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



# Green Pace Secure Development Policy

## Contents

[Green Pace Secure Development Policy 1](#_Toc798729220)

[Contents 1](#_Toc600587406)

[Overview 2](#_Toc1883714830)

[Purpose 3](#_Toc1660659810)

[Scope 3](#_Toc15385400)

[Module Three Milestone 3](#_Toc1800585532)

[Ten Core Security Principles 3](#_Toc1111943584)

[C/C++ Ten Coding Standards 5](#_Toc1699453269)

[Coding Standard 1 6](#_Toc1391318549)

[Coding Standard 2 8](#_Toc1731020544)

[Coding Standard 3 10](#_Toc1596259120)

[Coding Standard 4 12](#_Toc440087718)

[Coding Standard 5 15](#_Toc1365528889)

[Coding Standard 6 17](#_Toc1763526494)

[Coding Standard 7 19](#_Toc1303424979)

[Coding Standard 8 21](#_Toc338274957)

[Coding Standard 9 23](#_Toc241998594)

[Coding Standard 10 25](#_Toc1925992252)

[Defense-in-Depth Illustration 27](#_Toc418599382)

[Project One 28](#_Toc244788443)

[Revise the C/C++ Standards 28](#_Toc1393899281)

[Risk Assessment 28](#_Toc1461909557)

[Automated Detection 28](#_Toc941874083)

[Automation 28](#_Toc1345693376)

[Summary of Risk Assessments 29](#_Toc1703877424)

[Create Policies for Encryption and Triple A 30](#_Toc374602816)

[Map the Principles 32](#_Toc2080533938)

[Audit Controls and Management 33](#_Toc2068598965)

[Enforcement 34](#_Toc148625136)

[Exceptions Process 34](#_Toc1114254334)

[Distribution 34](#_Toc738317369)

[Policy Change Control 35](#_Toc1117793455)

[Policy Version History 35](#_Toc119453374)

[Appendix A Lookups 35](#_Toc1245647345)

[Approved C/C++ Language Acronyms 35](#_Toc1801547594)

