**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 | Every input received from a user, file, or external system should be thoroughly validated before being processed. Validation helps prevent a range of attacks such as buffer overflows, injection attacks, and data corruption. Data should be checked for correct type, size, format, and range. Whitelisting acceptable input is preferred over blacklisting. |
| 1. Heed Compiler Warnings | Compiler warnings serve as early indicators of problematic code, including issues like uninitialized variables, data truncation, or type mismatches. Ignoring these warnings could lead to security vulnerabilities, especially in memory management or pointer usage. Developers should configure compilers to treat warnings as errors and address them promptly to ensure robust code. |
| 1. Architect and Design for Security Policies | Security must be integrated during the planning and design stages, not as an afterthought. A secure architecture includes defined access control mechanisms, secure data flows, encryption strategies, and a clear separation of privileges. Threat modeling and security reviews at the design phase can identify and mitigate risks early in the development cycle. |
| 1. Keep It Simple | Complex code increases the likelihood of bugs, which may become security vulnerabilities. By simplifying control flows, avoiding unnecessary dependencies, and writing modular code, developers can reduce the attack surface. Code that is easier to read is also easier to audit and maintain. |
| 1. Default Deny | By default, systems should refuse access unless explicitly permitted. This principle helps avoid unauthorized access resulting from misconfigurations or overlooked permissions. It ensures a secure baseline by requiring deliberate configuration to grant access rights, which reduces risk. |
| 1. Adhere to the Principle of Least Privilege | Applications, users, and processes should operate using only the permissions necessary to perform their intended functions. For example, a function that only reads data should not have write access. Enforcing this principle limits the potential damage in case of a breach or error. |
| 1. Sanitize Data Sent to Other Systems | Data shared with other systems, such as databases or web services, should be cleansed of potentially harmful content. This includes removing or encoding special characters, stripping executable code, and validating data formats. Proper sanitation prevents injection attacks and ensures that downstream systems function securely. |
| 1. Practice Defense in Depth | No single security mechanism is foolproof. Defense in depth involves layering multiple security controls across the application stack, such as input validation, authentication, logging, and encryption. If one control fails, others continue to protect the system. This layered approach is essential for resilient systems. |
| 1. Use Effective Quality Assurance Techniques | Thorough testing, including unit tests, integration tests, static code analysis, and peer reviews are vital for catching both functional defects and security vulnerabilities. QA processes should include security-specific tests, such as fuzz testing and penetration testing, to validate the resilience of the software. |
| 1. Adopt a Secure Coding Standard | Following a recognized secure coding standard like the SEI CERT C++ standard provides developers with vetted best practices. These guidelines reduce the likelihood of introducing vulnerabilities through common programming mistakes. Regular training and enforcement of these standards promote a security-focused development culture. |

### 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** | **Declare Objects with Appropriate Storage Duration** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | In C++, the storage duration of an object determines its lifetime and visibility. Using inappropriate storage duration can lead to severe issues like memory leaks, dangling pointers, or data races. For example, allocating dynamic memory for short-lived objects wastes resources, while relying on automatic duration for long-lived objects may cause premature destruction. This rule ensures that objects are declared in a way that matches their intended lifespan and usage context, which is especially important in large, secure systems where resource management and predictable behavior are critical. |

| **Noncompliant Code** |
| --- |
| In this example, dynamic memory (new) is used to allocate a simple integer value that is only used within the scope of the function. This is inefficient and potentially dangerous. If the delete statement is accidentally omitted or skipped due to an exception or early return, the memory will not be freed, resulting in a memory leak. Additionally, using pointers for trivial operations adds unnecessary complexity and opens the door to use-after-free or double-free errors if mismanaged. |
| void calculate() {  int\* value = new int(42); // Dynamically allocated for no reason  std::cout << \*value << std::endl;  delete value; // Must remember to clean up  } |

| **Compliant Code** |
| --- |
| This version improves both readability and safety by allocating the integer on the stack using automatic storage duration. The object is destroyed automatically when the function scope ends, eliminating the need for manual deallocation. This not only avoids memory leaks but also reduces the surface area for security issues such as heap corruption or dangling pointers. Using stack memory here is more efficient and adheres to the principle of least complexity. |
| void calculate() {  int value = 42; // Stored on the stack, automatically managed  std::cout << value << std::endl;  } |

