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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_heading=h.gjdgxs)

[Purpose 2](#_heading=h.30j0zll)

[Scope 2](#_heading=h.1fob9te)

[Module Three Milestone 2](#_heading=h.3znysh7)

[Ten Core Security Principles 2](#_heading=h.4d34og8)

[C/C++ Ten Coding Standards 3](#_heading=h.2s8eyo1)

[Coding Standard 1 4](#_heading=h.3rdcrjn)

[Coding Standard 2 5](#_heading=h.26in1rg)

[Coding Standard 3 6](#_heading=h.lnxbz9)

[Coding Standard 4 7](#_heading=h.35nkun2)

[Coding Standard 5 8](#_heading=h.1ksv4uv)

[Coding Standard 6 9](#_heading=h.2jxsxqh)

[Coding Standard 7 10](#_heading=h.z337ya)

[Coding Standard 8 11](#_heading=h.28h4qwu)

[Coding Standard 9 13](#_heading=h.nmf14n)

[Coding Standard 10 14](#_heading=h.37m2jsg)

[Defense-in-Depth Illustration 15](#_heading=h.1mrcu09)

[Project One 15](#_heading=h.46r0co2)

[1.](#_heading=h.2lwamvv) Revise the C/C++ Standards 15

[2.](#_heading=h.111kx3o) Risk Assessment 15

[3.](#_heading=h.3l18frh) Automated Detection 15

[4.](#_heading=h.206ipza) Automation 15

[5.](#_heading=h.4k668n3) Summary of Risk Assessments 16

[6.](#_heading=h.2zbgiuw) Create Policies for Encryption and Triple A 16

[7.](#_heading=h.1egqt2p) Map the Principles 17

[Audit Controls and Management 18](#_heading=h.3ygebqi)

[Enforcement 18](#_heading=h.2dlolyb)

[Exceptions Process 18](#_heading=h.sqyw64)

[Distribution 19](#_heading=h.3cqmetx)

[Policy Change Control 19](#_heading=h.1rvwp1q)

[Policy Version History 19](#_heading=h.4bvk7pj)

[Appendix A Lookups 19](#_heading=h.2r0uhxc)

[Approved C/C++ Language Acronyms 19](#_heading=h.1664s55)

## 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 | Ensure that all input data is verified before being processed by the system. This prevents malicious data from compromising system functionality by limiting inputs to expected values or formats. |
| 1. Heed Compiler Warnings | Compilers generate warnings about possible code vulnerabilities. Treat these warnings seriously, as they often flag potential issues in logic, buffer handling, and memory management, which can lead to security risks if ignored. |
| 1. Architect and Design for Security Policies | Security must be an integral part of the system's architecture. This principle emphasizes building systems with security in mind, incorporating features such as encryption, access control, and threat detection mechanisms at the design phase to avoid vulnerabilities later on. |
| 1. Keep It Simple | Complexity is the enemy of security. Overcomplicated systems are more prone to vulnerabilities, as they are harder to test, maintain, and defend. Simple, straightforward designs reduce potential attack surfaces and ease the identification of flaws. |
| 1. Default Deny | By default, systems should deny access to resources unless explicitly allowed. This principle limits the possibility of unauthorized access and prevents attackers from exploiting open access or default settings. |
| 1. Adhere to the Principle of Least Privilege | Each component or user in the system should operate with the minimum level of privileges necessary to perform its function. This limits the impact of an attack if an account or component is compromised, reducing the damage potential. |
| 1. Sanitize Data Sent to Other Systems | Before sending data to external systems, ensure it is sanitized. This involves removing harmful characters and formatting errors that could lead to vulnerabilities like SQL injection or command injection. |
| 1. Practice Defense in Depth | Layer security measures to provide multiple barriers to attackers. Instead of relying on a single point of defense, use a variety of security mechanisms (firewalls, encryption, intrusion detection, etc.) to provide redundancy in protecting the system. |
| 1. Use Effective Quality Assurance Techniques | Rigorous testing, code reviews, and static analysis tools are critical in identifying and addressing security flaws early. These practices ensure code quality and catch issues that could be exploited by attackers. |
| 1. Adopt a Secure Coding Standard | Follow established secure coding standards like SEI CERT C++ or other language-specific guidelines. This helps ensure consistent, secure programming practices across the entire development team, reducing the risk of introducing security vulnerabilities through coding errors. |

### 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** | **Correct Data Type Usage** |
| --- | --- | --- |
| **Data Type** | STD-001-DCL | Correct data type usage is essential for ensuring that variables and functions behave as expected, minimizing the risk of undefined behavior, type mismatches, or overflows. Using incorrect data types can lead to security vulnerabilities like buffer overflows or data corruption. |

| **Noncompliant Code** |
| --- |
| This example uses a C-style variadic function to add a series of integers together. Calling this function without passing a terminating 0 or passing a non-integer type results in undefined behavior. |
| #include <cstdarg>  int add(int first, int second, ...) {  int r = first + second;  va\_list va;  va\_start(va, second);  while (int v = va\_arg(va, int)) {  r += v;  }  va\_end(va);  return r;  } |

| **Compliant Code** |
| --- |
| This example uses a variadic template function with function parameter packs, ensuring that only integers are passed and preventing undefined behavior. |
| #include <type\_traits>  template <typename Arg, typename std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  int add(Arg f, Arg s) { return f + s; }  template <typename Arg, typename... Ts, typename std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  int add(Arg f, Ts... rest) {  return f + add(rest...);  } |