## 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 | Validate Input Data is the process of thoroughly checking data provided by external systems or users to ensure that the data satisfies expected formats and ranges before it is handled by the system. This principle should be applied whenever data is received from an external source like from file uploads, API calls, and user inputs, and it should be applied at the point of data entry into the system whether it's from the user interface level, API endpoint level, or database level. In order to prevent processing errors, system crashes, and security vulnerabilities like SQL injection attacks, various methods, such as format checks, range checks, and presence checks should be implemented to ensure data integrity and system security. |
| 1. Heed Compiler Warnings | Heed Compiler Warnings emphasizes the importance of reviewing warnings from the compiler because they can indicate potential issues that may lead to bugs or unexpected behaviors. Although the program can still run since the warnings are not errors, it is important to carefully inspect and understand compiler warnings during the compilation stage of every program, so that issues like possible loss of data due to type conversions, deprecation warnings, and mismatched types can be addressed and corrected. Other warnings like unused variables and unreachable code should also be attended to as it indicates a piece of code is not functioning as expected. Overall, any programming environment that provides compiler warnings should be inspected to prevent potential runtime errors, improve the reliability of the code, and enhance the overall quality of software. |
| 1. Architect and Design for Security Policies | Architect and Design for Security Policies involves incorporating security considerations throughout the entire lifecycle of a system so that security becomes an integral part of a system rather than an addition that is applied at the end. By emphasizing security and identifying potential security risks during the initial stages of a system to its design, implementation, and maintenance stages, the likelihood of vulnerabilities can be significantly reduced while also making it easier to manage and maintain security over time. This can be done by implementing a combination of security best practices and industry standards throughout architectural design that address how user data is stored, how a system will handle authentication and authorization, and how a system will be updated. Examples of its application can involve selecting secure frameworks and components during the planning phase, conducting threat modeling exercises during the design phase, and regular security audits combined with vulnerability assessments during the maintenance phase. |
| 1. Keep It Simple | Keep it Simple stresses the importance of simplicity in design because complex systems are harder to understand, harder to manage, and more likely to contain errors or vulnerabilities. By keeping systems and security measures simple throughout the entire lifecycle of a policy and in areas like the architecture, code, user interface, and security measures of a system, the overall efficiency and effectiveness of a system can be significantly improved. Overall, it is important to implement clear and understandable security protocols that utilize user-friendly tools and provide comprehensive training to its users so that everyone that interacts with the system is well-equipped to maintain its security. |
| 1. Default Deny | Default Deny refers to blocking all access by default and only allowing actions that are specifically stated based on a defined security policy. This principle is the opposite of Default Allow and considers all actions as potentially harmful, so an action is restricted unless explicitly allowed. This principle hinders usability by being more restrictive, but it helps to minimize the attack surface and reduce the risk of security breaches. It can be applied in network security where a firewall only allows specific types of traffic from trusted sources, system security where access controls grant specific privileges necessary for intended functions, and application security where specific inputs must match a defined pattern in order to be accepted. |
| 1. Adhere to the Principle of Least Privilege | Adhering to the Principle of Least Privilege means providing users with the minimum level of access necessary for them to complete their job functions. By restricting access and preventing users from accessing sensitive information or resources that they do not need for their duties, potential damages from security breaches are minimized, monitoring user activities is simplified, and the risk of unauthorized access by malicious attackers is reduced. Whether it is database management, network security, or system administration, access should be restricted to only what is necessary for a user to perform their specific role or task. For example, a graphic designer does not need the same level of system access as a network administrator, which can protect the system from potential threats if the designer’s account becomes compromised. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing Data Sent to Other Systems is the process of cleaning and preparing data before it is transferred to another system in order to protect the security and integrity of the receiving system. By scrubbing data and ensuring data does not contain malicious code, sensitive information, or any threat that can compromise a system whenever data is being transferred from one system to another, the data and system it interacts with can be protected along with preventing the disruption of the functionality of the system or the utility of the data. It can be performed during routine data backups, data migration, or when sharing data with third-party vendors, and it can be applied at points of data transfer like during file transfers at the database level or when moving data from one application to another. Common examples of data sanitization are data masking, data anonymization, pseudonymization, data deletion, and data scrambling. |
| 1. Practice Defense in Depth | Practicing Defense in Depth involves using a layered approach to security where multiple security measures are implemented throughout various levels of a system so that if one security measure fails, others are in place to prevent the breach or mitigate its damages. This principle should be practiced at all times with any system that requires protection, and it can be applied at all levels of a system, such as physical, network, host, application, and data areas. For example, a company can encrypt sensitive data, require two-factor authentication for system access, implement a firewall to protect its network, and physically lock their server rooms to prevent unauthorized access to sensitive data. |
| 1. Use Effective Quality Assurance Techniques | Using Effective Quality Assurance Techniques centralizes around applying techniques and various practices like integration testing, performance testing, and security testing throughout the software development lifecycle to improve the quality of a system and prevent defects that increase development costs. Quality Assurance techniques should be applied at every stage of the software development process and can include methods like code reviews where peers review code for errors or improvements, unit tests ensure code acts as expected, and system testing verifies an entire system meets the necessary requirements. For security, static and dynamic application security testing, risk assessment, audits, and penetration testing should be used to maintain the integrity, confidentiality, and availability of data within a system along with ensuring that the system is secure and free from any vulnerabilities that can be exploited by malicious attackers. |
| 1. Adopt a Secure Coding Standard | Adopting a Secure Coding Standard is the practice of adhering to a specific set of rules and guidelines during the development of software to guarantee that code is free from vulnerabilities that can lead to security breaches. It should be applied throughout the software development lifecycle from the initial design phases to the final testing and deployment stages, and secure coding standards should be applied in the development environment along with the testing environment. Various companies like Microsoft, Google, and Apple provide guides for developers on secure coding standards that involve practices like implementing proper error handling, validating input, and using secure communication protocols. However, secure coding standards and practices should be used depending on the specific security requirements of a project and the nature of the application that is being developed. |

### 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** | **Always Initialize your Variables** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Each variable has a type that the compiler uses to determine the size and layout of a variable’s memory, and it’s important to initialize variables to ensure they start with a value that is appropriate for their data type. By initializing variables, it prevents the variable from being in an indeterminate state that contains a garbage value and can cause undefined behavior. It also improves code readability and helps avoid logical errors from unexpected or incorrect values. |

| **Noncompliant Code** |
| --- |
| Declaring a variable without initializing it means the variable can hold any value that was previously at its memory location and lead to unpredictable behavior. Especially when a pointer isn’t initialized, a segmentation fault can occur since the pointer can be pointed anywhere. |
| int x;  int \*p;  \*p = 5; // can crash program because p points to a random memory location |

| **Compliant Code** |
| --- |
| Correctly initialize variables to ensure there is a defined value before it is used in any operations. By properly initializing pointers, we can change the value of variables like “x” in a safe and predictable way when there is an assigned value to the memory location that it is pointing to. |
| int x = 10;  int \*p = &x;  \*p = 5; // safe because p points to x |

