**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 | It's crucial to thoroughly check and sanitize any data that enters a system to prevent potential vulnerabilities or attacks, such as injection attacks or buffer overflows. By validating input data, developers ensure that only safe and expected information is processed, reducing the risk of exploitation by malicious actors. |
| 1. Heed Compiler Warnings | Developers should take compiler warnings seriously during software development, as they often signal potential weaknesses or errors that could be exploited by attackers. Paying attention to these warnings helps identify and address vulnerabilities early in the development process, enhancing the overall security of the software. |
| 1. Architect and Design for Security Policies | Security considerations should be integrated into the architectural and design phases of system development to ensure that security is a fundamental aspect of the system's structure. By aligning system architecture with security policies and requirements, developers can build robust and resilient systems that effectively mitigate security risks. |
| 1. Keep It Simple | Complexity is the enemy of security. By keeping systems and processes simple and straightforward, developers reduce the potential attack surface and make it easier to identify and address security vulnerabilities. Simplifying design and implementation also improves maintainability and reduces the likelihood of unintended consequences. |
| 1. Default Deny | Adopting a default-deny approach to system access means that all access is denied by default, and only explicitly authorized permissions are granted. This principle helps minimize the risk of unauthorized access and ensures that users and systems have access only to the resources they need to perform their tasks, enhancing overall security posture. |
| 1. Adhere to the Principle of Least Privilege | Granting users and systems the minimum level of access and permissions required to fulfill their roles reduces the potential impact of compromised accounts or malicious activities. By adhering to the principle of least privilege, developers limit the damage that could result from unauthorized access or misuse of privileges. |
| 1. Sanitize Data Sent to Other Systems | Before transmitting data to external systems or services, it's essential to sanitize and validate it to prevent the introduction of vulnerabilities or unintended consequences. This principle ensures that data exchanged between systems is safe and reliable, safeguarding against potential security breaches or data corruption. |
| 1. Practice Defense in Depth | Implementing multiple layers of security controls, such as firewalls, intrusion detection systems, and encryption, provides redundant protection and mitigates the impact of security breaches. By deploying defense-in-depth strategies, organizations enhance their resilience to cyber threats and improve overall security posture. |
| 1. Use Effective Quality Assurance Techniques | Robust quality assurance processes, including code reviews, testing, and vulnerability scanning, play a crucial role in identifying and remedying security issues throughout the software development lifecycle. By employing effective quality assurance techniques, developers ensure that software meets security requirements and is free from critical vulnerabilities. |
| 1. Adopt a Secure Coding Standard | Following established secure coding standards and best practices helps minimize the risk of introducing vulnerabilities into software during development. By adhering to a secure coding standard, developers mitigate common security risks and ensure that software is built with security in mind from the outset. |

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-206-DCL30-CPP | Declare objects const or constexpr if the objects’ values do not change after construction |

| **Noncompliant Code** |
| --- |
| Noncompliant Code Example (Differing Storage Durations)  In this noncompliant code example, the address of the variable c\_str with automatic storage duration is assigned to the variable p, which has static storage duration. The assignment itself is valid, but it is invalid for c\_str to go out of scope while p holds its address, as happens at the end of dont\_do\_this(). |
| #include <stdio.h>    **const** **char** \*p;  **void** dont\_do\_this(**void**) {  **const** **char** c\_str[] = "This will change";    p = c\_str; /\* Dangerous \*/  }    **void** innocuous(**void**) {  **printf**("%s\n", p);  }    **int** main(**void**) {    dont\_do\_this();    innocuous();  **return** 0;  } |

| **Compliant Code** |
| --- |
| Compliant Solution (Same Storage Durations)  In this compliant solution, p is declared with the same storage duration as c\_str, preventing p from taking on an indeterminate value outside of this\_is\_OK(): |
| **void** this\_is\_OK(**void**) {  **const** **char** c\_str[] = "Everything OK";  **const** **char** \*p = c\_str;    /\* ... \*/  }  /\* p is inaccessible outside the scope of string c\_str \*/ |