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

| **Principles(s):**   * **4. Keep It Simple:** This principle promotes clarity and minimalism in code. By avoiding dynamic memory for trivial use cases, the code becomes simpler and easier to maintain and audit. * **8. Practice Defense in Depth:** Automatic storage avoids risks that could emerge if heap management fails. It’s one of many layers that help secure the application from misuse or mishandling. * **10. Adopt a Secure Coding Standard:** This standard is part of SEI CERT’s best practices. Enforcing predictable object lifetimes adheres to recognized security coding guidelines.   **How the Principles Map:**   * *Keep It Simple* reduces the attack surface by avoiding complex and unnecessary memory management logic. * *Defense in Depth* ensures that even if higher-level logic fails, memory safety is reinforced by default behaviors. * *Secure Coding Standards* provide the underlying justification for automatic storage duration as the preferred approach when applicable. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.4 | cpp:S6003 | Detects unnecessary dynamic allocations when stack memory is sufficient. |
| Cppcheck | 2.10 | memleak, uninitvar | Reports memory leaks and uninitialized variables often related to heap misuse. |
| Clang-Tidy | 17.0 | modernize-use-auto, clang-analyzer-cplusplus.NewDeleteLeaks | Highlights unnecessary heap allocations and warns about leaks. |
| CodeSonar | 7.3 | MEM.LEAK, MEM.SCOPE | Performs deep static analysis to track storage duration and improper allocation. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Ensure that operations on signed integers do not result in overflow** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Signed integer overflow in C++ leads to undefined behavior, which can cause unpredictable results, application crashes, or exploitable vulnerabilities. Unlike unsigned integers, which wrap around on overflow, signed integers exhibit no guaranteed behavior when their value exceeds the representable range. To ensure predictable and secure program behavior, all arithmetic operations—especially those involving user input or large values—must be checked for potential overflow before execution. |

| **Noncompliant Code** |
| --- |
| In this example, adding two large positive integers can cause overflow if the resulting value exceeds the maximum range representable by an int. This is undefined behavior and may not be detected at compile-time or even at runtime, making it dangerous and unpredictable. The application could crash, behave erratically, or provide incorrect results, especially if the values originate from user input. |
| int add(int a, int b) {  return a + b; // No overflow check  } |

| **Compliant Code** |
| --- |
| This version includes an explicit check for potential overflow before performing the addition. If an overflow is detected, the function logs a warning and returns a safe fallback value. This protects the integrity of the application and ensures the system behaves in a well-defined, secure manner under all conditions. |
| #include <limits>  #include <iostream>  int add(int a, int b) {  if ((b > 0) && (a > std::numeric\_limits<int>::max() - b)) {  std::cerr << "Overflow detected!" << std::endl;  return -1; // Handle safely  }  if ((b < 0) && (a < std::numeric\_limits<int>::min() - b)) {  std::cerr << "Underflow detected!" << std::endl;  return -1; // Handle safely  }  return a + b; // Safe to add  } |

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

| **Principle(s):**   * **1. Validate Input Data:** The operands must be validated before performing arithmetic operations to ensure they won’t trigger overflow or underflow. * **2. Heed Compiler Warnings:** Some compilers warn about suspicious arithmetic patterns. Addressing these can prevent overflow early. * **10. Adopt a Secure Coding Standard:** The practice of bounds checking before signed operations is an SEI CERT standard, reinforcing secure behavior. * **8. Practice Defense in Depth:** Overflow checks add a layer of runtime safety to catch values that compile-time tools or prior validations may miss.   **How the Principles Map:**   * *Validate Input Data* ensures values passed into arithmetic functions are within safe bounds. Preventing integer overflows is an essential form of data validation. * *Heed Compiler Warnings* addresses build-time hints that could identify unsafe arithmetic. Combined with static analysis, this reinforces safer code. * *Adopt a Secure Coding Standard* ensures developers follow vetted rules like INT32-C: Ensure that operations on signed integers do not result in overflow. * *Defense in Depth* supports runtime protection when static checks are insufficient, ensuring even unexpected edge cases don’t lead to failure. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.4 | c:S2755 | Flags arithmetic without overflow protections |
| Cppcheck | 2.13 | signedOverflow, arithmeticOverflow | Identifies risky arithmetic and missing checks |
| Clang-Tidy | 17.0 | cppcoreguidelines-pro-bounds-pointer-arithmetic | Detects dangerous use of pointer and int math |
| Coverity | 2024.03 | INT\_OVERFLOW | Pinpoints signed overflow violations at runtime |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Range Check Element Access** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | When accessing characters in a std::string using the [] operator, there is no automatic check to ensure that the specified index is within the string’s length. If the index is invalid—either too large or negative—this results in undefined behavior, which can crash the program, corrupt memory, or open up security holes. Using the .at() method provides safer access, as it checks the index and throws a standard exception if it’s invalid. This allows the program to handle the error gracefully instead of failing unexpectedly. Range checking is especially critical when working with data that comes from users or external systems. |