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

| **Principles(s):**  4. Keep it Simple – Initializing variables is a simple and effective way to reduce complexity and prevent errors by applying predictable and known values to them.  9. Use Effective Quality Assurance Techniques – Having uninitialized variables reduces the quality of code and can cause bugs when variables are used before they are assigned a value.  10. Adopt a Secure Coding Standard – Initializing variables is a common requirement because uninitialized variables can lead to undefined behavior that attackers can exploit. Initializing variables ensures the security of code and prevents variables from pointing to arbitrary memory locations. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | 12 | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -Wuninitialized  clang-analyzer-  core.UndefinedBinaryOperatorResult | Does not catch all instances of this rule, such as uninitialized values read from heap-allocated memory |
| CodeSonar | 8.1p0 | LANG.STRUCT.RPL  LANG.MEM.UVAR | Return pointer to local  Uninitialized variable |
| PVS-Studio | 7.31 | V546, V573, V614, V670, V679, V730, V788, V1007, V1050 |  |
| RuleChecker | 22.10 | uninitialized-read | Partially checked |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Use Constants Instead of Magic Numbers** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | To make code easier to understand and maintain, use constants with descriptive naming conventions to provide context on what numbers represent rather than having magic numbers that appear without any explanation of their meaning or context. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, it is unclear what 100 represents or why 7 is used, which makes it more difficult to understand and maintain the code without context. |
| int array[100];  int nextDay = (currentDay + 1) % 7; |

| **Compliant Code** |
| --- |
| By using named constants like MAX\_SIZE and DAYS\_IN\_WEEK in the example, it becomes clear what the maximum size of the array is and allows other developers to quickly understand what the number represents. |
| const int MAX\_SIZE = 100;  int array[MAX\_SIZE];  const int DAYS\_IN\_WEEK = 7;  int nextDay = (currentDay + 1) % DAYS\_IN\_WEEK; |

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

| **Principles(s):**  4. Keep it Simple – Using constants over magic numbers clarifies the intent of code, and it makes it easier for other developers to read and understand.  9. Use Effective Quality Assurance Techniques – Constants prevent errors that can occur from the misuse of magic numbers, and they only require to be changed in one place when modified.  10. Adopt a Secure Coding Standard – Constants instead of magic numbers is a common standard because it makes code more predictable and less prone to errors, which improves its security. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | 2 | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ECLAIR | 1.2 | CC2.DCL06 | Fully implemented |
| Helix QAC | 2024.1 | C3120,  C3121,  C3122,  C3123,  C3131,  C3132 |  |
| Parasoft C/C++ test | 2023.1 | CERT\_C-DCL06-a | Use meaningful symbolic constants to represent literal values |
| Polyspace Bug Finder | R2024a | CERT C:Rec. DCL06-C | Checks for hard-coded buffer size and hard-coded loop boundary |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Use Range Checking for Element Access** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Ensure that a string or other data structure is not accessed with an out-of-bounds index to prevent runtime errors and potential security vulnerabilities. Use string methods to throw an out\_of\_range exception when an index is not within the range of the string. |

| **Noncompliant Code** |
| --- |
| When calling get\_index(), the value returned by the call can be greater than the number of elements stored in the string, which can cause undefined behavior. Always check that a function does not return a greater than or equal to the size of a string where the index is out of range. |
| #include <string>  extern std::size\_t get\_index();  void function() {  std::string str(“0123”);  str[get\_index()] = ‘2’;  } |

| **Compliant Code** |
| --- |
| Use the “at” function of the std::string class with a try and catch block to check that a position is not greater than or equal to the size of a string. Throw an out of range error to prevent undefined behavior when attempting to modify an out-of-range element. |
| #include <stdexcept>  #include <string>  extern std::size\_t get\_index();  void function() {  std::string str(“0123”);  try {  str.at(get\_index()) = ‘2’;  } catch (std::out\_of\_range &) {  // Handle error  }  } |

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

| **Principles(s):**  1. Validate Input Data – Range checking ensures that the index used to access an element in a string is within the valid range of indices for that string, so that it prevents an out-of-range access that can lead to unpredictable behavior.  4. Keep it Simple – Using simple built-in functions like .at() will make code easier to understand and apply effective range checking that prevents errors when developers manually implement it.  9. Use Effective Quality Assurance Techniques – Range checking prevents bugs that occur when elements outside the valid range of a string are accessed.  10. Adopt a Secure Coding Standard – Range checking is a common requirement because it prevents buffer overflow and other types of errors that can lead to security vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium | 6 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.MEM.BO  LANG.MEM.BU  LANG.MEM.TBA  LANG.MEM.TO  LANG.MEM.TU | Buffer overrun  Buffer underrun  Tainted buffer access  Type overrun  Type underrun |
| Helix QAC | 2024.1 | C++3162,  C++3163,  C++3164,  C++3165 |  |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-STR53-a | Guarantee that container indices are within the valid range |
| Polyspace Bug Finder | R2024a | CERT C++:STR53-CPP | Checks for array access out of bounds, array access with tainted index, and pointer dereference with tainted offset |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Always Use Parameterized Queries or Prepared Statements** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Use parameterized queries or prepared statements to ensure that malicious attackers cannot change the intent of a query or gain unauthorized access that can result in data corruption and data loss. Inputs from users should be treated strictly as data rather than SQL code. |