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

| **Principles(s):**  The Principle of Least Privilege to the coding standard STD-206-DCL30-CPP (Declare objects const or constexpr if the objects’ values do not change after construction), it maps directly to the "Data Type" principle by focusing on the type and mutability of variables. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.2 | Rule S5700 | Detects assignments of local variables with automatic storage duration to pointers with wider scope, helping to identify potential issues like dangling pointers. |
| GCC Compiler | 10.3 | “-werror=return-local-addr” | Warns and treats as errors the assignment of local variable addresses to pointers with broader scopes during compilation, enforcing compliance with const or constexpr usage policies. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-ERR34-C | Do not rely on unspecified behavior; This standard advises against relying on unspecified behavior in error handling and input validation. Relying on unspecified behavior can lead to unpredictable results and may introduce security vulnerabilities or unexpected program behavior. |

| **Noncompliant Code** |
| --- |
| Noncompliant Code Example (atoi())  This noncompliant code example converts the string token stored in the buff to a signed integer value using the atoi() function: |
| #include <stdlib.h>    **void** func(**const** **char** \*buff) {  **int** si;    **if** (buff) {      si = **atoi**(buff);    } **else** {      /\* Handle error \*/    }  } |

| **Compliant Code** |
| --- |
| Compliant Solution (strtol())  The strtol(), strtoll(), strtoimax()), strtoul(), strtoull(), strtoumax(), strtof(), strtod(), and strtold() functions convert a null-terminated byte string to long int, long long int, intmax\_t, unsigned long int, unsigned long long int, uintmax\_t, float, double, and long double representation, respectively.  This compliant solution uses strtol() to convert a string token to an integer and ensures that the value is in the range of int: |
| #include <errno.h>  #include <limits.h>  #include <stdlib.h>  #include <stdio.h>    **void** func(**const** **char** \*buff) {  **char** \*end;  **int** si;    **errno** = 0;    **const** **long** sl = **strtol**(buff, &end, 10);    **if** (end == buff) {      (**void**) **fprintf**(stderr, "%s: not a decimal number\n", buff);    } **else** **if** ('\0' != \*end) {      (**void**) **fprintf**(stderr, "%s: extra characters at end of input: %s\n", buff, end);    } **else** **if** ((LONG\_MIN == sl || LONG\_MAX == sl) && ERANGE == **errno**) {      (**void**) **fprintf**(stderr, "%s out of range of type long\n", buff);    } **else** **if** (sl > INT\_MAX) {      (**void**) **fprintf**(stderr, "%ld greater than INT\_MAX\n", sl);    } **else** **if** (sl < INT\_MIN) {      (**void**) **fprintf**(stderr, "%ld less than INT\_MIN\n", sl);    } **else** {      si = (**int**)sl;        /\* Process si \*/    }  } |

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

| **Principles(s):** The **Principle of Least Privilege** asserts that entities (users, processes, etc.) should have only the minimum access rights necessary to perform their tasks. This principle minimizes potential security risks by limiting unnecessary access and exposure of sensitive information. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify SCA | 20.2.0 | Error Handling System | Fortify Static Code Analyzer (SCA) scans source code for vulnerabilities, including improper error handling that could lead to information exposure. |
| Checkmarx | 9.0.0 | Information Leakage | Checkmarx identifies potential information leakage vulnerabilities in error messages by analyzing code paths and data flow within applications. |
| Coverity | 2021.06 | Privacy Violation | Coverity scans for privacy violations, including unintentional exposure of sensitive data in error messages during static code analysis. |
| SonarQube | 9.2 | Sensiotive Data Exposure | [Insert text.] |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-STR51-CPP | Properly null-terminate strings; This standard emphasizes the importance of properly null-terminating strings to avoid buffer overflows and ensure correct string operations. Null-terminated strings are a fundamental aspect of many C++ operations, and failure to properly terminate strings can lead to undefined behavior, data corruption, and security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Noncompliant Code Example  In this noncompliant code example, a std::string object is created from the results of a call to std::getenv(). However, because std::getenv() returns a null pointer on failure, this code can lead to undefined behavior when the environment variable does not exist (or some other error occurs). |
| #include <cstdlib>  #include <string>    **void** f() {    std::string tmp(std::**getenv**("TMP"));  **if** (!tmp.empty()) {      // ...    }  } |

| **Compliant Code** |
| --- |
| Compliant Solution  In this compliant solution, the results from the call to std::getenv() are checked for null before the std::string object is constructed. |
| #include <cstdlib>  #include <string>    **void** f() {  **const** **char** \*tmpPtrVal = std::**getenv**("TMP");    std::string tmp(tmpPtrVal ? tmpPtrVal : "");  **if** (!tmp.empty()) {      // ...    }  } |