| **Noncompliant Code** |
| --- |
| This code assumes the index will always be valid when accessing a string using the [] operator. If the input function returns a value outside the valid range, it will lead to a crash or unpredictable behavior. |
| #include <string>  size\_t getIndex();  void accessChar() {  std::string data = "example";  char ch = data[getIndex()]; // No bounds check  } |

| **Compliant Code** |
| --- |
| This safer version uses std::string::at() to access the character. If the index is invalid, an exception is thrown, which is caught and logged. This protects the program from unsafe memory access and improves fault tolerance. |
| #include <string>  #include <iostream>  #include <stdexcept>  size\_t getIndex();  void accessChar() {  std::string data = "example";  try {  char ch = data.at(getIndex()); // Bounds-checked  std::cout << "Character: " << ch << std::endl;  } catch (const std::out\_of\_range&) {  std::cerr << "Error: Index is outside the valid range." << std::endl;  }  } |

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

| **Principles(s):**   * **1. Validate Input Data:** The index returned from getIndex() could originate from untrusted or user-provided input. By checking that it is within the valid range using .at(), the program prevents illegal memory access. * **8. Practice Defense in Depth:** Combining bounds-checking and exception handling provides multiple layers of defense in case of invalid input. * **9. Use Effective Quality Assurance Techniques:** This standard can be reinforced with static analysis tools and exception-handling tests during the QA process. * **4. Keep It Simple:** Using .at() streamlines error handling and promotes readable, safer code compared to complex custom bounds checks.   **How the Principles Map:**   * *Validate Input Data* ensures only safe indices are used to access elements, especially when the index comes from external or user sources. * *Defense in Depth* applies by catching exceptions that would otherwise result in a crash or memory corruption. * *Effective QA* practices such as fuzz testing can be used to verify that edge cases (like out-of-range indices) are handled properly. * *Keep It Simple* is followed by relying on built-in language features (.at() and try-catch) instead of custom logic. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.4 | cpp:S3512 | Detects direct index access without bounds-checks in containers like string |
| Cppcheck | 2.13 | outOfBounds | Flags possible out-of-bounds access in strings and arrays |
| Clang-Tidy | 17.0 | clang-analyzer-cplusplus | Includes analyzers for unsafe memory access and exceptions |
| Fortify SCA | 2024 | BufferOverflow.String | Recognizes unsafe string access patterns and flags them |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Avoid Using Data from an Untrusted Source Without Validation** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Using external input directly in logic decisions—such as user roles, permissions, file paths, or system state—without first verifying that the input is valid can open a program to serious security flaws. Attackers can craft inputs that bypass security logic or exploit unexpected behaviors. In C++, where input is often read from CLI arguments, files, or network streams, validating assumptions about format, range, and type is critical. Input should always be checked against known constraints before it’s used in any condition that affects control flow or system behavior. |

| **Noncompliant Code** |
| --- |
| This example reads a username from input and uses it directly in a conditional check to simulate a login decision, without checking if the value is actually valid or safe. If input is not verified against a list of registered users, this comparison allows any user who guesses or supplies the string "admin" to gain elevated access. There’s no actual validation or authentication. |
| #include <string>  #include <iostream>  void login(const std::string& username) {  if (username == "admin") {  std::cout << "Admin access granted.\n";  } else {  std::cout << "Standard user access.\n";  }  } |

| **Compliant Code** |
| --- |
| This improved version checks the input against a known list of trusted users and includes authentication logic before granting access. This ensures only validated identities affect system decisions. Usernames are validated against an approved list before they’re used in any condition that affects the program’s control flow. This eliminates input-based privilege escalation attacks. |
| #include <string>  #include <iostream>  #include <set>  bool isValidUser(const std::string& user) {  static const std::set<std::string> allowedUsers = {"admin", "user1", "guest"};  return allowedUsers.find(user) != allowedUsers.end();  }  void login(const std::string& username) {  if (!isValidUser(username)) {  std::cout << "Access denied.\n";  return;  }  if (username == "admin") {  std::cout << "Admin access granted.\n";  } else {  std::cout << "Standard user access.\n";  }  } \*ptr = 42;  std::free(ptr);  ptr = nullptr; // Prevent dangling pointer  // ptr is not used after being freed  } |