| **Noncompliant Code** |
| --- |
| Do not allow the user input to be directly concatenated into the SQL command. |
| std::string userInput = "userInput";    sqlite3\* db;  sqlite3\_stmt\* stmt;    int result = sqlite3\_open("test.db", &db);    if (result == SQLITE\_OK) {  std::string sql = "SELECT \* FROM users WHERE username = '" + userInput + "'";  result = sqlite3\_prepare\_v2(db, sql.c\_str(), -1, &stmt, 0);    if (result == SQLITE\_OK) {  while ((result = sqlite3\_step(stmt)) == SQLITE\_ROW) {  // Process the row of data  }  }  sqlite3\_finalize(stmt);  }  sqlite3\_close(db); |

| **Compliant Code** |
| --- |
| Use parameterized queries and the “?” as a placeholder for user input. User input should be treated as a literal string and should not be allowed to be part of the SQL command. |
| std::string userInput = "userInput";    sqlite3\* db;  sqlite3\_stmt\* stmt;    int result = sqlite3\_open("test.db", &db);    if (result == SQLITE\_OK) {  std::string sql = "SELECT \* FROM users WHERE username = ?";  result = sqlite3\_prepare\_v2(db, sql.c\_str(), -1, &stmt, 0);    if (result == SQLITE\_OK) {  // bind the value of the ? placeholder  sqlite3\_bind\_text(stmt, 1, userInput.c\_str(), -1, SQLITE\_STATIC);    while ((result = sqlite3\_step(stmt)) == SQLITE\_ROW) {  // Process the row of data  }  }  sqlite3\_finalize(stmt);  }  sqlite3\_close(db); |

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

| **Principles(s):**  1. Validate Input Data – By using parameterized queries and binding user inputs to an SQL statement, user input is validated to prevent malicious code from attackers.  3. Architect and Design for Security Policies – This code is designed to prevent SQL injection attacks, which is part of most security policies that attempt to prevent cyber-attacks.  7. Sanitize Data Sent to Other Systems - Prepared statements sanitize user inputs and ensure they do not negatively impact SQL statements.  8. Practice Defense in Depth – By using prepared statements, input validation, and access controls, multiple layers of security measures are implemented to protect the integrity of the system.  9. Use Effective Quality Assurance Techniques – Parameterized queries improve the quality of code and ensure that user inputs do not introduce malicious code into the system.  10. Adopt a Secure Coding Standard – Parameterized queries are a common requirement to prevent SQL injection attacks and stop attackers from gaining unauthorized access to information. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | 18 | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 |  | Supported by stubbing/taint analysis |
| CodeSonar | 8.1p0 | IO.INJ.COMMAND  IO.INJ.FMT  IO.INJ.LDAP  IO.INJ.LIB  IO.INJ.SQL  IO.UT.LIB  IO.UT.PROC | Command injection  Format string injection  LDAP injection  Library injection  SQL injection  Untrusted Library Load  Untrusted Process Creation |
| Coverity | 2024.1 | TAINTED\_STRING | Fully implemented |
| Parasoft C/C++ test | 2023.1 | CERT\_C-STR02-a  CERT\_C-STR02-b  CERT\_C-STR02-c | Protect against command inj  Protect against file name inj  Protect against SQL injection |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Do Not Access Freed Memory** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Do not access memory after it has been freed because it can lead to undefined behavior since the memory could have been reallocated and modified. Accessing and modifying freed memory can lead to data corruption or segmentation faults that can cause the program to crash. |

| **Noncompliant Code** |
| --- |
| Code should not attempt to access memory once it has been freed. Attempting to access freed memory will lead to undefined behavior and it is important to not use a pointer after the memory it points to has been freed. |
| int\* p = new int(5);  delete p; // memory is freed  \*p = 10; // accessing freed memory |

| **Compliant Code** |
| --- |
| After memory has been freed, it should not be accessed. Pointers should be set to nullptr to prevent dangling pointers and malicious code that can exploit freed memory to execute harmful operations or access sensitive information. |
| int\* p = new int(5);  delete p; // memory is freed  // no further access to p  p = nullptr; // pointer is set to nullptr |

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

| **Principles(s):**  2. Heed Compiler Warnings – Warnings will highlight potential memory leaks or attempts to access freed memory, which can lead to undefined behavior.  3. Keep it Simple – By simply not accessing freed memory and preventing situations like dangling pointers, fear of data corruption and segmentation faults can be avoided.  6. Adhere to the Principle of Least Privilege – Code should only be able to access the resources it needs to perform its task, and once memory is freed, code should not have the privilege to access it.  8. Practice Defense in Depth – There should be various checks that ensure memory is properly deleted and that code cannot access freed memory.  9. Use Effective Quality Assurance Techniques – Checking that code is not accessing freed memory through code reviews and testing will catch instances where freed memory is being accessed and improve the quality of code.  10. Adopt a Secure Coding Standard – Memory should be managed correctly to prevent undefined behavior and potential security vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | 18 | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | ALLOC.UAF | Use after free |
| Coverity | v7.5.0 | USE\_AFTER\_FREE | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| LDRA tool suite | 9.7.1 | 483 S, 484 S | Partially implemented |
| Polyspace Bug Finder | R2024a | CERT C++:MEM50-CPP | Checks for pointer access out of bounds, deallocation of previously deallocated pointer, use of previously freed pointer |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use Assertions for Programming Errors Not Runtime Errors** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions should check for conditions that should never occur in the code, and they are a valuable tool for catching bugs and logical errors. They are designed for debugging and development purposes, and they should not be used for error handling in a production environment. |