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

| **Principles(s):** **Principle of Least Privilege**: This principle emphasizes that every program and user of the system should operate using the least set of privileges necessary to complete the job. It minimizes the number of potential entry points for malicious activity and the impact of accidental errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify SCA | 20.2.0 | Buffer Overflow | Fortify Static Code Analyzer (SCA) identifies potential buffer overflows during string manipulation by analyzing code paths and memory access patterns. |
| Checkmarx | 9.0.0 | Memory Corruption | Checkmarx scans for potential memory corruption issues related to improper string handling, focusing on buffer boundaries and data flow analysis. |
| Coverity | 2021.06 | String Handling | Coverity detects vulnerabilities related to improper string handling that could lead to buffer overflows or memory corruption, offering fixes and recommendations. |
| SonarQube | 9.2 | Buffer Overrun | SonarQube flags code that may result in buffer overflows, especially during string operations, providing insights and remediation steps to secure the code. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | FIO30-C | Exclude user input from format strings; This standard advises excluding user input from format strings to prevent SQL injection and other types of injection attacks. When user input is directly included in format strings, it can introduce vulnerabilities if the input contains format specifiers or special characters. To mitigate this risk, user inputs should be properly sanitized and validated before being used in format strings. This ensures that the input does not alter the intended operation of the code, thereby preventing potential security breaches. |

| **Noncompliant Code** |
| --- |
| Noncompliant Code Example  The incorrect\_password() function in this noncompliant code example is called during identification and authentication to display an error message if the specified user is not found or the password is incorrect. The function accepts the name of the user as a string referenced by user. This is an exemplar of untrusted data that originates from an unauthenticated user. The function constructs an error message that is then output to stderr using the C Standard fprintf() function. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    **void** incorrect\_password(**const** **char** \*user) {  **int** ret;    /\* User names are restricted to 256 or fewer characters \*/  **static** **const** **char** msg\_format[] = "%s cannot be authenticated.\n";  **size\_t** len = **strlen**(user) + **sizeof**(msg\_format);  **char** \*msg = (**char** \*)**malloc**(len);  **if** (msg == NULL) {      /\* Handle error \*/    }    ret = snprintf(msg, len, msg\_format, user);  **if** (ret < 0) {      /\* Handle error \*/    } **else** **if** (ret >= len) {      /\* Handle truncated output \*/    }  **fprintf**(stderr, msg);  **free**(msg);  } |

| **Compliant Code** |
| --- |
| Compliant Solution (fputs())  This compliant solution fixes the problem by replacing the fprintf() call with a call to fputs(), which outputs msg directly to stderr without evaluating its contents: |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    **void** incorrect\_password(**const** **char** \*user) {  **int** ret;    /\* User names are restricted to 256 or fewer characters \*/  **static** **const** **char** msg\_format[] = "%s cannot be authenticated.\n";  **size\_t** len = **strlen**(user) + **sizeof**(msg\_format);  **char** \*msg = (**char** \*)**malloc**(len);  **if** (msg == NULL) {      /\* Handle error \*/    }    ret = snprintf(msg, len, msg\_format, user);  **if** (ret < 0) {      /\* Handle error \*/    } **else** **if** (ret >= len) {      /\* Handle truncated output \*/    }  **fputs**(msg, stderr);  **free**(msg);  } |

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

| **Principles(s):** **Principle of Fail-Safe Defaults**: This principle emphasizes that, by default, access should be denied and only explicitly granted when necessary. It ensures that systems fail securely and do not inadvertently expose vulnerabilities or sensitive data in the event of an error. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify SCA | 20.2.0 | Unchecked Return Value | Fortify Static Code Analyzer (SCA) detects instances where function return values are not checked, highlighting potential areas for error handling improvement. |
| Checkmarx | 9.00 | Return Value Checks | Checkmarx scans for functions with ignored return values, ensuring that error handling is implemented to prevent unexpected behavior and vulnerabilities. |
| Coverity | 2021.06 | Error Handling | Coverity identifies cases where return values of critical functions are not checked, recommending changes to ensure robust error management |
| SonarQube | 9.2 | Function Return Value | SonarQube flags unhandled function return values, providing insights and remediation steps to enforce proper error checking and handling in the code |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM34-C | Only free memory allocated dynamically; This standard emphasizes the importance of only freeing memory that has been dynamically allocated. Freeing memory that was not allocated using dynamic memory allocation functions (such as malloc, calloc, realloc) can lead to undefined behavior, including memory corruption, crashes, and security vulnerabilities. By adhering to this standard, developers can prevent a range of memory-related errors, ensuring the stability and security of their applications. |

| **Noncompliant Code** |
| --- |
| Noncompliant Code Example  This noncompliant code example sets c\_str to reference either dynamically allocated memory or a statically allocated string literal depending on the value of argc. In either case, c\_str is passed as an argument to free(). If anything other than dynamically allocated memory is referenced by c\_str, the call to free(c\_str) is erroneous. |
| #include <stdlib.h>  #include <string.h>  #include <stdio.h>    **enum** { MAX\_ALLOCATION = 1000 };    **int** main(**int** argc, **const** **char** \*argv[]) {  **char** \*c\_str = NULL;  **size\_t** len;    **if** (argc == 2) {      len = **strlen**(argv[1]) + 1;  **if** (len > MAX\_ALLOCATION) {        /\* Handle error \*/      }      c\_str = (**char** \*)**malloc**(len);  **if** (c\_str == NULL) {        /\* Handle error \*/      }  **strcpy**(c\_str, argv[1]);    } **else** {      c\_str = "usage: $>a.exe [string]";  **printf**("%s\n", c\_str);    }  **free**(c\_str);  **return** 0;  } |

