 1 |
| STD-005-CPP | High | Possible | Low | High | 1 |
| STD-006-CPP | Medium | Possible | Low | Medium | 1 |
| STD-007-CPP | High | Likely | Low | High | 1 |
| STD-008-CPP | Medium | Unlikely | Low | Medium | 2 |
| STD-009-CPP | High | Possible | Low | High | 2 |
| STD-010-CPP | Medium | Likely | 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.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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 | All sensitive data stored on disk—whether in databases, file shares, backups, or removable media—must be encrypted using AES‑256 or stronger algorithms. Encryption keys must be managed in a centralized Key Management Service (KMS) with strict access controls. This policy ensures that if storage media are stolen or improperly decommissioned, the data remain unreadable to unauthorized parties. |
| Encryption in flight | All network communications carrying sensitive or regulated data (including internal service‑to‑service calls, API traffic, and external client connections) must use TLS 1.2 or higher with strong ciphers (e.g. ECDHE‑AES256‑GCM). Certificates must be issued by a trusted internal or public CA and rotated at least annually. This protects data from eavesdropping, man‑in‑the‑middle attacks, and tampering while in transit. |
| Encryption in use | Any sensitive data loaded into application memory—such as encryption keys, authentication tokens, or PII—must be protected using in‑process encryption or hardware‑backed trusted execution environments (e.g., Intel SGX, AMD SEV). Applications must zeroize buffers immediately after use. This prevents attackers who gain memory dumps or attach debuggers from extracting plaintext data during processing. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Every user and system identity must be authenticated before granting any access. This requires unique user IDs and strong multi‑factor authentication (MFA) for all interactive logins and privileged operations. Service‑to‑service calls must use mutual TLS or signed tokens. By verifying identity up front, we ensure only legitimate actors can initiate actions in our environment. |
| Authorization | Access to resources is granted based on the principle of least privilege. Role‑Based Access Control (RBAC) policies define which operations each role may perform (e.g., read, write, delete). Requests outside those rights must be denied. Access reviews occur quarterly to adjust privileges as roles change. This ensures users and services can act only within their approved scope. |
| Accounting | All security‑relevant events—including user logins/logouts, changes to the database schema or data, creation or deletion of user accounts, and file accesses—must be logged centrally in an immutable audit trail. Logs must capture timestamp, actor identity, action performed, and outcome. Retain logs for at least one year to support forensic investigations, compliance audits, and anomaly detection. |

**\***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
* **1. Do not cast to an out‑of‑range enumeration value**
* **Standard:** CERT INT‑50‑CPP  
  **Principles Applied:**
* **1. Validate Input Data**
  + Every integer-to-enum cast is preceded by an explicit range check.
* **4. Keep It Simple**
  + A single, clear if‑statement makes the check auditable and easy to understand.
* **10. Adopt a Secure Coding Standard**
  + Conforms directly to SEI CERT guideline INT‑50‑CPP.
* **2. Ensure safe integer conversions**
* **Standard:** CERT INT‑31‑C  
  **Principles Applied:**
* **1. Validate Input Data**
  + All conversions include bounds checking against type limits.
* **5. Default Deny**
  + Out‑of‑range values are refused (exception thrown) rather than silently truncated.
* **9. Use Effective Quality Assurance Techniques**
  + Range‑check logic can be verified by static analysis tools.
* **3. Guarantee sufficient storage for C‑strings**
* **Standard:** CERT STR‑50‑CPP  
  **Principles Applied:**
* **1. Validate Input Data**
  + Inputs must be length‑checked (e.g., via strncpy or strlen) before copy.
* **5. Default Deny**
  + If an input exceeds buffer capacity, reject it outright instead of truncating.
* **9. Use Effective Quality Assurance Techniques**
  + Static analyzers detect missing null‑space in buffer allocations.
* **4. Prevent SQL injection**
* **Standard:** (Custom SQL Injection)  
  **Principles Applied:**
* **1. Validate Input Data**
  + All user‑supplied strings are treated strictly as data.
* **5. Default Deny**
  + Disallow any query path that concatenates raw input.
* **7. Sanitize Data Sent to Other Systems**
  + Use parameterized queries/bound parameters.
* **8. Practice Defense in Depth**
  + Multiple layers (input validation + prepared statements) guard the database.
* **5. Do not access freed memory**
* **Standard:** CERT MEM‑50‑CPP  
  **Principles Applied:**
* **8. Practice Defense in Depth**
  + Nulling pointers after delete adds a secondary safety net.
* **9. Use Effective Quality Assurance Techniques**
  + Static tools reliably flag use‑after‑free patterns.
* **6. Do not abruptly terminate the program**
* **Standard:** CERT ERR‑50‑CPP  
  **Principles Applied:**
* **8. Practice Defense in Depth**
  + Replace assert() with structured error‑handling to avoid whole‑program crashes.
* **9. Use Effective Quality Assurance Techniques**
  + Static checks can enforce that production code uses graceful checks, not assertions.
* **7. Handle all exceptions**
* **Standard:** CERT ERR‑51‑CPP  
  **Principles Applied:**
* **8. Practice Defense in Depth**
  + Layered try/catch ensures no exception propagates uncaught.
* **9. Use Effective Quality Assurance Techniques**
  + Tools can verify that every throw has a matching catch.
* **8. Do not modify the standard namespaces**
* **Standard:** CERT DCL‑58‑CPP  
  **Principles Applied:**
* **3. Architect and Design for Security Policies**
  + Enforces clear separation between user and library code.
* **4. Keep It Simple**
  + Avoids confusing or conflicting extensions to std.
* **10. Adopt a Secure Coding Standard**
  + Directly follows SEI CERT guidance on namespace usage.
* **9. Do not depend on order of evaluation for side effects**
* **Standard:** CERT EXP‑50‑CPP  
  **Principles Applied:**
* **4. Keep It Simple**
  + Breaking complex expressions into simple statements ensures defined behavior.
* **9. Use Effective Quality Assurance Techniques**
  + Static analyzers flag unspecified side‑effect ordering.
* **10. Adopt a Secure Coding Standard**
  + Implements CERT’s requirement for well‑sequenced operations.
* **10. Close files when they are no longer needed**
* **Standard:** CERT FIO‑51‑CPP  
  **Principles Applied:**
* **4. Keep It Simple**
  + A clear “open—process—close” pattern prevents resource leaks.
* **8. Practice Defense in Depth**
  + Explicit fclose() calls add resilience against descriptor exhaustion.
* **9. Use Effective Quality Assurance Techniques**
  + Static checks detect missing file closures on all code paths.

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 |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [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 |