| **Noncompliant Code** |
| --- |
| Assertions should not check for conditions that can occur during normal operations of a program. Rather than use assertions to handle runtime errors, the code should check for the condition and handle it appropriately. |
| include <cassert>  void divide(int numerator, int denominator) {  assert(denominator != 0); // Assert that denominator is not zero  if (denominator == 0) {  // Handle division by zero  } else {  int result = numerator / denominator; // Rest of the function  }  } |

| **Compliant Code** |
| --- |
| In the compliant code, the assertion is used to ensure that the denominator is not zero before performing a division operation. If the denominator is zero, the assertion will fail and terminate the program, which will indicate to the programmer that there is a bug in the code that needs to be fixed. |
| include <cassert>    void divide(int numerator, int denominator) {  assert(denominator != 0); // Assert that denominator is not zero  int result = numerator / denominator; // Rest of the function  } |

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

| **Principles(s):**  1. Validate Input Data – Assertions should be used to check the validity of inputs to simplify the debugging process and prevent the propagation of errors through a system.  2. Heed Compiler Warnings – Failed assertions can act as a compiler warning and catch programming errors during code development.  4. Keep it Simple – Assertions are a simple and straightforward way to check for programming errors while also making the code more readable and maintainable.  9. Use Effective Quality Assurance Techniques – Using assertions early in the development phase will detect and fix bugs early, which will improve the overall quality of the code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL03 |  |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| CodeSonar | 8.1p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Catch and Handle All Exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | All exceptions should be caught and handled to ensure program stability and maintainability. Unhandled exceptions can cause a program to terminate unexpectedly, which can lead to potential data loss and poor user experience. |

| **Noncompliant Code** |
| --- |
| Because there is no code to catch and handle the exception when functionThatThrows throws a runtime error, the program will terminate unexpectedly once the exception is thrown. |
| void functionThatThrows() {  throw std::runtime\_error("An error occurred");  }    int main() {  functionThatThrows();  return 0;  } |

| **Compliant Code** |
| --- |
| It is important to catch and handle all exceptions so that we can control the program’s behavior and provide meaningful feedback to the user when errors occur. This prevents the program from terminating unexpectedly and allows an error message to be printed or logged for identifying and resolving the issue. |
| void functionThatThrows() {  throw std::runtime\_error("An error occurred");  }    int main() {  try {  functionThatThrows();  }  catch (const std::exception& e) {  std::cerr << "Caught exception: " << e.what() << std::endl;  // Handle the exception appropriately here  }  return 0;  } |

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

| **Principles(s):**  1. Validate Input Data – Checking input data for any exceptions before processing it ensures input data does not cause any unexpected or incorrect behaviors.  4. Keep it Simple – Catching and handling exceptions are a simple and understandable method for identifying errors and avoiding complex error handling mechanisms.  5. Default Deny – All exceptions should be caught and unhandled unless explicitly allowed to propagate.  8. Practice Defense in Depth – Catching and handling exceptions ensure multiple layers of security controls are implemented that prevent unhandled exceptions from causing issues.  9. Use Effective Quality Assurance Techniques – By catching and handling exceptions, the code can handle unexpected situations gracefully and allow developers to identify issues before they become a larger issue in production. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Medium | 4 | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |
| Polyspace Bug Finder | R2024a | CERT C++:ERR51-CPP | Checks for unhandled exceptions |
| RuleChecker | 22.10 | main-function-catch-all-early-catch-all | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Do Not Modify Standard Namespaces** |
| --- | --- | --- |
| Data Type/Declarations | [STD-008-CPP] | Modifying the standard namespace can cause undefined behavior, name clashes, portability issues, and make code harder to read and maintain. Define your own namespaces and leave the std namespace as it is so that standard library functionalities are not altered. |

| **Noncompliant Code** |
| --- |
| The std namespace is reserved for the C++ standard library and declarations or definitions should not be added to it to prevent name conflicts and undefined behavior. |
| namespace std {  void print\_hello() {  cout << "Hello, World!" << endl;  }  }    int main() {  std::print\_hello();  return 0;  } |

| **Compliant Code** |
| --- |
| Defining your own namespace and adding functions to it is a safe and compliant way to add functionality without risking undefined behaviors or conflicts with the std namespace. |
| namespace my\_namespace {  void print\_hello() {  std::cout << "Hello, World!" << std::endl;  }  }    int main() {  my\_namespace::print\_hello();  return 0;  } |

