**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 | [Input validation of user or external system data ensures that malicious attacks like SQL injection or cross-site scripting (XSS) shall not be possible by rejecting or sanitizing the unexpected or harmful data.] |
| 1. Heed Compiler Warnings | [Following compiler warnings may reveal some of the possible vulnerabilities and bugs way in advance during a development lifecycle. If they were ignored, most likely this software would keep being attacked.] |
| 1. Architect and Design for Security Policies | [Embedding the aspects of security policies and measures in the very design phases of software development secures a system. In that way, proactively protecting from various attack possibilities minimizes the occurrences of vulnerabilities while securing an entire system.] |
| 1. Keep It Simple | [Simplicity in code design and implementation reduces complexity, thus making it easier to find and fix security flaws. Complex code can mask vulnerabilities and be more difficult to maintain and secure.] |
| 1. Default Deny | [The default-deny approach means that access is denied by default and granted only when explicitly allowed. This principle minimizes unauthorized access and ensures that only intended actions are permitted.] |
| 1. Adhere to the Principle of Least Privilege | [Giving users and processes only those privileges that are strictly necessary for their functions limits the amount of damage that can be caused by either accidents or exploits. This principle helps in breach containment and reduces the attack surface.] |
| 1. Sanitize Data Sent to Other Systems | [It aids in sending data to other systems by sanitizing the data so that no malicious code or harmful data propagates the software, causing security breaches in interconnected systems. |
| 1. Practice Defense in Depth | [Having different levels of multiple layers of security forms a full-blown defense mechanism. If one layer is compromised, there are still other layers offering protection, making the effort of attacking the whole system much more complicated.] |
| 1. Use Effective Quality Assurance Techniques | [Rigorous testing and code review practices facilitate the early identification and mitigation of vulnerabilities. Performing effective quality assurance ensures that security flaws are detected and fixed before deployment.] |
| 1. Adopt a Secure Coding Standard | [Applying established secure coding practices, such as the SEI CERT C++ Coding Standard, helps guide developers in secure coding. By following these standards, common vulnerabilities can be avoided and code security can be improved.] |

### 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** | **Appropriate Data Types** |
| --- | --- | --- |
| **Data Type** | [STD-001-DAT] | [Avoid unexpected behavior: One gets the expected results if the correct data type is used. On the other hand, incorrect data types could result in unexpected behavior, including truncation, overflow, and bad calculations.  Performance improvement: The correct data type can optimize the usage of memory and enhance the performance of a program. For instance, using an integer for whole numbers instead of a floating-point number can reduce memory consumption.  Improving code readability: Clearly and descriptively named data types make code easier to read and, thus, maintain.] |

| **Noncompliant Code** |
| --- |
| [This code tries to store a big integer value into a short integer variable; this may cause overflow and show unexpected results.] |
| [ short int small\_value = 32768; // Maximum value for short int is 32767] |

| **Compliant Code** |
| --- |
| [This code uses an integer data type which is appropriate to hold the given value without the risk of overflow.] |
| [ int large\_value = 32768;] |
| This code uses a long long data type to store a very large integer value, ensuring sufficient space to prevent overflow.] |
| long long very\_large\_value = 9223372036854775807; // Maximum value for  long long |

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

| **Principles(s):** [Principle 1: Secure Coding Principles Ensures that the right data types are used to prevent vulnerabilities such as overflow, which would be taken advantage of by attackers.  Principle 2: Performance Optimization Using appropriate data types improves memory management and overall program performance.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| [High - Incorrect data types can lead to critical issues like overflow, resulting in serious vulnerabilities.] | [Likely - The chances of encountering data type issues are relatively high if best practices are not followed.] | [Medium - Fixing data type issues can be moderately costly in terms of time and resources, but it’s not as expensive as other critical bugs.] | [High - Ensuring correct data types is a priority to prevent potential exploits and maintain software integrity.] | [4 - Represents the overall risk level based on the combination of severity, likelihood, and remediation cost.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.0 | S2147 | SonarQube provides static analysis for ensuring the use of appropriate data types to prevent issues like overflow. |
| Coverity | 2022.12 | OVRFLW | Coverity identifies instances where inappropriate data types might cause overflow or truncation, ensuring compliance with data type standards. |
| Fortify | 19.2 | CWE-190 | Fortify scans code for potential integer overflow issues and helps in mitigating risks associated with improper data types. |
| PVS-Studio | 7.16 | V108 | PVS-Studio detects cases where incorrect data types might lead to unexpected behavior, optimizing code safety and performance. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Validating and Sanitizing of Data Values** |
| --- | --- | --- |
| **Data Value** | [STD-002-DAT] | [It helps avoid unexpected behavior like crashing, miscalculation, and even security vulnerabilities since it is very important that the data values be within a range and type and should not include some unexpected or malicious content.  Data Integrity: It ensures that the data integrity is preserved through validation and sanitization, which block invalid or corrupted data entries.  Security Risk Mitigation: Proper data sanitizing prevents a large variety of security vulnerabilities, ranging from SQL code injection to Cross-site scripting or even buffer overflows.] |