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

| **Principles(s):** The **Type Safety** principle ensures that variables and functions are defined using the correct data types, which helps prevent type mismatches and undefined behavior. This principle supports the correct data type usage standard by enforcing proper type usage at compile time, thus avoiding security vulnerabilities like buffer overflows or data corruption. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| **High** – Incorrect data type usage can lead to severe issues such as memory corruption, buffer overflows, and program crashes, potentially leading to exploitable vulnerabilities. | **Medium** – While these issues are somewhat less frequent in modern codebases due to static analysis tools, they remain prevalent enough to warrant attention in complex systems. | **Medium** – Correcting type issues may require refactoring functions and re-validating all affected code, but automated tools can help mitigate this cost. | **High** – Data type issues should be resolved early in the development lifecycle to avoid cascading problems throughout the system. | **2** – Represents a moderate risk due to the potential impact and importance of ensuring data type correctness in secure coding practices. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 (LTS) | C++ S124 – Avoid type mismatches and ensure correct casting between types. | SonarQube provides continuous static analysis and ensures compliance with coding standards. It checks for type safety violations and flags incorrect or risky data type usage, integrating easily into DevSecOps pipelines. |
| Clang-Tidy | 14.0 | cert-dcl21-c – Ensure that all integer data types used in variadic functions are properly defined. | Clang-Tidy is a powerful tool for identifying issues related to type mismatches, improper template usage, and enforcing correct data types during development. |
| Cppcheck | 2.7 | MISRA C++ Rule 5-0-2 – Enforces consistent and correct use of types, preventing type-related errors. | Cppcheck performs static code analysis focused on detecting coding errors, including those related to incorrect data type usage, in C/C++ codebases. It ensures compliance with secure coding standards and supports continuous integration. |
| PVS-Studio | 7.15 | V521 – Warns about potential data loss due to implicit type conversions. | PVS-Studio helps detect type conversion issues and type safety problems. Its static analysis reports identify potential vulnerabilities early, ensuring data type correctness during development. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Correct Data Value Usage** |
| --- | --- | --- |
| **Data Value** | STD-002-INT | Correct data value usage is essential for ensuring that operations involving integers or other data types do not result in unexpected behaviors such as overflow, underflow, or incorrect initializations. Integer overflow or underflow can cause security vulnerabilities and improper program behavior. |

| **Noncompliant Code** |
| --- |
| The function multiplies x by 1000 and adds y. If x is large enough, multiplying it by 1000 could cause integer overflow, leading to undefined behavior or incorrect results. |
| int multiply\_and\_add(int x, int y) {  int result = (x \* 1000) + y;  return result;  } |

| **Compliant Code** |
| --- |
| This version checks for integer overflow before performing the multiplication, preventing overflow errors. |
| #include <limits>  bool safe\_multiply(int x, int y, int& result) {  if (x > 0 && y > 0 && x > (std::numeric\_limits<int>::max() / y)) {  return false; // Overflow would occur  }  result = x \* y;  return true;  }  int multiply\_and\_add(int x, int y) {  int result;  if (!safe\_multiply(x, 1000, result)) {  // Handle overflow case  return -1;  }  return result + y;  } |

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

| **Principles(s):** The **Boundary Checking** principle ensures that operations on integers are handled within the valid bounds to avoid overflow and underflow conditions. This principle applies directly to this standard, as unchecked integer operations can result in undefined behavior or security vulnerabilities like memory corruption. Properly checking and handling integer boundaries mitigates the risk of such issues. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| **High** – Integer overflow can result in critical errors like memory corruption, leading to program crashes or exploitable vulnerabilities. | **Medium** – These issues remain frequent in arithmetic-heavy applications but can be mitigated with thorough static analysis. | **Medium** – Correcting overflow vulnerabilities may require code refactoring and runtime validation checks, which could add overhead but are necessary for secure coding. | **High** – These issues should be addressed early in the development lifecycle to prevent cascading failures or security breaches. | **2** – Integer overflow is a serious issue in secure code practices and should be addressed with proper remediation techniques. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 (LTS) | C++ S1247 – Detect integer overflows and arithmetic overflows | SonarQube ensures compliance with secure coding standards and helps detect integer overflows, flagging unsafe operations during arithmetic computations. |
| Clang-Tidy | 14.0 | cert-int31-c – Ensure integer overflow detection | Clang-Tidy performs static analysis to identify unsafe integer operations that can result in overflow. It checks for safe usage of signed and unsigned integers in boundary cases. |
| Cppcheck | 2.9 | MISRA C++ Rule 7-5-2 – Ensure proper use of arithmetic operations | Cppcheck detects coding errors related to improper use of integers in multiplication or addition, preventing overflow and enhancing overall code safety. |
| GCC (with -ftrapv flag) | 11.2 | Detect overflow at runtime | The -ftrapv flag in GCC ensures that arithmetic overflow is trapped at runtime, terminating the program when an overflow occurs, providing immediate feedback during testing. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Correct String Handling** |
| --- | --- | --- |
| **String Correctness** | STD-003-STR | Correct string handling is essential for preventing buffer overflows and ensuring data integrity. Strings should always be allocated with sufficient space for the data and the null terminator. Inadequate string handling can lead to vulnerabilities, such as memory corruption or security exploits. |