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

| **Principles(s):**  4. Keep it Simple – Not modifying the standard namespace will make it easier for other developers to read code while also preventing undefined behavior and name conflicts.  6. Adhere to the Principle of Least Privilege – Not modifying the standard namespace ensures that code does not have unnecessary access to the standard namespace.  10. Adopt a Secure Coding Standard – By not modifying the standard namespace, the code is more secure and prevents bugs since there won’t be any conflicts with standard library names. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium | 6 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.STRUCT.DECL.SNM | Modification of Standard Namespaces |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-DCL58-a | Do not modify the standard namespaces ‘std’ and ‘posix’ |
| Polyspace Bug Finder | R2024a | CERT C++:DCL58-CPP | Checks for modification of standard namespaces |
| PVS-Studio | 7.31 | V1061 |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Prevent Integer Overflow and Underflow** |
| --- | --- | --- |
| Data Value/Integers | [STD-009-CPP] | Because unsigned integers have a fixed size, operations should not cause the value of an unsigned integer to go beyond its maximum limit. Use appropriate checks and bounds with the correct data types to prevent malicious attackers from exploiting integer wrapping and executing arbitrary code that can lead to unexpected results. |

| **Noncompliant Code** |
| --- |
| This noncompliant code does not check for integer wrap and can result in unexpected behaviors where insufficient memory can lead to an exploitable vulnerability. |
| void func(unsigned int num\_a, unsigned int num\_b) {  unsigned int sum = num\_a + num\_b;  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| Perform a precondition test of the operands to ensure that there is no possibility of unsigned wrap. |
| void func(unsigned int num\_a, unsigned int num\_b) {  unsigned int sum;  if (UINT\_MAX - num\_a < num\_b) {  /\* Handle error \*/  }  else {  sum = num\_a + num\_b;  }  /\* ... \*/  } |

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

| **Principles(s):**  1. Validate Input Data – Preventing integer overflow and underflow by ensuring that data in calculations are valid will prevent integer wrapping and vulnerabilities that attackers can exploit.  4. Keep it Simple – Using checks before calculations prevents potential overflow and underflow while also making code easier to debug and maintain.  5. Default Deny – Checks will deny all actions that can cause an overflow and will only allow actions that calculate within the appropriate bounds.  6. Adhere to the Principle of Least Privilege – Code segments should not exceed its privilege of performing safe mathematical operations.  9. Use Effective Quality Assurance Techniques – Preventing integer overflow and underflow ensures calculations remains within their range and improves the overall quality of code.  10. Adopt a Secure Coding Standard – Preventing integer overflow and underflow is a common requirement for security standards because it prevents bugs and makes the code more secure. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | 9 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | Integer-overflow | Fully checked |
| CodeSonar | 8.1p0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  MISC.MEM.SIZE.ADDOFLOW  MISC.MEM.SIZE.BAD  MISC.MEM.SIZE.MULOFLOW  MISC.MEM.SIZE.SUBUFLOW | Addition overflow of allocation size  Integer overflow of allocation size  Multiplication overflow of “ “  Subtraction underflow of “ “  Addition overflow of size  Unreasonable size argument  Multiplication overflow of size  Subtraction underflow of size |
| Coverity | 2017.07 | TAINTED\_SCALAR  BAD\_SHIFT | Implemented |
| TrustInSoft Analyzer | 1.38 | signed\_overflow | Exhaustively verified |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Guarantee Storage for Strings has Sufficient Space** |
| --- | --- | --- |
| String Correctness | [STD-010-CPP] | Ensure that storage for strings is large enough to hold character data and the null terminator so that errors like buffer overflows do not occur. |

| **Noncompliant Code** |
| --- |
| Since the input is unbounded, a buffer overflow can occur when a user inputs more data than can be stored. |
| void func() {  char buffer[12];  std::cin >> buffer;  } |

| **Compliant Code** |
| --- |
| Use std::string instead of a bounded array to guard against buffer overflows and ensure that data is not truncated. |
| void func() {  std::string input;  std::string str1, str2;  std::cin >> str1 >> str2;  } |

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

| **Principles(s):**  1. Validate Input Data – Checking that there is sufficient storage for strings will ensure unexpected behaviors do not occur.  4. Keep it Simple – Utilizing the string class will automatically allocate enough memory to hold a value, and it will deallocate the memory once the string object is no longer needed.  6. Adhere to the Principle of Least Privilege – Functions should not exceed their privilege of storing data by guaranteeing there is sufficient storage for strings.  9. Use Effective Quality Assurance Techniques – Guaranteeing sufficient storage for strings ensures that code can handle different sizes of input data gracefully.  10. Adopt a Secure Coding Standard – Guaranteeing sufficient storage for strings is a common standard requirement because it prevents buffer overflows and data from being truncated, which can cause bugs and unexpected behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | 18 | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR31 | Detects calls to unsafe string function that may cause buffer overflow  Detects potential buffer overruns, including those caused by unsafe usage of fscanf() |
| CodeSonar | 8.1p0 | LANG.MEM.BO  LANG.MEM.TO  MISC.MEM.NTERM  BADFUNC.BO.\* | Buffer overrun  Type overrun  No space for null terminator  A collection of warning classes that report uses of library functions prone to internal buffer overflows |
| Coverity | 2017.07 | STRING\_OVERFLOW  BUFFER\_SIZE  OVERRUN  STRING\_SIZE | Fully implemented |
| PC-lint Plus | 1.4 | 421, 498 | Partially supported |