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

| **Principle(s):**   * **1. Validate Input Data:** All user-supplied input (e.g., usernames) is validated against a trusted set of allowed values. This ensures that only well-formed and authorized input can influence access control logic. * **5. Default Deny:** Access is denied to any username not explicitly approved, enforcing a secure default behavior. * **7. Sanitize Data Sent to Other Systems:** Although not sent to another system, the data is processed internally in a security-sensitive way (affecting logic decisions). Treating it with sanitation ensures it doesn’t compromise system behavior. * **8. Practice Defense in Depth:** Even if authentication is bypassed elsewhere, this input validation offers another line of defense against unauthorized access.   **How the Principles Map:**   * Validate Input Data is essential here, as it ensures that only approved inputs are allowed to impact critical control flow decisions such as admin access. * Default Deny is followed by restricting access to a whitelist of trusted users; any unrecognized input is explicitly rejected, eliminating unintended access. * Sanitize Data Sent to Other Systems applies because the input is interpreted by the program’s logic engine; validating that it matches expected values mitigates logic-based attacks. * Defense in Depth is achieved by enforcing strict input controls as one of several layers that defend against privilege escalation or misuse of program logic. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 LTS | c:S5131 | Detects use of untrusted input without validation |
| Cppcheck | 2.12 | useStlAlgorithm | Flags insecure or direct logic checks |
| Fortify | 23.1 | SQL Injection Check | Identifies potential logic flaws from unvalidated input |
| CodeQL | 2.14.4 | cpp/unvalidated-input | Traces insecure usage of input in control structures |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Do Not Access Freed Memory** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | One of the most dangerous and common memory-related vulnerabilities in C++ is accessing memory that has already been freed. Once deallocated, that memory location may be reused or returned to the operating system. Any future access through dangling pointers results in undefined behavior, such as crashes, data corruption, or security breaches. Developers must ensure that after memory is released, all references to it are removed or nullified to avoid use-after-free conditions. |

| **Noncompliant Code** |
| --- |
| This function releases memory with delete, then continues to access the pointer, triggering undefined behavior. Although ptr still holds an address, the memory it pointed to is no longer valid. Writing to it after deletion can overwrite unknown memory, leading to unpredictable program results. |
| void useAfterFree() {  int\* ptr = new int(100);  delete ptr;  \*ptr = 50; // Undefined behavior: accessing freed memory  } |

| **Compliant Code** |
| --- |
| This version deletes the memory and then sets the pointer to nullptr, effectively preventing future misuse. Setting ptr to nullptr ensures any later use can be safely checked. Most modern debuggers and static analyzers can catch null dereference errors but not dangling pointer misuse — so nulling the pointer helps both security and debugging. |
| void safeUse() {  int\* ptr = new int(100);  delete ptr;  ptr = nullptr; // Prevent dangling access  } |

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

| **Principles(s)**   * **1. Validate Input Data**: Indirectly applies, as the use of nullptr helps validate whether the pointer can be safely dereferenced. * **4. Keep It Simple:** Simplifying memory lifecycle by nullifying pointers reduces complexity and makes intent clearer. * **8. Practice Defense in Depth:** Nullifying the pointer after deletion provides a second layer of protection, even if code logic fails. * **9. Use Effective Quality Assurance Techniques:** Use-after-free errors are hard to detect manually, but automated tools can identify patterns associated with these issues. * **10. Adopt a Secure Coding Standard:** This practice follows SEI CERT recommendations for managing memory in secure systems.   **How the Principles Map:**   * Validate Input Data applies by treating memory references like input and validating pointer state before use. * Keep It Simple through the predictable pattern of delete then nullptr. * Defense in Depth is practiced by layering manual nulling with static analysis safeguards. * Effective QA Techniques encourage using memory debugging tools that catch these vulnerabilities. * Secure Coding Standards recommend clearing pointers after deletion to eliminate dangling references. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.2 | cpp:S9203 | Flags improperly freed memory or improper pointer usage. |
| Cppcheck | 2.12.1 | memleak, useAfterFree | Detects memory leaks and use-after-free vulnerabilities. |
| Clang-Tidy | 17.0.1 | clang-analyzer-cplusplus.NewDeleteLeaks | Identifies memory leaks and dangling pointers. |
| Valgrind | 3.21.0 | memcheck | Runtime memory checker for detecting use-after-free. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use Static Assertions for Constant Expression Checks** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions are useful for debugging, but assert() can incur runtime overhead and calls abort(), which is problematic in production and unsuitable for embedded or server systems. For compile-time checks, such as ensuring structure layout or constant expression constraints, a staticassertion or preprocessor directive (like #error) is safer and more efficient. This ensures correctness without runtime cost or behavior changes in release builds. |

| **Noncompliant Code** |
| --- |
| The assert() macro checks a property of a struct, which is vital to correct operation. But this check may be removed in release builds with NDEBUG. This assertion validates structure size — a critical assumption. If compiled with NDEBUG, this check disappears, making behavior dependent on build configuration, which risks unexpected bugs. |
| #include <assert.h>  struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };  int func(void) {  assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| This example uses a compile-time preprocessor directive to ensure structure size is validated regardless of build flags. This guarantees the check runs at compile time, halting compilation if the condition fails. It’s safer, universal across builds, and prevents size-related bugs. |
| struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };  #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))  #error "Structure must not have any padding"  #endif |