| **Compliant Code** |
| --- |
| Compliant Solution  This compliant solution eliminates the possibility of c\_str referencing memory that is not allocated dynamically when passed to free(): |
| #include <stdlib.h>  #include <string.h>  #include <stdio.h>    **enum** { MAX\_ALLOCATION = 1000 };    **int** main(**int** argc, **const** **char** \*argv[]) {  **char** \*c\_str = NULL;  **size\_t** len;    **if** (argc == 2) {      len = **strlen**(argv[1]) + 1;  **if** (len > MAX\_ALLOCATION) {        /\* Handle error \*/      }      c\_str = (**char** \*)**malloc**(len);  **if** (c\_str == NULL) {        /\* Handle error \*/      }  **strcpy**(c\_str, argv[1]);    } **else** {  **printf**("%s\n", "usage: $>a.exe [string]");  **return** EXIT\_FAILURE;    }  **free**(c\_str);  **return** 0;  } |

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

| **Principles(s):** **Principle of Complete Mediation**: This principle emphasizes that every access to a resource should be checked to ensure it is authorized. It ensures that no resource access is overlooked, preventing unauthorized or unintended resource usage. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17.0 | Memcheck | Valgrind's Memcheck tool detects memory management problems, including invalid free, memory leaks, and double-free errors, providing detailed reports for remediation |
| AddressSanitizer | GCC 10.2 | AddressSanitizer | AddressSanitizer (ASan) is a fast memory error detector that identifies memory errors, such as use-after-free, double-free, and memory leaks, aiding in debugging and fixing issues. |
| Coverity | 2021.06 | Dynamic Memory Issues | Coverity scans for dynamic memory issues, including incorrect deallocations and memory leaks, providing suggestions for proper memory management practices. |
| SonarQube | 9.2 | Memory Management | SonarQube checks for common memory management errors in C code, flagging issues like memory leaks and improper deallocations for developer review and correction. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | MSC12-C | Detect and handle standard library errors; This standard emphasizes the importance of detecting and handling errors returned by standard library functions. Using assertions ensures that the program correctly handles unexpected conditions and validates critical assumptions during runtime. By incorporating assertions, developers can enforce error checking systematically, which helps in catching and diagnosing errors early in the development cycle. Adhering to this standard ensures that programs are robust and less prone to crashes or undefined behavior due to unhandled standard library errors. |

| **Noncompliant Code** |
| --- |
| Noncompliant Code Example  This noncompliant code example demonstrates how dead code can be introduced into a program [Fortify 2006]. The second conditional statement, if (s), will never evaluate true because it requires that s not be assigned NULL, and the only path where s can be assigned a non-null value ends with a return statement. |
| **int** func(**int** condition) {  **char** \*s = NULL;  **if** (condition) {          s = (**char** \*)**malloc**(10);  **if** (s == NULL) {             /\* Handle Error \*/          }          /\* Process s \*/  **return** 0;      }      /\* Code that doesn't touch s \*/  **if** (s) {          /\* This code is unreachable \*/      }  **return** 0;  } |

| **Compliant Code** |
| --- |
| Compliant Solution  Remediation of dead code requires the programmer to determine why the code is never executed and then to resolve the situation appropriately. To correct the preceding noncompliant code, the return is removed from the body of the first conditional statement. |
| **nt** func(**int** condition) {  **char** \*s = NULL;  **if** (condition) {          s = (**char** \*)**malloc**(10);  **if** (s == NULL) {             /\* Handle error \*/          }          /\* Process s \*/      }      /\* Code that doesn't touch s \*/  **if** (s) {          /\* This code is now reachable \*/      }  **return** 0;  } |