### 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.

After looking at the diagram, we can modify the existing DevOps process to automate enforcement of the standards in this policy at various phases with the depth in defense depending on the budget and time constraints of project. Beginning from the Assess and plan phase, it’s important to use automation tools like Microsoft’s Threat Modeling Tool and OWASP Threat Dragon to evaluate the current state of the system and provide risk analysis for potential security threats. They will play an important role in obtaining a comprehensive understanding of the vulnerabilities in our system, and they will strongly influence how we plan and implement effective mitigation strategies in our design phase. In the design phase, policy as code frameworks like Open Policy Agent (OPA) can be used to automatically enforce design policies related to access control, authorization, and other service-level policies. For the build phase, automated tools like Nexus Repository OSS can be used to manage trusted repositories and contribute to a secure build environment. CPPCheck and Coverity can then be applied as static application security testing (SAST) tools to scan the codebase for security flaws. Once the Verify and Test Phase begins, dynamic application security testing (DAST) tools like OWASP ZAP and Burp Suite can identify runtime vulnerabilities in the code.

Once the Production phase starts, penetration testing can be performed, and configuration management tools like Chef can be used to automate the deployment of our C++ application with secure configurations. For the monitor and detect phase, intrusion detection systems (IDS) and Security Information and Event Management (SIEM) tools can monitor and analyze logs for unusual activities that could indicate a security breach. Automated response tools like IBM Resilient or any other Security Orchestration, Automation, and Response (SOAR) tools can be used to help respond to incidents and minimize their potential damages. Finally, in the maintain and stabilize phase, backup and recovery tools like Rubrik along with change management and monitoring tools like GitLab should be used to keep the production environment stable and secure.

### Summary of Risk Assessments

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

**\*\*5 being the greatest threat level, and 1 being the lowest threat level\*\***

**\*Credit to Aaron Ballman for SEI CERT C++ Coding Standard**

**Ballman, A. (n.d.). SEI CERT C++ Coding Standard. Software Engineering Institute.** [SEI CERT C++ Coding Standard](https://wiki.sei.cmu.edu/confluence/display/cplusplus/2+Rules)

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Probable | Medium | 12 (High) | 5 |
| STD-002-CPP | Low | Unlikely | Medium | 2 (Low) | 1 |
| STD-003-CPP | High | Unlikely | Medium | 6 (Med) | 3 |
| STD-004-CPP | High | Likely | Medium | 18 (High) | 5 |
| STD-005-CPP | High | Likely | Medium | 18 (High) | 5 |
| STD-006-CPP | Low | Unlikely | High | 1 (Low) | 1 |
| STD-007-CPP | Low | Probable | Medium | 4 (Low) | 1 |
| STD-008-CPP | High | Unlikely | Medium | 6 (Med) | 3 |
| STD-009-CPP | High | Likely | High | 9 (Med) | 3 |
| STD-010-CPP | High | Likely | Medium | 18 (High) | 5 |