| **Noncompliant Code** |
| --- |
| [This code uses user input directly, without any validation or sanitization. If there are malicious characters in the input, this may constitute a security vulnerability.] |
| [ std::string user\_input;  std::cin >> user\_input;  // Directly use user\_input without validation  std::cout << "You entered: " << user\_input << std::endl;] |

| **Compliant Code** |
| --- |
| [This code checks whether the user input is empty before using it, which can avoid some problems related to empty strings.] |
| [ std::string user\_input;  std::cin >> user\_input;  if (!user\_input.empty()) {  std::cout << "You entered: " << user\_input << std::endl;  } else {  std::cout << "No input entered." << std::endl;  }] |
| This code sanitizes the user's input to remove harmful characters before its usage. This includes HTML tags that might be used in XSS attacks. |
| #include <boost/regex.hpp>  std::string user\_input;  std::cin >> user\_input;  boost::regex html\_regex("<[^>]\*>");  std::string sanitized\_input = boost::regex\_replace(user\_input,  html\_regex, "");  std::cout << "Sanitized input: " << sanitized\_input << std::endl; |

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

| **Principles(s):** [Principle 1: Safe Input Handling Ensure all input is validated and sanitized so that malicious data does not affect the system negatively.  Principle 2: Data Integrity By input validation and sanitizing the input data, data integrity is assured in such a way that only valid and safe data are processed.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| [High - Unvalidated and unsanitized data can lead to critical vulnerabilities like SQL injection and XSS.] | [Very Likely - User inputs are common causes of vulnerabilities if not handled in a proper manner.] | [Medium - Validating and sanitizing properly requires moderate effort.] | [High - Data value validation and sanitization are necessary to prevent security violations.] | [5 - These have the highest risk due to the overlap of high severity and high likelihood.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.0 | S2076 | SonarQube checks for potential SQL injection vulnerabilities by identifying unvalidated user inputs. |
| Coverity | 2022.12 | SANITIZE | Coverity identifies places where input validation and sanitization are missing, helping to prevent injection attacks. |
| Fortify | 19.2 | CWE-79 | Fortify scans for potential cross-site scripting (XSS) vulnerabilities by checking for unsanitized inputs. |
| Veracode | 2023.1 | Input Validation | Veracode detects instances where input validation and sanitization are missing or inadequate. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Handle String Operations Safely** |
| --- | --- | --- |
| **String Correctness** | [STD-003-STR] | [Buffer overflows, besides other issues pertaining to incorrect string handling, can also be very disastrous in terms of security; they can give rise to memory corruption, thereby allowing an attack to take advantage.  Ensuring Data Integrity: Safe strings ensure the integrity of the data in a string as well as any other type.  Improves Reliability of the Code: Ensures that fewer unexpected bugs take place in the code.] |

| **Noncompliant Code** |
| --- |
| [This code uses strcpy() function which has a vulnerability to buffer overflow if the destination buffer is not big enough to contain the source string.] |
| [ char destination[10];  char source[] = "This is a long string.";  strcpy(destination, source); // Potential buffer overflow] |

| **Compliant Code** |
| --- |
| [This code uses strncpy() function, it limits the number of characters copied based on the destination buffer size so no buffer overflow would happen.] |
| [ char destination[10];  char source[] = "This is a long string.";  strncpy(destination, source, sizeof(destination) - 1);  destination[sizeof(destination) - 1] = '\0'; // Ensure null termination] |
| The code utilizes the class std::string and other facilities of the C++ Standard Library. It allows safer, more comfortable string processing with automatic memory management and bounds checking. |
| std::string destination = "This is a safe string.";  std::string source = "Another string";  destination = source; // Safe assignment |

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

| **Principles(s):** [Principle 1: Secure Memory Management Ensures strings are handled securely to prevent buffer overflows and memory corruption.  Principle 2: Data Integrity With secure string handling, data integrity is ensured so the data is valid and secure.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| [High - Buffer overflows can lead to serious security vulnerabilities and memory corruption.] | [Likely - Incorrect string handling is a common phenomenon if best practices are not used.] | [Medium - Implementing safe string handling requires moderate effort.] | [High - String operations must be handled securely to prevent security breaches.] | [4 - Refers to the overall risk level based on the severity, probability, and cost of fixing.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.0 | S2259 | SonarQube checks for potential buffer overflows and ensures safe string handling practices. |
| Coverity | 2022.12 | STRLCPY | Coverity identifies places where unsafe string handling functions are used and suggests safe alternatives. |
| Fortify | 19.2 | CWE-120 | Fortify scans for potential buffer overflow vulnerabilities caused by unsafe string handling. |
| PVS-Studio | 7.16 | V3012 | PVS-Studio detects unsafe string operations and provides recommendations for safe handling. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Avoid SQL Injection Vulnerabilities** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-SQL] | [Data Breaches: SQL injection allows an attacker to manipulate SQL queries by injecting malicious code into the database, hence breaching data, accessing unauthorized information, or compromising the system.  Loss of Sensitive Information: An attacker can disclose sensitive data such as user credentials, financial information, or intellectual property using SQL injection.  System Disruption: SQL injection is used in causing service disruptions through disturbing database operations, which results in data corruption.] |