| **Noncompliant Code** |
| --- |
| This example uses strcpy without checking the size of the destination buffer. If the dest buffer is smaller than the source string, this can lead to a buffer overflow, causing memory corruption or program crashes. |
| #include <cstring>  void copy\_string(char \*dest, const char \*src) {  strcpy(dest, src); // Potential buffer overflow if dest is smaller than src  } |

| **Compliant Code** |
| --- |
| In this compliant example, strncpy is used to copy the string while ensuring the size of the destination buffer is respected. The code also explicitly null-terminates the string to avoid any truncation or non-null terminated strings, following the best practices for secure string handling. |
| #include <cstring>  void copy\_string(char \*dest, const char \*src, size\_t size) {  strncpy(dest, src, size - 1); // Safe copy with size limit  dest[size - 1] = '\0'; // Ensure null termination  } |

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

| **Principles(s):** The **Input Validation and Representation** principle applies to this standard. It ensures that string operations are performed safely by validating input sizes and representations. By checking the length of strings and ensuring proper null termination, this principle prevents buffer overflows and data corruption due to improper string handling. This principle is integral to secure coding practices that prevent buffer overflows from happening. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| **High** – Incorrect string handling can lead to buffer overflows, memory corruption, or program crashes. These vulnerabilities are often exploited in security attacks, such as buffer overflow attacks, which can lead to system breaches. | **Medium** – String handling issues remain common, especially in legacy systems or codebases using unsafe C-style string functions like strcpy. Modern codebases tend to avoid these, but the threat still persists. | **Medium** – Correcting string handling issues involves reviewing and refactoring the code to replace unsafe functions with safer alternatives (e.g., strncpy). Automated tools can help mitigate the remediation cost. | **High** – Ensuring correct string handling is critical to avoid buffer overflows, which can lead to serious security vulnerabilities and system crashes. | **2** – This vulnerability has moderate to high impact and requires immediate attention to prevent significant security risks. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 (LTS) | C++ S5878 – Detect unsafe string manipulation (e.g., strcpy, sprintf, and similar functions) | SonarQube flags unsafe string manipulations and checks for buffer overflow risks due to improper string handling. It helps detect vulnerabilities in string operations that can lead to memory corruption or crashes. |
| Clang-Tidy | 14.0 | cert-str34-c – Ensure proper handling of string literals | Clang-Tidy performs static analysis to identify unsafe string operations that can cause buffer overflows, particularly in C/C++ codebases. It helps enforce the correct handling of string operations. |
| Coverity | 2023.3 | CWE-120 – Buffer Copy without Checking Size of Input | Coverity helps identify unsafe buffer handling in string operations, particularly around strcpy and memcpy. It ensures proper bounds checking to prevent buffer overflows. |
| Flawfinder | 2.0.19 | STR01-A – Detect buffer overflows caused by improper string handling | Flawfinder performs static analysis, focusing on security risks such as buffer overflows and improper string handling. It detects vulnerabilities in unsafe C functions and suggests replacements to ensure secure string manipulation. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevent SQL Injection Attacks** |
| --- | --- | --- |
| **SQL Injection** | STD-004-SQL | Preventing SQL injection is critical for ensuring that SQL queries are executed safely and do not allow malicious input to modify query execution. SQL injection attacks exploit vulnerabilities by inserting or altering SQL statements through user inputs, leading to unauthorized access, data leaks, or data manipulation. Avoiding direct string concatenation and using prepared statements or parameterized queries mitigates these risks. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, the user input is concatenated directly into the SQL query string. If user\_input contains malicious SQL (e.g., username'; DROP TABLE users;--), the query will execute destructive actions on the database. |
| #include <string>  #include <mysql/mysql.h>  void execute\_query(const std::string& user\_input) {  MYSQL\* connection;  std::string query = "SELECT \* FROM users WHERE username = '" + user\_input + "';"; // Vulnerable query  mysql\_query(connection, query.c\_str());  } |

| **Compliant Code** |
| --- |
| In this compliant solution, a prepared statement (mysql\_stmt\_prepare) is used along with a bound parameter (mysql\_stmt\_bind\_param), ensuring that user input is treated as data and not as executable SQL code. This effectively prevents SQL injection. |
| #include <mysql/mysql.h>  void execute\_query\_safe(const std::string& user\_input) {  MYSQL\* connection;  MYSQL\_STMT\* stmt = mysql\_stmt\_init(connection);  const char\* query = "SELECT \* FROM users WHERE username = ?";    mysql\_stmt\_prepare(stmt, query, strlen(query));  mysql\_stmt\_bind\_param(stmt, MYSQL\_TYPE\_STRING, user\_input.c\_str(), user\_input.size());  mysql\_stmt\_execute(stmt);    mysql\_stmt\_close(stmt);  } |