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

| **Principles(s):** **Principle of Fail-Safe Defaults**: This principle asserts that systems should default to a secure state in the event of a failure. This means that access should be denied by default and only granted when explicit permissions are available. The idea is to minimize potential damage by ensuring that failures or errors do not lead to insecure states or unintended access. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2021.06 | Error Handling Issues | Coverity scans for error handling issues, identifying locations in code where errors are not properly detected or reported, helping developers to enforce secure handling. |
| SonarQube | 9.2 | Error Detection | SonarQube's checker flags issues related to error detection and reporting, ensuring that errors are handled appropriately and securely. |
| CodeSonar | 6.0 | Runtime Error Analysis | CodeSonar detects runtime errors, including improper error handling, providing detailed reports to help developers implement robust error handling mechanisms |
| PVS-Studio | 7.16 | CWE-252 (Unchecked Error) | PVS-Studio checks for unhandled or unchecked errors in the code, ensuring that all errors are detected and handled properly. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR55-CPP | Honor exception specifications; This standard emphasizes the importance of adhering to exception specifications in C++ programs. Exception specifications define which exceptions a function might throw, and adhering to these specifications ensures that exceptions are handled consistently and predictably. By honoring exception specifications, developers can avoid unexpected behavior, ensure proper resource management, and maintain program stability. It also helps in maintaining clear contracts between different parts of the code, which is crucial for large and complex systems. Adhering to this standard leads to more robust and maintainable code by clearly defining and managing error conditions. |

| **Noncompliant Code** |
| --- |
| Noncompliant Code Example  In this noncompliant code example, a function is declared as nonthrowing, but it is possible for std::vector::resize() to throw an exception when the requested memory cannot be allocated. |
| #include <cstddef>  #include <vector>    **void** f(std::vector<**int**> &v, **size\_t** s) noexcept(**true**) {    v.resize(s); // May throw  } |

| **Compliant Code** |
| --- |
| Compliant Solution  In this compliant solution, the function's noexcept-specification is removed, signifying that the function allows all exceptions. |
| #include <cstddef>  #include <vector>    **void** f(std::vector<**int**> &v, **size\_t** s) {    v.resize(s); // May throw, but that is okay  } |

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

| **Principles(s):** **Principle of Complete Mediation**: This principle asserts that every access to a resource should be checked for compliance with a security policy. In the context of coding, this means that exceptions and errors must be handled properly and consistently across all parts of the codebase to ensure that no unauthorized or unintended actions occur as a result of unhandled exceptions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2021.06 | Exception Handling Issues | Coverity checks for issues related to exception handling, ensuring that exceptions are properly specified and handled, helping to enforce compliance with ERR55-CPP |
| SonarQube | 9.2 | Exception Management | SonarQube identifies problems with exception management, ensuring that exceptions are consistently handled according to specifications |
| CodeSonar | 6.0 | Exception Analysis | CodeSonar performs detailed analysis of exception handling, detecting inconsistencies and potential problems in exception specifications and management |
| PVS-Studio | 7.16 | V764(Exception Handling) | PVS-Studio flags issues related to exception handling, such as missing exception specifications or improper handling, ensuring compliance with exception management best practices. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | DCL03-CPP | Use a consistent and conventional naming scheme for types and objects; This standard emphasizes the importance of using consistent and conventional naming schemes for types and objects in C++ programs. Adhering to a consistent naming convention improves code readability and maintainability by making it easier for developers to understand the roles and purposes of different identifiers. It also reduces the likelihood of naming conflicts and helps in navigating large codebases more efficiently. By following this standard, teams can ensure that their codebase is more intuitive and easier to collaborate on, leading to fewer misunderstandings and errors during development. |

| **Noncompliant Code** |
| --- |
| Noncompliant Code Example  This noncompliant code uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly: |
| #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** |
| --- |
| Compliant Solution  For assertions involving only constant expressions, a preprocessor conditional statement may be used, as in this compliant solution: |
| **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(s):** The **Principle of Least Astonishment** maps to **DCL03-CPP** as it emphasizes the importance of declaring objects with appropriate storage durations. This ensures that the object’s lifecycle and visibility align with its intended use, avoiding unexpected behaviors and making the code more predictable and easier to understand. By properly managing object lifetimes, developers can avoid surprises related to resource management and scope, thus adhering to this principle. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2021.06 | Variable Scope Analysis | Coverity checks for issues related to variable scopes and lifetimes, ensuring that objects are declared with appropriate storage durations, supporting compliance with DCL03-CPP. |
| SonarQube | 9.2 | Variable Life Expectancy | SonarQube identifies problems with variable scopes and lifetimes, helping to ensure that objects have the correct storage duration according to best practices |
| CodeSonar | 6.0 | Storage Duration Check | CodeSonar performs detailed analysis of variable storage durations, detecting issues where objects may not have the appropriate lifetime, ensuring adherence to DCL03-CPP |
| PVS-Studio | 7.16 | V801 (Scope and Lifetime) | PVS-Studio flags issues related to variable scopes and lifetimes, such as improper storage duration, ensuring that objects are managed correctly in compliance with DCL03-CPP. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | MEM31-C | Free resources in all cases; This standard emphasizes the importance of ensuring that all allocated resources are properly freed, even in the event of errors or exceptions. In C++, this includes managing dynamically allocated memory, file handles, and other system resources. Failure to free resources can lead to resource leaks, which may degrade system performance, cause crashes, or exhaust system resources. By adhering to this standard, developers can ensure that their programs are robust and reliable, maintaining proper resource management and avoiding potential leaks and undefined behavior. Implementing RAII (Resource Acquisition Is Initialization) and smart pointers is a common practice to adhere to this standard. |