| **Noncompliant Code** |
| --- |
| [This code directly concatenates user input into an SQL query and is thus vulnerable to SQL injection attacks.] |
| [ std::string username = "user1";  std::string password = "password";  std::string query = "SELECT \* FROM users WHERE username='" + username  + "' AND password='" + password + "'";] |

| **Compliant Code** |
| --- |
| [This code uses a parameterized query. A parameterized query is one whose user input and SQL query structure are kept separate, and under no circumstances can an attacker inject malignant code.] |
| [ #include <mysql/mysql.h>  MYSQL\* conn;  MYSQL\_RES\* res;  MYSQL\_ROW row;  // ... (Connect to database) ...  std::string username = "user1";  std::string password = "password";  std::string query = "SELECT \* FROM users WHERE username=? AND  password=?";  MYSQL\_STMT\* stmt = mysql\_stmt\_init(conn);  mysql\_stmt\_prepare(stmt, query.c\_str(), query.length());  // Bind parameters  mysql\_stmt\_bind\_param(stmt, "ss", &username[0], &username.length(),  &password[0], &password.length());  // Execute query  mysql\_stmt\_execute(stmt);  // ... (Handle results) ...] |
| This code uses an SQL library that supports parameterized queries out of the box, along with other safety features. |
| #include <pqxx/connection>  pqxx::connection C("dbname=mydatabase user=myuser  password=mypassword");  std::string username = "user1";  std::string password = "password";  pqxx::nontransaction N(C);  pqxx::result R( N.exec\_params("SELECT \* FROM users WHERE username=$1  AND password=$2", username, password) ); |

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

| **Principles(s):** [Principle 1: Secure Input Handling Prevents SQL queries from being constructed securely to prevent injection attacks.  Principle 2: Data Integrity Parameterized queries are employed to maintain database integrity to ensure data accuracy and security.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| [Critical - SQL injection could lead to serious data breach and system compromise.] | [Very Likely - SQL injection is common if best practices are not followed.] | [Medium - Parameterized queries require moderate effort to implement.] | [Very High - It is very critical to prevent SQL injection to maintain database security.] | [5 - Represents the highest risk level since it pairs critical severity with high likelihood.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.0 | S3649 | SonarQube checks for potential SQL injection vulnerabilities by identifying unparameterized queries. |
| Coverity | 2022.12 | SQLINJ | Coverity identifies places where SQL injection vulnerabilities might occur and suggests safe alternatives. |
| Fortify | 19.2 | CWE-89 | Fortify scans for potential SQL injection vulnerabilities by checking for unparameterized SQL queries. |
| Veracode | 2023.1 | SQL Injection | Veracode detects instances where SQL injection vulnerabilities are present and provides recommendations for remediation. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Prevent Memory Errors** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-MEM] | [System Crashes: Buffer overflows, use-after-free, and dangling pointers are some of the memory errors that can cause system crashes and instability.  Security vulnerabilities: The results of memory-related bugs are normally security issues of exposing sensitive information to an attacker or executing arbitrary, malicious code.  Data Corruption: Memory errors may further cause corruption of data, leading to incorrect program behavior and possible loss of data.] |

| **Noncompliant Code** |
| --- |
| [This code uses malloc() for allocating memory but afterward forgets to free the result, causing a memory leak.] |
| [ int\* data = (int\*)malloc(10 \* sizeof(int));  // Use the data  // Forget to free the memory] |

| **Compliant Code** |
| --- |
| [This code has used malloc() to allocate memory, and free() when that memory is no longer required; thus, memory leak issues cannot occur in the code.] |
| [ int\* data = (int\*)malloc(10 \* sizeof(int));  // Use the data  free(data);] |
| This code uses smart pointers courtesy of the C++ Standard Library: std::unique\_ptr, std::shared\_ptr; thus, manual memory management is handled without common memory errors. |
| #include <memory>  std::unique\_ptr<int[]> data(new int[10]);  // Use the data  // No need to explicitly free the memory,  // unique\_ptr will handle it automatically |