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

| **Principles(s):** The **Input Validation and Sanitization** principle is critical for this standard. It ensures that user inputs are properly validated and sanitized before being used in SQL queries. This principle prevents user inputs from being treated as executable code, thus safeguarding against SQL injection attacks. The principle also emphasizes the use of parameterized queries, which directly address this vulnerability. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| **High** – SQL injection can lead to full database compromise, unauthorized access, data leakage, and potential system failures. | **Medium** – SQL injection remains a common attack vector, especially in web applications that improperly handle user inputs. | **Medium** – Refactoring may be required to implement parameterized queries, but automated detection tools can assist in identifying vulnerable code. | **High** – Addressing SQL injection vulnerabilities should be a top priority due to their potential severity. | **1** – Represents a high-level risk and requires immediate remediation to prevent critical security breaches. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 (LTS) | C++ S2077 – Detect SQL injection vulnerabilities | SonarQube flags unsafe SQL queries and detects SQL injection risks by identifying the use of dynamic SQL and untrusted inputs in queries. |
| Clang-Tidy | 14.0 | cert-sql-inj1-c – Ensure safe SQL query construction | Clang-Tidy helps detect unsafe SQL query constructions and flags improper string concatenations used in queries, enforcing parameterized queries as a best practice. |
| CodeQL | 2.8.4 | SQL Injection Query | CodeQL analyzes code to identify SQL injection vulnerabilities, particularly in cases where user input is used unsafely in SQL queries. It flags dangerous patterns and suggests the use of prepared statements. |
| OWASP ZAP | 2.10.0 | SQL Injection Detection Rule | OWASP ZAP performs dynamic analysis by scanning web applications for SQL injection vulnerabilities. It simulates attack scenarios to detect potential vulnerabilities and suggests mitigation techniques. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Proper Memory Management and Protection** |
| --- | --- | --- |
| **Memory Protection** | STD-005-MEM | Memory protection is critical for preventing issues such as memory leaks, buffer overflows, and other vulnerabilities associated with improper memory usage. Proper memory management ensures that memory is allocated, accessed, and freed correctly, reducing the risk of unexpected program crashes, data corruption, or security breaches due to buffer overflows or memory access violations. |

| **Noncompliant Code** |
| --- |
| This noncompliant example shows a memory leak where the allocated memory is not freed, leading to potential memory exhaustion if this function is called repeatedly. |
| #include <cstdlib>  void allocate\_memory() {  int\* data = (int\*) malloc(100 \* sizeof(int));  // Memory is allocated, but not freed  data[0] = 10;  } |

| **Compliant Code** |
| --- |
| This compliant example ensures that the allocated memory is freed after use, preventing memory leaks. |
| #include <cstdlib>  void allocate\_memory\_safe() {  int\* data = (int\*) malloc(100 \* sizeof(int));  if (data != nullptr) {  data[0] = 10;  free(data); // Memory is freed after use  }  } |

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

| **Principles(s):** The **Resource Management** Principle: This principle ensures that system resources such as memory are properly allocated and deallocated. Following this principle helps avoid memory leaks, where memory that is no longer needed is not freed, leading to performance degradation or even system crashes. By ensuring that allocated memory is freed after use, as shown in the compliant code, we prevent such vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| **High** – Memory leaks can lead to performance issues, resource exhaustion, or even application crashes, which can be exploited by malicious actors to cause denial of service. | **Medium** – Memory leaks are still prevalent in applications, especially when dynamic memory allocation is used heavily. However, with modern practices such as smart pointers in C++, the risk can be reduced. | **Medium** – Fixing memory leaks often involves a thorough review of the codebase and adding checks for proper deallocation. However, automated tools can assist with this task, reducing manual effort. | **High** – Addressing memory leaks should be a top priority due to the performance and stability risks they pose. | **1 –** This represents a high-priority issue that should be resolved immediately to prevent severe consequences. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.18.1 | Memory Leak Detector | Valgrind is a powerful tool for detecting memory leaks by running code through a set of tools to monitor memory usage and detect improper memory management. |
| Cppcheck | 2.9 | MISRA C++ Rule 18-0-4 – Proper Resource Management | Cppcheck performs static code analysis and checks for improper resource management, including memory leaks. |
| Clang-Tidy | 14.0 | cert-msc30-c – Detect dynamic memory issues | Clang-Tidy can analyze dynamic memory operations, ensuring proper memory allocation and deallocation. |
| GCC (with -fsanitize=address) | 11.2 | Detect memory leaks during runtime | The -fsanitize=address flag in GCC helps detect memory leaks and other memory access violations during runtime. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Proper Use of Assertions** |
| --- | --- | --- |
| **Assertions** | STD-006-ASR | Assertions should be used to detect and debug unexpected conditions during development. However, assertions must not be relied upon for checking user input or ensuring program correctness in production code. Assertions should not have side effects or change the state of the program. Overusing or misusing assertions can lead to program crashes or undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, an assertion is used to check whether the input data is null. This is inappropriate for production code because assertions may be disabled in some builds, leaving the program without null pointer checks and potentially leading to crashes or undefined behavior. |
| #include <assert.h>  void process\_data(int \*data) {  assert(data != NULL); // Should not be used for checking input in production code  // Process the data  \*data = 42;  } |

| **Compliant Code** |
| --- |
| In the compliant code, null pointers are checked using a proper if statement, and assertions are used only to validate conditions that should never fail. This ensures that the code behaves correctly both in development and production environments. |
| #include <assert.h>  void process\_data(int \*data) {  if (data == NULL) {  // Handle null pointer error  return;  }  assert(\*data != 0); // Assert a condition that should never be false  // Process the data  \*data = 42;  } |