### 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 process of encrypting data that is not being accessed or transferred into an unreadable format unless decrypted with a decryption key. The data is stored on a physical or virtual disk, and the data is encoded by using encryption algorithms that are generally categorized as either symmetric or asymmetric. In practice, symmetric encryption like Advanced Encryption Standard (AES) and Triple Data Encryption Standard (3DES) use the same key for both encryption and decryption, where the sender and receiver of the data both have access to the same secret key. This method is fastest and most efficient, which makes it suitable for encrypting large amounts of data. Asymmetric encryption like Rivest-Shamir-Adleman (RSA) and Elliptic Curve Cryptography, on the other hand, is more secure as it has a public key that is used for encryption and can be known to everyone while also having a separate private key that can only be used by an individual that is decrypting the data. This structure is more computationally intensive, but it is best for secure key exchange and digital signatures. Overall, encryption at rest should be used to protect confidential information in case a storage medium is lost, stolen, or accessed by unauthorized individuals. It also makes it more difficult for attackers to access readable information, adheres to compliance requirements involving personal information, and builds trust between a company and its customers. |
| Encryption in flight | Encryption in flight, or encryption in transit, involves the process of encrypting data while it is being transferred from one location to another. Whether the data is transported across a network or between systems, encryption in flight’s purpose is to protect sensitive information from being intercepted or captured by a third party during the data’s transmission. It can be applied in practice through methods like Hypertext Transfer Protocol Secure (HTTPS), Secure Sockets Layer/Transport Layer Security (SSL/TLS), and a Virtual Private Network (VPN). Overall, these methods and secure file transfer protocols allow organizations to enhance their security when transporting data from one place to another. |
| Encryption in use | Encryption in use, or runtime encryption, consists of encrypting data that is currently being accessed, read, updated, and processed in a computer’s memory. Because this is when data is immediately available and decrypted so the system can understand it, it is when data is most vulnerable. In practice, data should only be decrypted when necessary and should be re-encrypted as soon as possible to reduce the amount of time that the data is exposed. Secure computing environments like Trusted Execution Environments (TEEs) can be used to decrypt and process data securely as they reduce the risk of exposure by being isolated from the rest of a system. Another approach, especially for cloud computing environments where data might be processed on shared or potentially insecure systems, use of Secure Encrypted Virtualization (SEV) should be used to isolate the memory of each virtual machine from others running on the same physical server. Overall, these are common methods for achieving encryption in use, but a comprehensive approach that includes regular auditing, robust key management, and other industry standards are necessary to maintain the data’s security and protect it from malicious attackers. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the first step of the Triple-A framework, and it is the process of verifying the identity of a user, device, or system so that it can gain access to a network. The policy applies because authentication is the first line of defense against unauthorized access and assists with ensuring that the entity requesting access is who or what it claims to be. With regards to user logins, credentials like login information, biometric data, or digital certificates are verified against a database to ensure that only authorized users or systems can access the network. By establishing the identity of the entity requesting access, the system can enforce access controls and track activity accurately. Especially if a user needs to make changes to the database like update their user credentials, authentication must be performed to ensure that the user making changes is the rightful owner of the account. Just like with the addition of new users, unique credentials must be assigned to them and authenticated when they log into the system, so that the system knows what they are authorized to perform. |
| Authorization | After a user has been authenticated, authorization follows, which involves the process of denying or granting the user access to specific resources or services. Ultimately, the process determines what the user, device, or system has the permission to do, and it is usually based on access control lists, role-based access controls, or other policy-based security mechanisms. Depending on the user’s role and responsibilities, authorization decides the user level of access an account may have to a system’s resources and what changes they are able to make to a database. Other methods of how authorization is applied is through Discretionary Access Control (DAC), Attribute-Based Access Control (ABAC), and Mandatory Access Control (MAC). Overall, the purpose of authorization is to enforce that each account can only do what they are supposed to do and nothing more, which emphasizes effective security management and aligns with the principle of least privilege. |
| Accounting | Once users are authenticated and authorized, the process of accounting occurs, where the user is tracked and recorded based on their activities within a system. Data is collected about user activities like when they logged on, what files were accessed by users, the number of resources they’ve used, and what changes they may have made. After being collected, the data is securely stored and analyzed to identify patterns, detect potential security threats, and ensure compliance with policies. Reports can then be generated for purposes like auditing, planning, and billing. In the end, proper accounting improves security by detecting unusual behavior, resource management by tracking consumption, and overall planning by providing valuable insight on system details. When combined together with authentication and authorization, they form a comprehensive security framework that utilizes defense in depth best practices and makes it easier to maintain the integrity and security of a system. |

**\***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
* Standard: Operating System Logs
* 4. Keep it Simple – Clear and concise log entries make it easier to analyze log entries and identify potential threats.
* 6. Adhere to the Principle of Least Privilege – Accessing and modifying logs should have restricted access to prevent unauthorized access and potential security breaches.
* 8. Practice Defense in Depth – Being able to monitor and analyze logs adds an additional layer of defense towards detecting intrusion and facilitating an incident response.
* 9. Use Effective Quality Assurance Techniques – Regularly checking and testing the logging system can help identify and fix any issues or vulnerabilities that will improve the overall quality of the code.
* Standard: Firewall Logs
* 4. Keep it Simple – Firewall Logs make it straightforward and easy to identify and respond to potential threats.
* 5. Default Deny – Firewalls are capable of blocking all traffic by default and only allowing traffic that is explicitly permitted.
* 6. Adhere to the Principle of Least Privilege – Accessing and modifying logs should have restricted access to prevent unauthorized access and potential security breaches.
* Standard: Anti-malware logs
* 4. Keep it Simple – Anti-malware Logs simplify the process of identifying and responding to potential threats.
* 6. Adhere to the Principle of Least Privilege – Accessing and modifying logs should have restricted access to prevent unauthorized access and potential security breaches.
* 8. Practice Defense in Depth – Being able to monitor and analyze logs adds an additional layer of defense towards detecting intrusion and facilitating an incident response.
* 9. Use Effective Quality Assurance Techniques – Regularly checking and testing the logging system can help identify and fix any issues or vulnerabilities that will improve the overall quality of the code.

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 | 05/26/2024 | M3 Milestone: 10 Principles & initial Coding Standards | Michael Gagujas |  |
| 3.0 | 06/16/2024 | Security Policy completion for Project One of SNHU CS405. Added Risk Assessment, Automated Tools, Automation explanation, and Summary of Risk Assessments. | Michael Gagujas |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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