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

| **Principles**   * **2. Heed Compiler Warnings:** Ensures code is caught early through compile-time analysis rather than relying on potentially disabled runtime checks. * **9. Use Effective Quality Assurance Techniques:** Enforces structure constraints without dependency on runtime conditions, improving reliability and portability. * **10. Adopt a Secure Coding Standard:** Following SEI CERT's guidance ensures all checks remain valid across builds.   **How the Principles Map:**   * *Heed Compiler Warnings* is fulfilled by shifting checks to compile-time where violations are unignorable. * *Effective QA Techniques* are enhanced by using compile-time assertions that work across toolchains and configurations. * *Secure Coding Standards* promote uniform practices like static assertions to ensure consistency and security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 16.0+ | |  | | --- | |  |  |  | | --- | | modernize-use-static-assert | | Suggests replacing assert() with static\_assert when the condition is a constant expression. |
| GCC | 13.1+ | Built-in Support | static\_assert is natively supported and enforced at compile time. Violations halt compilation. |
| Cppcheck | 2.12 | Manual Rule / Custom Check | Can be configured to flag uses of assert() in conditions evaluable at compile time. |
| SonarQube | 10.x | cpp:S5863 (if enabled in custom profile) | |  | | --- | |  |  |  | | --- | |  |   Flags runtime assert() usage that could be replaced with compile-time checks. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Do Not Leak Resources When Handling Exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | When an exception is thrown, normal control flow is disrupted, potentially bypassing any resource deallocation logic that follows. If dynamic memory or other system resources are not released properly, this can cause memory leaks, resource exhaustion, or inconsistent system states. The best way to prevent resource leaks is to use **RAII (Resource Acquisition Is Initialization)**, where resources are tied to object lifetimes and automatically cleaned up when the object goes out of scope. |

| **Noncompliant Code** |
| --- |
| This code allocates a resource with new and calls a method that may throw an exception. If the exception occurs, delete is never called, causing a memory leak. If process\_item() throws an exception, the call to delete pst; is skipped. This results in a memory leak since the exception interrupts cleanup. |
| #include <new>  struct SomeType {  SomeType() noexcept;  ~SomeType();  void process\_item() noexcept(false);  };  void f() {  SomeType \*pst = new (std::nothrow) SomeType();  if (!pst) return;  try {  pst->process\_item();  } catch (...) {  throw;  }  delete pst;  } |

| **Compliant Code** |
| --- |
| In this version, the resource is properly deleted inside the catch block to ensure that cleanup occurs even when an exception is thrown. The catch block ensures pst is deleted even if an exception occurs. Alternatively, modern C++ would recommend using a smart pointer (e.g., std::unique\_ptr) or stack-allocated object to fully embrace RAII and avoid manual cleanup altogether. |
| #include <new>  struct SomeType {  SomeType() noexcept;  ~SomeType();  void process\_item() noexcept(false);  };  void f() {  SomeType \*pst = new (std::nothrow) SomeType();  if (!pst) return;  try {  pst->process\_item();  } catch (...) {  delete pst;  throw;  }  delete pst;  } |

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

| **Principle(s):**   * **9. Use Effective Quality Assurance Techniques:** This standard reflects the need for thorough exception handling in unit tests and peer reviews to ensure resource cleanup is verified in both success and failure paths. * **8. Practice Defense in Depth:** Adding exception-safe cleanup mechanisms ensures a secondary layer of protection when something unexpected occurs. * **4. Keep It Simple**: By adopting RAII or ensuring catch blocks handle deallocation, the code remains clear and robust. * **10. Adopt a Secure Coding Standard:** This practice enforces proper memory management, especially when exceptions might divert control flow.   **How the Principles Map:**   * Effective QA validates that exception paths are safe, and resources aren't leaked using static analysis and test coverage. * Defense in Depth applies by offering cleanup logic even if something goes wrong. * Keep It Simple encourages minimal, easy-to-follow exception-safe designs. * Secure Coding Standard adherence means developers avoid use-after-free or leak bugs, which are common in manual memory handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 16.0+ | bugprone-exception-escape | Detects functions that might throw exceptions and escape cleanup logic. |
| Cppcheck | 2.12 | memleak | Identifies potential memory leaks, including exception paths. |
| SonarQube | 10.x | cpp:S125 | Highlights exception flows that may bypass resource cleanup. |
| Coverity | 2023.3 | RESOURCE\_LEAK, UNCAUGHT\_EXCEPT | Finds missing deallocation in exception paths and unhandled exceptions. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Do Not Read Uninitialized Memory** |
| --- | --- | --- |
| Expressions | [STD-008-CPP] | Reading from uninitialized memory leads to undefined behavior. The contents of such memory are indeterminate and can vary across platforms and compilers. This may cause unpredictable behavior, security vulnerabilities (e.g., leaking stack contents), and data corruption. All variables, especially local stack variables, must be explicitly initialized before use. This ensures consistent behavior and avoids difficult-to-debug issues. |