| **Noncompliant Code** |
| --- |
| Noncompliant Code Example  In this noncompliant example, the object allocated by the call to malloc() is not freed before the end of the lifetime of the last pointer text\_buffer referring to the object: |
| #include <stdlib.h>    **enum** { BUFFER\_SIZE = 32 };    **int** f(**void**) {  **char** \*text\_buffer = (**char** \*)**malloc**(BUFFER\_SIZE);  **if** (text\_buffer == NULL) {  **return** -1;    }  **return** 0;  } |

| **Compliant Code** |
| --- |
| Compliant Solution  In this compliant solution, the pointer is deallocated with a call to free(): |
| #include <stdlib.h>    **enum** { BUFFER\_SIZE = 32 };    **int** f(**void**) {  **char** \*text\_buffer = (**char** \*)**malloc**(BUFFER\_SIZE);  **if** (text\_buffer == NULL) {  **return** -1;    }    **free**(text\_buffer);  **return** 0;  } |

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

| **Principles(s):** The Principle of Resource Management directly relates to MEM31-C as this standard focuses on the correct deallocation of dynamically allocated memory. By ensuring that memory is freed when it is no longer needed, developers adhere to good resource management practices, preventing memory leaks that can degrade system performance and stability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17.0 | Memcheck | Valgrind's Memcheck tool helps identify memory leaks and improper memory management, ensuring that dynamically allocated memory is properly freed, supporting MEM31-C. |
| Coverity | 2021.06 | Memory Leak Detection | Coverity detects potential memory leaks by identifying where dynamically allocated memory may not be freed, helping ensure compliance with MEM31-C |
| Clang Static Analyzer | 12.0.1 | Memory Management Checker | Clang Static Analyzer flags issues with dynamic memory allocation and deallocation, helping to identify and fix memory leaks in accordance with MEM31-C |
| PVS-Studio | 7.16 | V773 (Memory Leaks) | PVS-Studio identifies potential memory leaks by checking for dynamically allocated memory that is not properly freed, ensuring adherence to MEM31-C |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | CON50-CPP | Do not destroy mutexes while they are locked; This standard emphasizes the importance of ensuring that mutexes are not destroyed while they are still locked. Mutexes are synchronization primitives used to protect shared data from concurrent access, and destroying a locked mutex can lead to undefined behavior, such as deadlocks, data corruption, or crashes. Adhering to this standard ensures that mutexes are properly managed and only destroyed after they have been unlocked, maintaining the integrity of the synchronization mechanism. This practice contributes to the stability and reliability of concurrent programs, preventing potential concurrency issues and ensuring safe access to shared resources. |

| **Noncompliant Code** |
| --- |
| Noncompliant Code Example  This noncompliant code example creates several threads that each invoke the do\_work() function, passing a unique number as an ID.  Unfortunately, this code contains a race condition, allowing the mutex to be destroyed while it is still owned, because start\_threads() may invoke the mutex's destructor before all of the threads have exited. |
| #include <mutex>  #include <thread>    **const** **size\_t** maxThreads = 10;    **void** do\_work(**size\_t** i, std::mutex \*pm) {    std::lock\_guard<std::mutex> lk(\*pm);      // Access data protected by the lock.  }    **void** start\_threads() {    std::**thread** threads[maxThreads];    std::mutex m;    **for** (**size\_t** i = 0; i < maxThreads; ++i) {      threads[i] = std::**thread**(do\_work, i, &m);    }  } |

| **Compliant Code** |
| --- |
| Compliant Solution  This compliant solution eliminates the race condition by extending the lifetime of the mutex. |
| #include <mutex>  #include <thread>    **const** **size\_t** maxThreads = 10;    **void** do\_work(**size\_t** i, std::mutex \*pm) {    std::lock\_guard<std::mutex> lk(\*pm);      // Access data protected by the lock.  }    std::mutex m;    **void** start\_threads() {    std::**thread** threads[maxThreads];    **for** (**size\_t** i = 0; i < maxThreads; ++i) {      threads[i] = std::**thread**(do\_work, i, &m);    }  } |