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

| **Principles(s):** The **Validation and Error Handling** principle ensures that assertions are used in a secure and appropriate manner. This principle is applied directly to this standard, as assertions are crucial for detecting and debugging unexpected conditions during development. By ensuring that assertions are not misused in production code (for example, using them for input validation), we avoid potential program crashes and undefined behavior in secure systems. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| **Medium** – Misusing assertions in production code can result in program crashes or undefined behavior, especially when assertions are disabled in some builds. | **Low** – Assertions are typically used during development rather than in production, making this less likely, but still a risk. | **Low** – Correcting improper use of assertions in development code is relatively simple, involving refactoring assertion logic to proper error handling. | **Medium** – Addressing this risk ensures stable code in both development and production environments. | **2** – Represents a moderate risk, as assertion misuse could lead to failures in production if not handled properly. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 (LTS) | C++ S5773 – Detect misuse of assertions | SonarQube checks for proper use of assertions in the code, flagging improper usage that could cause issues if assertions are disabled in production builds. |
| Cppcheck | 2.9 | MISRA C++ Rule 16.0-1 | Cppcheck helps ensure that assertions are only used for conditions that should never fail in development environments, not for input validation. |
| Clang-Tidy | 14.0 | cert-err33-c – Detect improper use of assertions | Clang-Tidy checks for improper use of assertions and recommends better practices, such as replacing assertions with proper error handling in production code. |
| PVS-Studio | 7.15 | V1065 – Check for improper use of assertions | PVS-Studio identifies potential misuse of assertions in production environments and flags cases where assertions should be replaced with explicit error handling logic. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Proper Exception Handling** |
| --- | --- | --- |
| **Exceptions** | STD-007-EXC | Proper exception handling is critical to ensuring that programs can recover gracefully from errors and avoid undefined behavior or crashes. Using exceptions correctly also ensures that resources are properly released, and error conditions are propagated safely. Inappropriate use of exceptions, such as handling exceptions that should not be caught or using exceptions for non-exceptional conditions, can introduce security vulnerabilities or inconsistent program behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the exception handling is misused by catching exceptions of type std::exception without specific actions, leading to improper error handling. |
| #include <iostream>  #include <stdexcept>  void process\_data(int data) {  try {  if (data < 0) {  throw std::runtime\_error("Negative data not allowed");  }  // Process the data  }  catch (std::exception &e) {  // No specific handling, exception is just caught  std::cerr << "Exception caught: " << e.what() << std::endl;  }  } |

| **Compliant Code** |
| --- |
| In the compliant solution, specific actions are taken after catching exceptions, and exception handling is performed appropriately, including propagating critical errors. |
| #include <iostream>  #include <stdexcept>  void process\_data(int data) {  try {  if (data < 0) {  throw std::runtime\_error("Negative data not allowed");  }  // Process the data  }  catch (const std::runtime\_error &e) {  // Handle the specific exception  std::cerr << "Runtime error: " << e.what() << std::endl;  throw; // Propagate the exception after logging  }  catch (const std::exception &e) {  // Handle general exceptions  std::cerr << "General error: " << e.what() << std::endl;  }  } |

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

| **Principles(s):** The **Error Handling** principle ensures that exceptions are used appropriately in a way that allows the program to recover gracefully from errors. It mandates the correct handling and propagation of exceptions, ensuring that the program doesn't catch exceptions it shouldn't, and avoids inconsistent behavior or security risks. This principle supports safe exception handling by ensuring that critical errors are logged and propagated, preventing silent failures. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| **Medium** – Improper exception handling can cause silent failures, prevent correct error logging, and lead to undefined program behavior. While it might not directly lead to security breaches, it can result in vulnerabilities if exceptions are not correctly handled and propagated. | **Medium** – Exception misuse is common in codebases where error handling isn't prioritized. Catching general exceptions without proper logging or propagation is a frequent issue. | **Medium** – Refactoring exception handling can involve extensive code review and validation, but it is critical to ensure that errors are logged, propagated, or handled appropriately. | **High** – Correct handling of exceptions is crucial for error recovery and stability of the application. High priority should be given to refactoring improper exception handling early in the development lifecycle. | **2** – This issue is of moderate severity, affecting both error recovery and program stability if not handled correctly. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 (LTS) | C++ S3626 – Detect improper use of exceptions | SonarQube detects improper exception handling, including catching general exceptions, missing error logging, and issues with exception propagation. |
| Cppcheck | 2.9 | MISRA C++ Rule 15-5-1 – Catch appropriate errors | Cppcheck checks for violations of exception safety, including catching too general exceptions and lack of proper handling for runtime errors. |
| Clang-Tidy | 14.0 | cert-err52-cpp – Ensure safe exception handling | Clang-Tidy flags improper uses of exception handling, including catch-all blocks that may lead to silent failures and unhandled exceptions. |
| PVS-Studio | 7.15 | V580 – Detect improper exception usage | PVS-Studio identifies improper exception use, including cases where specific exceptions should be caught instead of general ones, and ensures error logging. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Correct Resource Management** |
| --- | --- | --- |
| Resource Management | STD-008-RSC | Correct resource management is essential for ensuring that dynamically allocated resources, such as memory, file handles, and network connections, are properly allocated and released. Failing to manage resources correctly can result in memory leaks, data corruption, and system instability. Resource management errors can lead to vulnerabilities like double free errors or resource exhaustion. |