| **Noncompliant Code** |
| --- |
| This function declares a local variable but never assigns it a value before use. The value of x is indeterminate. Returning it causes the function to produce undefined and potentially dangerous results, including leaking memory contents. |
| int get\_value() {  int x; // Uninitialized  return x; // Undefined behavior  } |

| **Compliant Code** |
| --- |
| This corrected version initializes the variable before it's used. By initializing x with a known value, the function avoids undefined behavior. This also makes the logic clearer and easier to maintain. |
| int get\_value() {  int x = 0; // Safe initialization  return x;  } |

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

| **Principle(s):**   * **1. Validate Input Data:** Ensuring variables have known and valid values before use is equivalent to validating the internal state of the program, which improves reliability and prevents corrupted logic flow. * **9. Use Effective Quality Assurance Techniques:** Detecting uninitialized variables is a common task for static analyzers and compilers with strict warning settings. These tools should be part of the QA pipeline. * **2. Heed Compiler Warnings:** Many compilers issue warnings when a variable may be used uninitialized. Developers should treat these as errors and correct them before deployment. * **4. Keep It Simple:** Initializing variables at declaration simplifies reasoning about code behavior and prevents subtle bugs related to default values or memory garbage.   **How the Principles Map:**   * Validate Input Data ensures variables begin with valid values, avoiding reliance on unpredictable memory states. * Effective QA Techniques enable automated tools to detect this problem early in development. * Heeding Compiler Warnings addresses undefined behavior that arises from poor variable handling. * Keeping It Simple encourages initializing variables to reduce ambiguity and boost code clarity. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12 | uninitvar | Identifies usage of variables that may be uninitialized. |
| Clang-Tidy | 15.0 | clang-analyzer-core.UndefinedBinaryOperatorResult | Detects undefined behavior from uninitialized values. |
| GCC | 13.1 | -Wuninitialized | Warns when a variable may be used before initialization. |
| Visual Studio | 2022 | /analyze static analysis | Detects uninitialized variable use through code analysis. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do Not Cast to an Out-of-Range Enumeration Value** |
| --- | --- | --- |
| Integers | [STD-009-CPP] | Casting arbitrary integers to an enumeration type without validation introduces the possibility of assigning values outside the defined enumerators. This can lead to undefined behavior, platform-dependent results, and serious logical flaws. Enumerations are often used in decision-making constructs such as switch statements or conditional checks. If a cast integer does not match any defined enum value, it may bypass critical logic branches or execute unintended code paths. In secure systems, this flaw can be exploited to disrupt control flow, elevate privileges, or expose vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This example directly casts an integer to an enum type without checking that the integer corresponds to a valid enum value. Even though the code tries to validate the enum afterward, the cast itself may already invoke undefined behavior, depending on the platform's enum implementation. It is unsafe to perform the cast before validation. |
| enum EnumType {  First,  Second,  Third  };  void f(int intVar) {  EnumType enumVar = static\_cast<EnumType>(intVar);  if (enumVar < First || enumVar > Third) {  // Attempt to handle invalid value  }  } |

| **Compliant Code** |
| --- |
| This example checks that the value is valid before performing the cast to an enumeration. By validating the range of intVar before casting, this version ensures that the cast is safe and the resulting value is a defined enumerator, maintaining program correctness. |
| enum EnumType {  First,  Second,  Third  };  void f(int intVar) {  if (intVar < First || intVar > Third) {  // Handle invalid input safely  return;  }  EnumType enumVar = static\_cast<EnumType>(intVar);  } |