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

| **Principles(s):** The Principle of Resource Management directly relates to CON50-CPP as this standard focuses on the correct deallocation of dynamically allocated memory. Ensuring that memory is properly deallocated when it is no longer needed prevents memory leaks and other resource-related issues, maintaining the efficiency and stability of the application. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17.0 | Memcheck | Valgrind's Memcheck tool helps identify memory leaks and improper memory deallocation, ensuring compliance with the CON50-CPP standard by checking for correctly freed memory. |
| Coverity | 2021.06 | Memory leak Detection | Coverity detects potential issues with dynamically allocated memory, including improper deallocation, ensuring that memory is managed according to the CON50-CPP standard. |
| Clang Static Analyzer | 12.0.1 | Memory management Checker | Clang Static Analyzer flags issues related to dynamic memory allocation and deallocation, helping to identify and resolve improper memory management practices per CON50-CPP |
| PVS-Studio | 7.16 | V773 (memory Leaks) | PVS-Studio identifies potential memory leaks by checking if dynamically allocated memory is properly deallocated, supporting adherence to the CON50-CPP standard |
| CppCheck | 2.6 | Memory leak Detection | Cppcheck helps in detecting memory leaks and improper deallocation of memory, providing valuable insights to ensure compliance with the CON50-CPP standard. |

### 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.

***Green Pace integrates automation deeply into their DevSecOps approach, ensuring that security and compliance are foundational at every phase of software development. They deploy tools early in the lifecycle to assess threats, analyze code, and enforce coding standards, thereby identifying vulnerabilities promptly and aligning with regulatory requirements effectively. Automated monitoring and incident response mechanisms in production enable swift detection and mitigation of security threats, maintaining the stability and security of their systems. This comprehensive automation strategy not only strengthens their security posture but also enhances operational efficiency, supporting continuous improvement and resilience across their DevOps workflows.***

***By embedding automation throughout their DevSecOps process, Green Pace not only prioritizes security but also enhances the reliability and efficiency of their software development lifecycle. This proactive approach ensures that security considerations are addressed from design through production, leveraging automated tools for threat detection, compliance checks, and incident response. By automating monitoring and maintenance tasks, Green Pace maintains a secure environment, swiftly identifying and mitigating security risks while optimizing operational processes. This integrated automation strategy not only enhances their ability to meet regulatory requirements and industry standards but also fosters a culture of continuous improvement and innovation within their DevOps operations.***

### 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-ERR34-C | High | Unlikely | Medium | High | 2 |
| STD-STR51-CPP | High | Unlikely | Medium | High | 2 |
| FIO30-C | Medium | Moderate | Low | High | 3 |
| MEM34-C | High | Moderate | Medium | High | 2 |
| MSC12-C | High | Moderate | Medium | High | 2 |
| ERR55-CPP | High | Moderate | Medium | High | 2 |
| DCL03-CPP | High | Moderate | Medium | High | 2 |
| MEM31-C | High | Moderate | Medium | High | 2 |
| CON50-CPP | High | Moderate | 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 | Encryption at rest involves encrypting data stored in databases, files, or storage systems when it is not actively being used. Implemented databases, file systems, and backups to prevent unauthorized access to stored data. This policy is applied to all sensitive data stored within the organization to mitigate risks of data breaches, especially in case of physical theft or unauthorized access to storage systems. It ensures compliance with data protection laws and regulations. |
| Encryption in flight | Encryption in flight secures data while it is being transmitted over networks, protecting it from interception by unauthorized parties. Utilized in protocols like HTTPS, SSL/TLS for web traffic, and VPNs for secure network communication. This policy is essential for safeguarding data integrity and confidentiality during transmission between systems and networks. It prevents eavesdropping and man-in-the-middle attacks, ensuring secure communication channels. |
| Encryption in use | Encryption in use protects data while it is being processed or accessed by applications or users, ensuring confidentiality during active manipulation. Applied in memory encryption, application-level encryption, or cryptographic protection mechanisms. This policy is critical for protecting sensitive information from unauthorized access or exposure during processing. It prevents data leakage or unauthorized access by malicious software or users with elevated privileges. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identity of users or systems attempting to access resources, ensuring only authorized entities gain access. Implemented through passwords, biometrics, tokens, or multi-factor authentication (MFA). This policy applies universally to all systems and applications requiring secure access, preventing unauthorized access attempts and identity theft. It supports compliance with security standards and protects sensitive information from unauthorized access. |
| Authorization | Authorization determines the actions and privileges users or systems are allowed based on their authenticated identity and role. Managed through access control lists (ACLs), role-based access control (RBAC), permissions, and policy-based access controls. This policy is crucial for enforcing least privilege principles, ensuring users only have access to resources necessary for their roles. It prevents unauthorized activities and protects sensitive data from unauthorized modification or deletion. |
| Accounting | Accounting involves monitoring and logging user activities, changes to systems, access to sensitive data, etc., for accountability and forensic purposes. Implemented through logging mechanisms, audit trails, monitoring tools, and security information and event management (SIEM) systems. This policy is essential for detecting security incidents, supporting forensic investigations, and ensuring compliance with regulatory requirements (e.g., GDPR, PCI DSS). It provides transparency and accountability in system operations, enabling organizations to track and analyze security events effectively. |