| **Noncompliant Code** |
| --- |
| File is opened but not closed, leading to a resource leak. |
| #include <fstream>  void read\_file(const std::string& filename) {  std::ifstream file(filename);  if (!file) {  // Handle file open error  return;  }  // File is not closed  } |

| **Compliant Code** |
| --- |
| RAII ensures the file is closed automatically when going out of scope. |
| #include <fstream>  void read\_file(const std::string& filename) {  std::ifstream file(filename);  if (!file) {  // Handle file open error  return;  }  // File is closed automatically when 'file' goes out of scope  } |

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

| **Principles(s):** The **RAII (Resource Acquisition Is Initialization)** principle ensures that resources like memory, file handles, or network connections are properly managed and released. RAII ties the lifecycle of the resource to the lifetime of an object, preventing resource leaks by ensuring that resources are automatically released when an object goes out of scope. In the compliant code example, RAII ensures that the file is closed when the function scope ends, preventing resource exhaustion or crashes due to resource leaks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| **High** — Resource mismanagement can lead to serious issues, such as file handle exhaustion, memory leaks, data corruption, or system instability. It can even be exploited to cause denial of service (DoS) attacks. | **Medium** — Resource leaks and mismanagement are frequent, especially in older codebases or when manual memory management is prevalent. However, modern C++ code tends to use RAII patterns, reducing the occurrence of such issues. | **Medium** — Fixing resource leaks involves reviewing and refactoring the code to ensure all resources are correctly handled, possibly introducing RAII or smart pointers to handle the automatic release of resources. | **High** — Preventing resource leaks should be addressed early in the development process to avoid system instability and security vulnerabilities like denial of service. | **2** — Resource management issues are of moderate to high severity and need to be addressed through proper techniques like RAII. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.18.1 | Memory Leak Detection | Valgrind helps detect memory leaks and resource management issues in C++ applications by analyzing runtime memory allocations and deallocations. |
| Cppcheck | 2.9 | cert-msc32-c – Proper Resource Handling | Cppcheck performs static analysis to flag improper resource management, such as missing close() calls for file streams or double-free scenarios. |
| Clang-Tidy | 14.0 | cert-msc31-c – Detect Leaked Resources | Clang-Tidy helps detect resource management issues in code, particularly ensuring proper RAII usage for file streams and other resource handles. |
| GCC (with -fsanitize=address) | 11.2 | Detect resource leaks at runtime | The -fsanitize=address option in GCC detects memory and resource leaks at runtime, ensuring that resources like file handles are properly managed. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Proper Input Validation** |
| --- | --- | --- |
| Input Validation | STD-009-VAL | Proper input validation is necessary to prevent unexpected or malicious inputs from compromising system functionality. By ensuring that inputs are properly validated, we can prevent security vulnerabilities such as buffer overflows, injection attacks, and improper resource handling. |

| **Noncompliant Code** |
| --- |
| Does not check if input string size exceeds buffer size. |
| #include <cstring>  void copy\_input(const char\* input) {  char buffer[10];  strcpy(buffer, input); // Potential buffer overflow  } |

| **Compliant Code** |
| --- |
| Uses safer alternative strncpy and checks the size of input. |
| #include <cstring>  void copy\_input(const char\* input) {  char buffer[10];  strncpy(buffer, input, sizeof(buffer) - 1);  buffer[sizeof(buffer) - 1] = '\0'; // Ensures null termination  } |

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

| **Principles(s):** The **Input Validation and Error Handling** principle is crucial for this standard. Proper input validation ensures that the inputs are correctly checked before they are processed, preventing buffer overflows, injection attacks, and other security breaches. This principle applies directly to this standard as it emphasizes validating inputs to ensure correct functionality and prevent vulnerabilities due to unexpected or malicious inputs. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| **High** – Buffer overflows and improper input validation can lead to severe security risks, including arbitrary code execution or system compromise. | **Medium** – Many legacy systems and even modern applications face risks from improper input validation, especially when dealing with user-supplied data. | **Medium** – Correcting improper input validation requires reviewing all instances where input is handled and ensuring bounds checks are applied. Automated tools can assist in detecting improper validation practices. | **High** – Addressing input validation issues should be a top priority, as it can prevent critical vulnerabilities that may lead to security breaches. | **1** – This represents a high-priority security issue, requiring immediate remediation to prevent exploits or crashes caused by improper input handling. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 (LTS) | C++ S5787 – Detect improper input validation | SonarQube helps detect issues with improper input handling, flagging dangerous string manipulations or unchecked input bounds. |
| Clang-Tidy | 14.0 | cert-str34-c – Ensure proper input validation | Clang-Tidy identifies unsafe input handling operations, particularly around buffer overflows, helping ensure that all input is correctly validated. |
| Coverity | 2023.3 | CWE-120 – Buffer Copy Without Checking Size of Input | Coverity helps detect buffer overflows due to improper input validation, ensuring bounds checks are applied and input sizes are validated. |
| Flawfinder | 2.0.19 | STR31-C – Detect unsafe string handling | Flawfinder detects dangerous input operations, including unchecked buffer copies and unsafe string functions, ensuring that all inputs are validated. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Error Handling | STD-010-ERR | Proper error handling ensures that the program can handle unexpected conditions gracefully and recover where possible. Failing to handle errors correctly can result in undefined behavior, crashes, or data corruption. Robust error handling practices mitigate these risks and improve software reliability. |