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

| **Principles(s):**   * **1. Validate Input Data:** Ensures values used to construct enumerators are within the defined range. * **4. Keep It Simple:** Validating before casting maintains clarity and avoids undefined behavior. * **8. Practice Defense in Depth:** Preventing unsafe casts protects against edge case exploits, adding another layer of safety. * **9. Use Effective Quality Assurance Techniques:** Static analyzers and runtime checks can ensure enumeration safety and catch potential logic flaws early.   **How the Principles Map:**   * *Validate Input Data* ensures that values being cast to enums are explicitly checked for correctness to prevent undefined behavior. * *Keep It Simple* is reflected in the logical flow: check first, cast after. It avoids convoluted and risky casting patterns. * *Defense in Depth* is applied by combining explicit checks with language constraints and runtime safeguards. * *Effective QA Techniques* support enumeration handling through tools like static analyzers and boundary testing to catch misuse scenarios. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 16.0+ | clang-analyzer-core.UndefinedEnumCast | Detects and warns on improper casts to enums where the value may be invalid |
| Cppcheck | 2.12+ | invalidEnumCast (custom check) | Highlights suspicious casts not backed by valid range checking |
| SonarQube | 10.x | cpp:S3519 | Warns when casting to an enum is performed without a range check |
| GCC | 12+ | -Wconversion, -Wswitch-enum | Identifies type mismatch and incomplete enum handling in control statements |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Only Free Memory Allocated Dynamically** |
| --- | --- | --- |
| Memory Management | [STD-010-CPP] | Calling free() or delete on memory that was not dynamically allocated results in undefined behavior. This includes memory from stack variables, statically allocated memory, or placement new. Such mismanagement can lead to heap corruption, application crashes, or exploitable vulnerabilities. Therefore, deallocation should only occur on pointers returned by dynamic allocation functions like new, new[], malloc(), or calloc(). |

| **Noncompliant Code** |
| --- |
| In this code, memory from a statically allocated array is incorrectly passed to delete, even though it was never dynamically allocated. Although placement new was used to construct the object, delete must not be used to deallocate the memory. The memory was not allocated with new but comes from a local array. This results in undefined behavior. |
| #include <iostream>  struct S {  S() { std::cout << "S::S()\n"; }  ~S() { std::cout << "S::~S()\n"; }  };  void f() {  alignas(S) char space[sizeof(S)];  S\* s1 = new (&space) S;    // Incorrect: delete used on memory not from new  delete s1; // Undefined behavior  } |

| **Compliant Code** |
| --- |
| This version explicitly calls the destructor instead of using delete, which correctly handles the object without affecting memory it doesn’t own. By explicitly calling the destructor and avoiding delete, this version safely cleans up the object without corrupting memory. This is the correct approach for memory obtained via placement new. |
| #include <iostream>  struct S {  S() { std::cout << "S::S()\n"; }  ~S() { std::cout << "S::~S()\n"; }  };  void f() {  alignas(S) char space[sizeof(S)];  S\* s1 = new (&space) S;  // Proper cleanup: only call the destructor, no delete  s1->~S(); // Safe and defined  } |

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

| **Principles(s):**   * **1. Validate Input Data:** Improper handling of memory deallocation may stem from incorrect assumptions about where the memory came from; by treating all memory origins cautiously, we prevent misuse. * **10. Adopt a Secure Coding Standard:** This standard enforces safe memory management and aligns with industry best practices to avoid undefined behavior. * **9. Use Effective Quality Assurance Techniques:** Static analysis and memory safety tools should be used during development and testing to ensure compliance with memory management rules. * **3. Architect and Design for Security Policies:** Good memory management starts with a sound architecture that clearly distinguishes memory ownership and lifecycle responsibilities.   **How the Principles Map:**   * *Validate Input Data* promotes careful tracking of memory origins to ensure memory passed to delete or free() is valid. * *Adopt a Secure Coding Standard* ensures consistent memory handling practices that align with established best practices. * *Use Effective QA Techniques* supports catching improper memory operations through tools like Coverity and Cppcheck. * *Architect and Design for Security Policies* aids in structuring programs so memory lifecycle is clearly defined, reducing developer error. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.9 | memleak | Detects memory mismanagement and usage of unallocated or freed memory. |
| Clang-Tidy | 14.0 | cppcoreguidelines-no-malloc | Flags invalid memory usage and incorrect use of delete/free operations. |
| CodeSonar | 2024.1 | MEM-UNINIT | Identifies unsafe memory operations, including placement new and delete mismatches. |
| Coverity | 2024.03 | UNINIT.CALL | Detects undefined behavior due to improper object lifecycle management. |

### 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**

To ensure continuous compliance with the secure coding practices outlined in this policy, automation will be embedded throughout Green Pace’s development and operational workflows. The DevSecOps diagram shows how security is integrated into each phase of both pre-production and production, rather than being a separate final step. Leveraging Green Pace’s existing DevOps infrastructure, automation tools will be introduced or adjusted to align with security requirements at every stage of the pipeline.

**Pre-Production Phase**

At the planning and assessment stage, automated tools can assist in identifying potential risks tied to code changes, external dependencies, or regulatory updates. For instance, security-focused scanners can be configured to check for outdated libraries or known exploits during backlog refinement or code review. During the design stage, integrating static analysis tools like Clang-Tidy, Cppcheck, or SonarQube directly into the development environment helps enforce security guidelines by flagging issues such as unsafe pointer usage or improper memory access as code is written. When building the application, automation will enforce source integrity by using trusted repositories, and compiler settings will be applied automatically to detect problematic code patterns. These checks ensure that builds are both functional and secure from the ground up. In the testing and verification phase, automated testing frameworks will validate compliance through unit, integration, and security-specific testing. This includes techniques like fuzz testing and memory analysis, which help uncover hidden flaws such as buffer overflows or resource leaks.