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

| **Principles(s):** [Principle 1: Safe Memory Management Ensures the memory is correctly allocated, used, and released to prevent leaks, corruption, and illegal access.  Principle 2: Data Integrity By safe memory management, the data integrity is preserved such that the data is not corrupted and accurate and secure.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| [Critical - Memory faults can cause system crashes, security exposures, and data corruption.] | [Likely - Memory management issues are common if best practices are not followed.] | [High - Memory faults can be tricky and time-consuming to remediate.] | [Very High - Preventing memory problems is critical to system stability and security.] | [5 - Represents the highest risk level due to the combination of critical severity with high likelihood.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.0 | S1872 | SonarQube checks for potential memory leaks and ensures proper memory management practices. |
| Coverity | 2022.12 | MEMLEAK | Coverity identifies memory leaks and use-after-free vulnerabilities, helping to prevent memory errors. |
| Fortify | 19.2 | CWE-401 | Fortify scans for potential memory leaks and improper memory handling, ensuring safe memory management. |
| Valgrind | 3.17.0 | Memcheck | Valgrind detects memory leaks, invalid memory access, and use-after-free errors, providing detailed reports for remediation. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use Assertions to Check for Invariants** |
| --- | --- | --- |
| **Assertions** | [STD-006-ASS] | [Assessment of early detection of errors: Using assertions may permit early detection of programming errors during the development process. This leads to easier debugging and reduces unexpected behavior. Code becomes more reliable as runtime checks forestall invalid conditions that result in runtime errors.  Better Documentation: Assertions may document the expected behavior and the assumptions, making them explicit, that is, from the code, about the program.] |

| **Noncompliant Code** |
| --- |
| [This code does not use assertions that might check for invalid input values.] |
| [ int divide(int numerator, int denominator) {  return numerator / denominator;  }] |

| **Compliant Code** |
| --- |
| [This code uses an assertion against division by zero.] |
| [ #include <cassert>  int divide(int numerator, int denominator) {  assert(denominator != 0);  return numerator / denominator;  }] |
| This code uses assertions that could test preconditions or postconditions for a function's behavior. |
| #include <cassert>  int square(int x) {  assert(x >= 0); // Precondition: x must be non-negative  int result = x \* x;  assert(result >= 0); // Postcondition: result must be non-negative    return result;  } |

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

| **Principles(s):** [Principle 1: Safe Code Documentation Assertions explicitly state the required behavior and assumptions, programming the logic of the code and preventing ambiguity.  Principle 2: Solid Error Handling Through the help of assertions, mistakes can be detected early on, and illegal conditions are handled by the code, making it more reliable in general.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| [Medium - Not utilizing assertions can lead to hidden programming defects, but it will not frequently produce severe security defects.] | [Likely - Programming bugs are epidemic if best practice is not employed.] | [Low - Adding assertions is generally a low-cost effort.] | [Medium - Reliability of code depends on assertions being implemented, but it is not as bad as other safety measures.] | [3 - It refers to a moderate level of threat based on the combination of severity, likelihood, and cost of remediation.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.0 | S2234 | SonarQube checks for the presence of assertions and ensures they are used appropriately to validate assumptions. |
| Coverity | 2022.12 | ASSERT | Coverity identifies places where assertions can be added to improve error detection and code reliability. |
| Fortify | 19.2 | CWE-617 | Fortify scans for potential places where assertions are missing, ensuring that assumptions and invariants are properly checked. |
| PVS-Studio | 7.16 | V556 | PVS-Studio detects conditions where assertions should be used to validate assumptions and improve code safety. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Thrown Appropriately** |
| --- | --- | --- |
| **Exceptions** | [STD-007-EXC] | [Error Handling: Exceptions are a well-structured approach to handling exceptional conditions; for example, invalid input, resource exhaustion, and network errors-gracefully, in a tidy and maintainable way.  More Readable Code: Exceptions tidy up your code as they separate the error-handling code from the program's flow.  Graceful Degradation: Exceptions allow a program to handle unexpected situations with grace by not allowing any sudden crashes, thus making a program more stable.] |

| **Noncompliant Code** |
| --- |
| [This code signals errors with error codes, which makes the code harder to read and maintain.] |
| [ int open\_file(const char\* filename) {  FILE\* file = fopen(filename, "r");  if (file == nullptr) {  return -1; // Error code  }  // ... (Use the file) ...  fclose(file);  return 0; // Success  }] |

| **Compliant Code** |
| --- |
| [This code signals an error by throwing an exception. By doing so, error handling becomes explicit, hence improving code readability.] |
| [ #include <stdexcept>  void open\_file(const char\* filename) {  FILE\* file = fopen(filename, "r");  if (file == nullptr) {  throw std::runtime\_error("Failed to open file");  }  // ... (Use the file) ...  fclose(file);  }] |
| This code uses custom exception classes to provide more specific information about the error, thus making it easier to diagnose and handle errors. |
| #include <stdexcept>  class FileOpenError : public std::runtime\_error {  public:  FileOpenError(const std::string& filename)  : std::runtime\_error("Failed to open file: " + filename) {}  };  void open\_file(const char\* filename) {  FILE\* file = fopen(filename, "r");  if (file == nullptr) {  throw FileOpenError(filename);  }  // ... (Use the file) ...  fclose(file);  } |