| **Noncompliant Code** |
| --- |
| Error is returned, but no action is taken by the caller. |
| int divide(int a, int b) {  if (b == 0) {  return -1; // Error code for divide by zero  }  return a / b;  }  void process() {  int result = divide(10, 0);  // Result is not checked, leading to unexpected behavior  } |

| **Compliant Code** |
| --- |
| Error is properly handled by the caller. |
| #include <iostream>  int divide(int a, int b, int& result) {  if (b == 0) {  return -1; // Error code for divide by zero  }  result = a / b;  return 0;  }  void process() {  int result;  if (divide(10, 0, result) != 0) {  std::cerr << "Error: Cannot divide by zero\n";  } else {  std::cout << "Result: " << result << "\n";  }  } |

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

| **Principles(s):** The **Error Handling** principle ensures that errors are detected and handled correctly to prevent unexpected behavior, crashes, or corrupted data. By handling divide-by-zero errors and other critical failures effectively, we ensure that the program remains stable and reliable under various circumstances. This principle supports proper error handling techniques to avoid leaving the program in an inconsistent or unsafe state, especially in cases of division or similar operations that could cause crashes if unchecked. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| **High** – Errors like divide-by-zero can cause crashes or undefined behavior, potentially leading to data corruption or system failure. | **Medium** – Errors are relatively common in production systems, particularly in complex arithmetic operations or code that doesn't implement robust error handling. | **Medium** – Implementing error handling requires code refactoring, but automated tools and systematic reviews can help reduce manual overhead. | **High** – Critical errors should be caught and handled to ensure system reliability and prevent failures. | **2** – This represents a moderate risk but a high-priority issue to resolve to ensure program stability and error recovery. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 (LTS) | C++ S5644 – Detect unchecked errors | SonarQube detects cases where errors, especially those returned by functions, are not handled correctly. |
| Cppcheck | 2.9 | MISRA C++ Rule 18-0-1 | Cppcheck performs static analysis to flag unchecked errors and suggests fixes for improving error handling. |
| Clang-Tidy | 14.0 | cert-err34-c – Proper error handling | Clang-Tidy flags improper error handling in functions and ensures that error codes are checked before continuing. |
| PVS-Studio | 7.15 | V1074 – Detect unchecked return values | PVS-Studio identifies unchecked function return values and prompts the developer to handle potential error conditions. |

### 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 plays a vital role in ensuring that security practices are consistently applied throughout the software development lifecycle. Green Pace's existing DevOps infrastructure can be enhanced by integrating security at every stage of development, transitioning from DevOps to DevSecOps. This approach embeds security measures into the process early and ensures that they remain active throughout the lifecycle, as seen in the DevSecOps diagram.

* **Assess and Plan:** During the initial planning phase, the security landscape is evaluated, regulatory changes are reviewed, and potential risks are identified. Automation tools can assist in threat modeling and impact analysis, providing insights into new security threats and potential vulnerabilities. Tools like SonarQube can automate code analysis at this stage to assess vulnerabilities early.
* **Design:** Security-focused design patterns, such as OWASP principles and test-driven development, ensure that security is built into the design from the start. Automated tools like Coverity can be used to scan design documents and architectures for potential security flaws, ensuring that any security gaps are addressed before coding begins.
* **Build:** Automated security checks can be integrated into the build process. Tools like Clang-Tidy can be used to ensure that secure coding standards are adhered to during development. This process ensures that the codebase remains secure and that any vulnerabilities introduced during development are caught before they reach production.
* **Verify and Test:** Vulnerability scanning and compliance testing can be automated using tools like Cppcheck and PVS-Studio. These tools help detect issues related to data handling, memory management, and secure coding practices. They ensure that the software is rigorously tested against secure coding standards and that any vulnerabilities are flagged and resolved before deployment.
* **Transition and Health Check:** When transitioning software to production, automated security checks can verify that security configurations, such as encryption settings and authentication measures, are correctly implemented. Tools like OWASP ZAP can run penetration tests to identify any vulnerabilities before the software goes live.
* **Monitor and Detect:** Once the software is in production, automated monitoring tools like SIEM (Security Information and Event Management) can detect intrusions or abnormal behaviors in real-time. These tools can automate threat detection and provide immediate alerts, ensuring that security incidents are quickly identified and addressed.
* **Respond:** Automation can also help during the response phase by automatically isolating affected systems, rolling back to previous versions, or applying security patches. This minimizes the damage from security incidents and ensures a rapid response.
* **Maintain and Stabilize:** Automated tools ensure that systems remain stable after a security incident. They help in assessing the system against the security baseline and returning the system to a secure state.