**Transition to Production and Maintenance**

Before deployment, during the transition and configuration step, automated penetration testing and configuration validation tools will verify that the system is secure and properly set up. Any misconfiguration or policy violation will halt the deployment process until resolved. Once in production, tools that monitor logs, detect anomalies, and perform real-time analysis will provide continuous feedback about the application’s behavior. Automated alerts will trigger if abnormal patterns suggest a security breach or coding issue that violates defined standards. In the incident response phase, automated rollback scripts or container isolation procedures may be triggered immediately to contain threats and minimize damage. Finally, during post-deployment maintenance, automated regression tests and baseline verification checks will confirm the system remains secure after updates or hotfixes. These tools will ensure that fixes don’t unintentionally introduce new risks.

### 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 | High | Likely | Medium | High | 1 |
| STD-003-CPP | High | Likely | Low | High | 1 |
| STD-004-CPP | High | Likely | Medium | High | 1 |
| STD-005-CPP | High | Likely | Low | High | 1 |
| STD-006-CPP | Medium | Unlikely | Low | Medium | 2 |
| STD-007-CPP | High | Likely | Medium | High | 1 |
| STD-008-CPP | High | Likely | Low | High | 1 |
| STD-009-CPP | Medium | Likely | Low | Medium | 2 |
| STD-010-CPP | High | Likely | Low | High | 1 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest refers to the practice of securing data that is stored on any device or storage medium. This includes files saved to disk, backups, databases, or any other persistent storage solution. The policy at Green Pace requires all sensitive or confidential information, including user credentials, client data, and system configurations to be encrypted using advanced algorithms like AES-256. By enforcing this, the organization ensures that if storage devices are compromised or lost, the data remains unreadable without the appropriate decryption keys. This policy is critical for safeguarding long-term data storage, maintaining compliance with industry standards, and minimizing the impact of potential breaches. |
| Encryption in flight | Encryption in flight ensures that data being transmitted over a network is protected from eavesdropping or tampering. This policy mandates that all data exchanges between systems, including communications over the internet or within internal networks, must be encrypted using secure transmission protocols such as TLS (Transport Layer Security) version 1.2 or higher. For example, data transmitted via APIs, emails, or user login sessions must be protected with HTTPS to prevent unauthorized interception. The use of encrypted connections significantly reduces the risk of data leakage or manipulation during transfer, especially in remote work environments or public networks. |
| Encryption in use | Encryption in use addresses the protection of data while it is being actively processed by applications, particularly in memory (RAM). This policy is essential in situations where sensitive data, such as personally identifiable information (PII), authentication tokens, or encryption keys are temporarily handled in unencrypted form during program execution. At Green Pace, encryption in use is implemented through trusted execution environments (TEEs) or hardware-based isolation technologies like Intel Software Guard Extensions (SGX). These tools ensure that even if a system is compromised, sensitive data in memory is not exposed. The policy supports a comprehensive data protection strategy that goes beyond traditional storage and transmission encryption methods. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identity of users, systems, or devices before granting access to protected resources. Green Pace enforces strong authentication methods, including multi-factor authentication (MFA), for all users accessing administrative functions, secure databases, or remote services. This includes the use of passwords in combination with time-sensitive codes, biometric data, or security tokens. Authentication is the first line of defense against unauthorized access, reducing the chances of intrusion from compromised credentials. It ensures that only verified users can initiate sessions, access protected areas, or interact with sensitive components. |
| Authorization | Authorization determines what actions an authenticated user is permitted to perform within the system. At Green Pace, the authorization policy adheres to the principle of least privilege, users are granted the minimum level of access required to fulfill their responsibilities. For example, a front-line employee may have access to read client profiles but not to edit or delete them. Authorization roles are carefully assigned, regularly reviewed, and updated when a user’s role changes, or access is no longer required. This prevents privilege escalation attacks and minimizes potential damage from internal or external threats. |
| Accounting | Accounting, also known as auditing, involves the systematic tracking and logging of user activities. This includes recording login times, system commands, file access, database changes, and failed access attempts. Green Pace mandates that all user interactions with critical systems are logged and stored securely to support accountability and forensic investigations. These logs are reviewed periodically to detect abnormal behavior and ensure compliance with organizational policies. Accounting provides transparency into system use and helps identify potential security issues before they escalate. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 05/25/2025 | Module 3 Milestone | Oscar Rosa |  |
| 1.2 | 06/13/2025 | Project One | Oscar Rosa |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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