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

| **Principles(s):** [Principle 1: Systematic Error Handling Exceptions provide a systematic and viable means of handling errors, thus improving the reliability and maintainability of the program.  Principle 2: Readability Owing to the use of exceptions, error-handling code becomes separated from normal program flow and therefore code maintainability and readability are enhanced.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| [Medium - Use of error codes could reduce code readability and maintainability but generally doesn't result in serious security vulnerabilities.] | [Likely - Error handling with error codes is common if best practices are not followed.] | [Low - Implementing exceptions is generally a low-cost effort.] | [Medium - Exceptions are important for resilient and maintainable code.] | [3 - Represents a moderate risk level based on the sum of severity, likelihood, and remediation cost.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.0 | S3658 | SonarQube checks for the use of exceptions and ensures they are used appropriately for error handling. |
| Coverity | 2022.12 | EXCEPT | Coverity identifies places where exceptions should be used to improve error handling and code readability. |
| Fortify | 19.2 | CWE-396 | Fortify scans for the use of error codes and suggests using exceptions for better error handling. |
| PVS-Studio | 7.16 | V566 | PVS-Studio detects conditions where exceptions should be used to handle errors and improve code safety. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Correctly Allocate and Release Resources** |
| --- | --- | --- |
| [Student Choice: Resource Management] | [STD-008-RES] | [Resource Leaks: Poorly allocated or deallocated resources such as files, network, and database connections lead to resource leaks, which are very harmful and degrade system performance and stability.  System Stability: Resource leaks result in the eventual exhaustion of resources, which will crash a system or even deny services.  Data Integrity: Sometimes data gets corrupted due to the release of certain resources or may leave the system in an inconsistent state.] |

| **Noncompliant Code** |
| --- |
| [This example opens a file but leaks the resource.] |
| [ FILE\* file = fopen("data.txt", "r");  // Use the file  // Forget to close the file] |

| **Compliant Code** |
| --- |
| [This uses a try/catch block with a finally type block to make sure the file is always closed, no matter if an exception is thrown out.] |
| [ #include <stdexcept>  FILE\* file = fopen("data.txt", "r");  if (file == nullptr) {  throw std::runtime\_error("Failed to open file");  }  try {  // Use the file  } catch (...) {  fclose(file);  throw;  }  fclose(file);] |
| This example uses smart pointer std::unique\_ptr along with a custom delete which automatically closes the file when the unique\_ptr has gone out of scope. This happens even if an exception is thrown during the execution in between. |
| #include <memory>  struct FileDeleter {  void operator()(FILE\* file) const {  fclose(file);  }  };  std::unique\_ptr<FILE, FileDeleter> file(fopen("data.txt", "r"));  if (file == nullptr) {  throw std::runtime\_error("Failed to open file");  }  // Use the file  // No need to explicitly close the file,  // unique\_ptr will handle it automatically |

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

| **Principles(s):** [Principle 1: Resource Management Ensures correct allocation and release of resources to prevent leaks, system stability, and data integrity.  Principle 2: System Reliability Correct management of resources ensures the system is stable and reliable, with fewer crashes and refusals to service.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| [High - Resource leaks can lead to system crashes, performance degradation, and even corruption of data.] | [Likely - Resource management issues are common when best practices are not followed.] | [Medium - Proper resource management requires moderate effort to apply.] | [High - Proper allocation and deallocation of resources matter to maintain system stability and data integrity.] | [4 - Is the general level of risk decided by the total of severity, likelihood, and remediation cost.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.0 | S2095 | SonarQube checks for resource leaks and ensures proper management practices are followed. |
| Coverity | 2022.12 | RESLEAK | Coverity identifies resource leaks, such as unclosed file descriptors, helping to prevent resource management issues. |
| Fortify | 19.2 | CWE-775 | Fortify scans for resource leaks and improper management of resources, ensuring safe and reliable code. |
| Valgrind | 3.17.0 | Memcheck | Valgrind detects memory leaks and invalid resource usage, providing detailed reports for remediation. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Use Proper Logging and Error Handling** |
| --- | --- | --- |
| [Student Choice: Logging and Error Handling] | [STD-009-LOG] | [Debugging and Troubleshooting: Logs serve as an excellent source of information in debugging and problem resolution, thus granting easy identification and rectification of problems.  Monitoring and Analysis: Logs can be employed to monitor system behavior, analyze performance, and discover anomalies or possible security incidents.  Log-Postmortem Analysis: Logs are very essential in conducting post-mortem analysis in cases of incidents. It helps in ascertaining the actual cause of failure and thus trying to avoid similar cases in times to come.] |

| **Noncompliant Code** |
| --- |
| [This code uses printf() for simple output, which is not very informative and contextual in case debugging is needed.] |
| [ if (file == nullptr) {  printf("Error opening file.\n");  }] |

| **Compliant Code** |
| --- |
| [This is code that employs a logging framework-whether it be log4cplus, spdlog, or something else-to log error messages with timestamps, log levels (e.g., ERROR, WARNING), and context information (e.g., file name, function name).] |
| [ #include <spdlog/spdlog.h>  if (file == nullptr) {  spdlog::error("Failed to open file: {}", filename);  }] |
| This code throws an exception and, using a logging framework, logs an error message, thereby offering some structure in how the error will be handled and reported. |
| #include <stdexcept>  #include <spdlog/spdlog.h>  if (file == nullptr) {  spdlog::error("Failed to open file: {}", filename);  throw std::runtime\_error("Failed to open file");  } |