**\***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

**Principles and Standards Mapping**

1. **Least Privilege**
   * **Standards:** STD-001-CPP, STD-ERR34-C, STD-STR51-CPP, FIO30-C, MEM34-C, MSC12-C, ERR55-CPP, DCL03-CPP, MEM31-C, CON50-CPP
   * **Justification:** Limits access rights to only what's necessary, reducing security risks by minimizing unnecessary privileges across all specified standards.
2. **Defense in Depth**
   * **Standards:** STD-001-CPP, STD-ERR34-C, STD-STR51-CPP, FIO30-C, MEM34-C, MSC12-C, ERR55-CPP, DCL03-CPP, MEM31-C, CON50-CPP
   * **Justification:** Implements multiple layers of security controls across error handling, memory management, file operations, and others to ensure comprehensive protection.
3. **Access Control**
   * **Standards:** STD-001-CPP, STD-ERR34-C, STD-STR51-CPP, FIO30-C, MEM34-C, MSC12-C, ERR55-CPP, DCL03-CPP, MEM31-C, CON50-CPP
   * **Justification:** Regulates access to resources based on policies and permissions, preventing unauthorized access and ensuring proper use of sensitive data.
4. **Security by Design**
   * **Standards:** STD-001-CPP, STD-ERR34-C, STD-STR51-CPP, FIO30-C, MEM34-C, MSC12-C, ERR55-CPP, DCL03-CPP, MEM31-C, CON50-CPP
   * **Justification:** Integrates security measures into systems from the outset, embedding protections such as encryption and secure configurations early in development.
5. **Separation of Duties**
   * **Standards:** STD-001-CPP, STD-ERR34-C, STD-STR51-CPP, FIO30-C, MEM34-C, MSC12-C, ERR55-CPP, DCL03-CPP, MEM31-C, CON50-CPP
   * **Justification:** Assigns distinct responsibilities to different roles, enhancing oversight and preventing conflicts of interest in handling security-related tasks.
6. **Least Common Mechanism**
   * **Standards:** STD-001-CPP, STD-ERR34-C, STD-STR51-CPP, FIO30-C, MEM34-C, MSC12-C, ERR55-CPP, DCL03-CPP, MEM31-C, CON50-CPP
   * **Justification:** Minimizes shared resources and dependencies to limit the impact of security breaches and vulnerabilities across interconnected systems.
7. **Economy of Mechanism**
   * **Standards:** STD-001-CPP, STD-ERR34-C, STD-STR51-CPP, FIO30-C, MEM34-C, MSC12-C, ERR55-CPP, DCL03-CPP, MEM31-C, CON50-CPP
   * **Justification:** Simplifies security practices to improve efficiency and effectiveness, ensuring robust protection while reducing complexity.
8. **Fail-Safe Defaults**
   * **Standards:** STD-001-CPP, STD-ERR34-C, STD-STR51-CPP, FIO30-C, MEM34-C, MSC12-C, ERR55-CPP, DCL03-CPP, MEM31-C, CON50-CPP
   * **Justification:** Establishes secure configurations and defaults to minimize vulnerabilities and unauthorized access attempts in all specified standards.
9. **Complete Mediation**
   * **Standards:** STD-001-CPP, STD-ERR34-C, STD-STR51-CPP, FIO30-C, MEM34-C, MSC12-C, ERR55-CPP, DCL03-CPP, MEM31-C, CON50-CPP
   * **Justification:** Validates and authorizes all accesses to resources, ensuring consistent security enforcement and preventing unauthorized actions.
10. **Least Astonishment**
    * **Standards:** STD-001-CPP, STD-ERR34-C, STD-STR51-CPP, FIO30-C, MEM34-C, MSC12-C, ERR55-CPP, DCL03-CPP, MEM31-C, CON50-CPP
    * **Justification:** Ensures security practices are predictable and transparent, aligning with user expectations and organizational policies across all specified standards.

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 5/24/2024 | Module 3 Milestone | Jorge Torres |  |
| 3.0 | 6/18/2024 | Project One | Jorge Torres |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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