By embedding automation throughout the DevSecOps lifecycle, Green Pace can maintain a strong security posture, reduce manual overhead, and ensure compliance with secure coding standards.

### 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-INT | High | Medium | Medium | High | 2 |
| STD-003-STR | High | Medium | Medium | High | 2 |
| STD-004-SQL | High | Medium | Medium | High | 1 |
| STD-005-MEM | High | Medium | Medium | High | 2 |
| STD-006-ASR | Medium | Low | Low | Medium | 2 |
| STD-007-EXC | Medium | Medium | Medium | High | 2 |
| STD-008-RSC | High | Medium | Medium | High | 2 |
| STD-009-VAL | High | Medium | Medium | High | 1 |
| STD-010-ERR | High | Medium | Medium | High | 2 |

### 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 | Ensures that stored data, such as databases or files on hard drives, is secure by encrypting it. This is vital in case of physical theft of the device or unauthorized access to storage systems, preventing data breaches. |
| Encryption in flight | Protects data while it is being transmitted over networks, such as emails or web requests, ensuring that intercepted data remains secure. This prevents man-in-the-middle attacks and unauthorized access during communication. |
| Encryption in use | Safeguards data as it is being processed by applications. This type of encryption ensures that even while data is actively being used in memory or by programs, it remains protected from unauthorized access. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Verifies the identity of users or systems attempting to access resources, typically via login credentials. This is critical for ensuring that only legitimate users can interact with the system. |
| Authorization | Determines what authenticated users are allowed to do. Once users are authenticated, authorization rules decide their level of access to files, applications, and data. |
| Accounting | Tracks and records the actions taken by users within the system. This allows for monitoring and auditing user activities, ensuring transparency, and identifying potential security violations. |

### 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.

**Mapping Principles to Coding Standards**

1. STD-001-CPP: Data Type Correctness Policy
   1. Principle: Type Safety
   2. Justification: Type safety ensures that data is handled correctly and prevents vulnerabilities caused by type mismatches or incorrect conversions. This principle directly supports data type correctness by preventing type-related errors that could compromise system integrity.
2. STD-002-INT: Data Value Correctness Policy
   1. Principle: Input Validation
   2. Justification: Proper input validation ensures that all values are within acceptable bounds, preventing data value errors like overflows or invalid data. This supports data value correctness by ensuring that values used in the system are valid and secure.
3. STD-003-STR: String Correctness Policy
   1. Principle: Buffer Management
   2. Justification: Buffer management helps prevent buffer overflows or underflows when handling strings, which is crucial for maintaining string correctness. This principle ensures that strings are correctly sized and processed securely.
4. STD-004-SQL: SQL Injection Policy
   1. Principle: Input Validation
   2. Justification: Input validation ensures that user inputs are sanitized before being included in SQL queries, preventing SQL injection attacks. This principle directly addresses SQL injection vulnerabilities by ensuring inputs are treated securely.
5. STD-005-MEM: Memory Protection Policy
   1. Principle: Proper Memory Management
   2. Justification: Proper memory management ensures that memory is allocated, accessed, and freed correctly, preventing memory leaks, buffer overflows, or security breaches. This principle helps maintain memory safety and system stability.
6. STD-006-ASR: Assertions Policy
   1. Principle: Error Handling
   2. Justification: Assertions help identify unexpected states during development, ensuring that errors are caught early. However, assertions must not be used in production to handle user inputs or conditions, and proper error handling should be employed.
7. STD-007-EXC: Exceptions Policy
   1. Principle: Error Handling and Control Flow
   2. Justification: Effective error handling ensures that exceptions are used correctly, allowing programs to handle errors gracefully without exposing the system to vulnerabilities. This principle ensures that exceptions are used to control the flow and catch unexpected conditions securely.
8. STD-008-RSC: Resource Management Policy
   1. Principle: Resource Management
   2. Justification: Resource management ensures that resources such as file handles and memory are properly allocated and released. This prevents resource leaks, ensuring the system runs efficiently and securely without exhausting available resources.
9. STD-009-VAL: Input Validation Policy
   1. Principle: Input Validation
   2. Justification: Input validation is crucial to preventing vulnerabilities such as buffer overflows, injection attacks, and improper resource handling. This principle ensures that all inputs are checked and processed securely before being handled by the system.
10. STD-010-ERR: Error Handling Policy
    1. Principle: Error Reporting and Handling
    2. Justification: Proper error handling ensures that errors are caught, logged, and addressed without leaving the system in an insecure or inconsistent state. This principle ensures that errors are managed in a way that supports system reliability and security.

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 | 09/22/2024 | Module 3 Submission | Joseph Dengler |  |
| 1.3 | 10/13/2024 | Final Project Submission | Joseph Dengler |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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