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

| **Principles(s):** [Principle 1: Structured Error Handling With logging and exceptions, errors are handled in a structured manner, and code is made more reliable and maintainable.  Principle 2: System Monitoring Logs enable proper monitoring of system behavior so that issues can be identified and resolved in advance.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| [Medium - Poor logging and error handling can make debugging and repair more difficult but generally is not a cause for critical security flaws.] | [Likely - Logging and error handling problems are common unless best practices are employed.] | [Low - Correct logging and error handling are typically a low-cost solution.] | [Medium - Correct logging and error handling are required to provide stable and maintainable code.] | [3 - A moderate risk level based on the severity, likelihood, and remediation cost.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.0 | S2629 | SonarQube checks for proper logging practices and ensures that meaningful log messages are used. |
| Coverity | 2022.12 | LOGCHK | Coverity identifies places where logging is missing or inadequate and suggests improvements. |
| Fortify | 19.2 | CWE-778 | Fortify scans for potential logging issues and ensures that logs are used effectively for error handling and monitoring. |
| Logcheck | 1.3.23 | Log Analysis | Logcheck analyzes log files to identify anomalies and security incidents, helping to ensure proper logging practices. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Use Meaningful and Informative Comments** |
| --- | --- | --- |
| [Student Choice: Code Comments] | [STD-010-COM] | [Code Readability: Smaller and concise comments enhance code readability and, consequently, maintainability for other developers to go through and perform changes accordingly.  Documentation: Comments provide very useful documentation in explaining the purpose of the code, logic, and design decisions behind the code.  Debugging and Maintenance: The presence of comments considerably helps during debugging and maintenance, thus giving context and insight into the behavior of the code.] |

| **Noncompliant Code** |
| --- |
| [Irrelevant comments-only code, which does not give any useful information.] |
| [ // This function does something  int do\_something(int x) {  // Increment x  return x + 1;  }] |

| **Compliant Code** |
| --- |
| [This code includes informative comments: describes what the function does and what the parameters are.] |
| [ // This function increments the given integer value  int increment(int x) {  return x + 1;  }] |
| Comments explaining complex logic or not-so-obvious algorithms. |
| // This function calculates the factorial of a number using recursion  int factorial(int n) {  if (n == 0) {  return 1;  } else {  return n \* factorial(n - 1);  }  } |

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

| **Principles(s):** [Principle 1: Code Documentation Comments are effective documentation of the purpose, logic, and design decisions of the code and therefore make the code easy to understand and maintain.  Principle 2: Readability Clear and concise comments enhance the readability of the code in a manner that developers can easily read and work with the code.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| [Medium - Lack of sufficient meaningful comments would make the code difficult to understand and maintain, which could lead to debugging and update issues.] | [Likely - One of the most common issues if best practices are not followed is lack of comments.] | [Low - Adding meaningful comments is generally a low-cost effort.] | [Medium - To have comments guarantee the code is stable and maintainable.] | [3 - Is an average risk level based on severity, likelihood, and cost of remediation together.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.0 | S125 | SonarQube checks for the presence of comments and ensures they are meaningful and informative. |
| Coverity | 2022.12 | COMCHK | Coverity identifies places where comments are missing or inadequate and suggests improvements. |
| Fortify | 19.2 | CWE-615 | Fortify scans for the absence of meaningful comments and ensure that comments are used effectively for documentation. |
| Clang-Tidy | 12.0 | readability-misleading-indentation | Clang-Tidy checks for misleading or missing comments and suggests improvements to enhance code readability. |

### 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 with the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

[Automation is key to guaranteeing coding standards adherence and a secure development process. Green Pace's existing DevOps process can be made better by applying automation at various stages of the DevSecOps pipeline. I provide guidelines below on where and how to modify the DevOps process to automate coding standards enforcement, referring to the provided DevSecOps diagram.

In the Assess and Plan phase, automation tools like threat modeling and regulatory change monitoring systems must be utilized. These tools can provide real-time updates and notifications on regulation updates and can integrate with backlogs tools for the automation of task prioritization. For example, using tools like OWASP Threat Dragon in threat modeling and integrating it with project management tools like Jira, security tasks can be monitored and prioritized.

As a part of the Design phase, the design verification tools, and security testing tools have to be included. The automated security testing tools can ensure adherence to security-driven design principles and best practices. SonarQube and Fortify are automatically verifying tools that check the code for security best practices and security design flaws.

In the Build phase, there must be use of continuous integration (CI) pipelines and trusted repository management tools. Automated build tools that ensure the use of trusted repositories and open-source components in a secure manner can be integrated into CI pipelines with security auditing. For instance, integrating tools like Jenkins with dependency scan tools like Snyk ensure that only secure dependencies are used while building.

In the Verify and Test stage, the CI/CD pipeline must include vulnerability scanning tools and security testing tools. Automated vulnerability scanning can be achieved using tools like OWASP ZAP and Burp Suite in the testing phase.

In the Transition and Health Check phase, IaC tools and penetration testing tools must be used. Automating security setting configuration and deployment through IaC tools and automating regular penetration testing maintains a secure posture. Automated security configuration can be done with tools like Terraform and AWS CloudFormation, and penetration testing can be done using Metasploit.

During the Monitor and Detect phase, log collection tools, SIEM tools, and analytics tools must be used. Automated security event log collection tools and SIEM tools can be run continuously to monitor security events, and machine learning-based analytics tools can detect anomalies and threats. Logs can be gathered and analyzed by means of tools such as Splunk and Elasticsearch, and anomaly detection can be achieved by means of tools such as Anomaly Detection.

During the Respond phase, incident response tools and orchestration tools must be implemented. Incident response tasks can be automated, for example, blocking attacks and undoing changes, using orchestration tools like Ansible and Puppet.

Finally, in the Maintain and Stabilize phase, there is a need to utilize baseline assessment tools and configuration management tools. Automated scanning of baseline security, together with configuration management solutions, ensures that systems are reverted to a steady baseline when a problem arises. Baseline and configuration management can be automated with products like Chef and Puppet.

With the inclusion of these automation steps into the DevSecOps pipeline, Green Pace can provide continuous enforcement and compliance with the set standards, thereby enhancing the security posture of the company. Not only is secure coding practice guaranteed to be in effect, but security is embedded as a part of the development process, from planning to production.]

### 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-001-DAT | High | Likely | Medium | High | 4 |
| STD-002-DAT | High | Very Likely | Medium | High | 5 |
| STD-003-STR | High | Likely | Medium | High | 4 |
| STD-004-SQL | Critical | Very Likely | Medium | Very High | 5 |
| STD-005-MEM | Critical | Likely | High | Very High | 5 |
| STD-006-ASS | Medium | Likely | Low | Medium | 3 |
| STD-007-EXC | Medium | Likely | Low | Medium | 3 |
| STD-008-RES | High | Likely | Medium | High | 4 |
| STD-009-LOG | Medium | Likely | Low | Medium | 3 |
| STD-010-COM | Medium | Likely | Low | Medium | 3 |
| STD-001-DAT | High | Likely | Medium | High | 4 |
| STD-002-DAT | High | Very Likely | Medium | High | 5 |
| STD-003-STR | High | Likely | Medium | High | 4 |

### 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 | [Definition: Encryption at rest refers to the fact that data once stored on a physical storage medium such as a hard disk, SSD, or tape backup is secured. The stored data is encrypted, and without authorization, any person who comes into physical possession of the storage device cannot use it.  How It Is Used: Information in databases, file systems, and other media storage are encrypted with algorithms such as AES-256. The keys are stored securely so that only authenticated users can decrypt and read the information.  Why the Policy is Enacted: The policy has the consequence that confidential data cannot be hijacked or divulged in the event where media for storage gets lost, pilfered, or compromised. It is imposed on any information that is stored by Green Pace infrastructure, for example, customers' data, monetary data, and intellectual properties.] |
| Encryption in flight | [Definition: Flight encryption is the encryption of data as it travels within networks. It includes data passed through the internet, intranets, or other communication systems.  Usage: Data is encrypted by systems such as TLS (Transport Layer Security) as it travels. It serves to protect data from being intercepted or altered by unauthorized individuals.  Why the Policy Exists: The policy exists to ensure that data is stored confidentially and in integrity while it is being transferred between servers, clients, and other network devices. It controls all means of communication through which Green Pace transfers sensitive data, including emails, web traffic, and API calls.] |
| Encryption in use | [Description: Encryption in use accounts for data being protected as it is used and processed by applications. It achieves this by enabling data to remain protected, even when the data is in active use.  Application: These methods involve such as homomorphic encryption and secure enclaves, to enable data being processed to be encrypted without exposing plain text. It also includes encryption that occurs within the application itself, and encryption of memory.  Why the Policy is Used: The policy ensures sensitive data is maintained when it is being processed to reduce exposure risks when there are memory dumps, unauthorized access, or other security vulnerabilities. The policy is used in any application and process that handles sensitive data in Green Pace's infrastructure.] |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | [Explanation: Authentication is the process of verifying users, devices, or systems' identities before allowing access to resources. It ensures that only the rightful entities get access to sensitive information and systems.  How it is Used: Multi-factor authentication (MFA) is applied to authenticate users' identities using more than one factor such as passwords, biometric data, and security tokens. Systems also utilize certificate-based authentication for services and devices.  Why the Policy Exists: This policy ensures that only verified and trusted parties gain access to Green Pace's systems and data. It avoids unauthorized entry and potential security breaches by confirming robust identity verification procedures. It applies to all user logins, including employees, contractors, and third-party partners.] |
| Authorization | [Explanation: Authorization is the process of granting or denying access to resources depending on an authenticated subject's privileges. It ensures users and systems receive access to only those resources for which they have permission.  How it is Used: Role-based access control (RBAC) and attribute-based access control (ABAC) are used for permissions management. Access rights are assigned depending on roles, business functions, and attributes such as place and time.  Why the Policy Exists: This policy assists in granting users and systems appropriate levels of access, reducing data breach and abuse risk. It regulates changes to the database, adding new users, and user access levels to files and systems in Green Pace.] |
| Accounting | [Definition: Accounting, also known as auditing or logging, is the process of tracking and recording user activity, system activity, and resource access. It is used to track all activities for accounting and compliance.  How it is Used: User logins, database modifications, additions of new users, access to files, and other such significant actions are recorded in detail. The logs are scanned and reviewed at regular intervals to detect anomalies and potential security intrusions.  Why the Policy is Applied: The policy guarantees all activities within the systems of Green Pace are traced and traceable. It aids in the discovery and investigation of security violations, regulatory compliance, and system integrity. It applies to all user-accessed files, user behavior, and system events.] |

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

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

### Map the Principles

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

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

* Operating system logs
* Firewall logs
* Anti-malware logs
  + 1. Appropriate Data Types (STD-001-DAT)
* Principles: 4 (Keep It Simple), 10 (Adopt a Secure Coding Standard)
* Reasoning:
* Keep It Simple: Proper data types make code simpler and less complicated and thus security vulnerabilities are simpler to spot and repair.
* Adopt a Secure Coding Standard: Secure coding standards adhere to the use of proper data types to prevent vulnerabilities like overflow and truncation.
  + 1. Validating and Sanitizing Data Values (STD-002-DAT)
* Principles: 1 (Validate Input Data), 7 (Sanitize Data Sent to Other Systems)
* Reasoning:
  + Validate Input Data: Prevents input data from being susceptible to harmful attacks like SQL injection and XSS.
  + Sanitize Data Being Sent to Other Systems: Prevents data being sent to other systems from being non-sanitized so as not to breach security in interfaced systems.

1. Handle String Operations Safely (STD-003-STR)

* Principles: 6 (Abide by the Principle of Least Privilege), 10 (Comply with a Secure Coding Standard)
* Reasoning:
* Follow the Principle of Least Privilege: Secure manipulation of strings prohibits data tampering and unauthorized data access.
* Follow a Secure Coding Standard: Implementation of secure coding rules ensures string manipulation does not produce weaknesses like buffer overflows.

1. Prevent SQL Injection Vulnerabilities (STD-004-SQL)

* Principles: 1 (Validate Input Data), 3 (Architect and Design for Security Policies)
* Justification:
* Validate Input Data: Verifies that all user data is validated to prevent SQL injection attacks.
* Architect and Design for Security Policies: Incorporating security into the design ensures that SQL injection vulnerabilities are addressed from the start.

1. Prevent Memory Errors (STD-005-MEM)

* Principles: 6 (Follow the Principle of Least Privilege), 9 (Implement Effective Quality Assurance Techniques)
* Justification:
* Follow the Principle of Least Privilege: Effective management of memory ensures that memory is only used, when necessary, thus removing the possibility of unauthorized access.
* Use Effective Quality Assurance Techniques: Stringent testing and code inspection help to catch and minimize memory bugs before deployment.

1. Use Assertions to Verify Invariants (STD-006-ASS)

* Principles: 2 (Respect Compiler Warnings), 5 (Default Deny)
* Reasoning:
* Respect Compiler Warnings: Using assertions helps to catch early on possible bugs during the development process.
* Default Deny: Avoiding incorrect states being caught late by assertions helps to maintain secure defaults.

1. Thrown Properly (STD-007-EXC)

* Principles: 3 (Architect and Design for Security Policies), 9 (Use Effective Quality Assurance Techniques)
* Justification:
* Architect and Design for Security Policies: Correct exception handling is implemented in the system to safely handle errors.
* Use Effective Quality Assurance Techniques: Proper utilization of exceptions increases code quality and security.

1. Correctly Allocate and Release Resources (STD-008-RES)

* Principles: 4 (Keep It Simple), 6 (Adhere to the Principle of Least Privilege)
* Justification
* Keep It Simple: Minimized resource management reduces the risk of errors and vulnerabilities.
* Follow the Principle of Least Privilege: Properly ensuring resources are assigned and released limits the opportunity for unauthorized access.

1. Apply Correct Logging and Error Handling (STD-009-LOG)

* Principles: 8 (Practice Defense in Depth), 9 (Use Effective Quality Assurance Techniques)
* Rationale:
* Practice Defense in Depth: Full logging and error handling provide multiple security monitors and response phases.
* Use Effective Quality Assurance Techniques: Adequate logging and error handling improve the overall quality and security of the code.

1. Use Meaningful and Informative Comments (STD-010-COM)

* Principles: 4 (Keep It Simple), 9 (Use Effective Quality Assurance Techniques)
* Justification:
* Keep It Simple: Code maintainability and readability are improved by brevity and concision of comments.
* Use Effective Quality Assurance Techniques: Descriptive comments facilitate code reviews and debugging and make the code in general better.

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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