**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** | **Explanation** |
| --- | --- |
| 1. ValidateInput Data | Ensure data adheres to expected formats and ranges. This validates the input and prevents injection attacks. |
| 1. Heed Compiler Warnings | Address compiler warnings. These warnings are alerts to potential issues that could result in security weaknesses. |
| 1. Architect and Design for Security Policies | Integrate security during the design and architecture phase of development. This sets the pace for security to be a fundamental part of the software resulting in strong security implementation. |
| 1. Keep It Simple | Simple code is easier to understand and maintain. By “keeping it simple” we can reduce risk of errors and mitigate vulnerabilities efficiently. |
| 1. Default Deny | Access permissions should not be granted unless there is explicit allowance. This secures the system by reducing unauthorized access that may result in unpermitted actions. |
| 1. Adhere to the Principle of Least Privilege | Always provide the minimum level of access necessary to perform needed functions. This applies to users and processes. By limiting privileges, impact of potential security breaches can be lessened. |
| 1. Sanitize Data Sent to Other Systems | This involves validation of data and removing or encoding data to ensure it conforms to expected formats and constraints. It is essentially sanitizing the data and aides in protecting against injection attacks. |
| 1. Practice Defense in Depth | Implement multiple layers of security to provide back-up defense in the event of a breach. These added barriers enhance overall security. It is important to remember to balance layers of security with functionality of the system. |
| 1. Use Effective Quality Assurance Techniques | Employ quality assurance techniques such as testing and code reviews. These assist in earlier detection of security vulnerabilities and result in timely fixes. |
| 1. Adopt a Secure Coding Standard | Use secure coding standards such as SEI CERT C++ Coding Standard. This provides guidelines for writing code and outlines best practices. This will result in more robust security measures that are applied in a consistent manner. |

### 

### C/C++ Ten Coding Standards

### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Do not define a C-style variadic function - C-style variadic functions are error-prone and hard to maintain. They lack type safety, making it easy to introduce bugs and vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Using a C-style variadic function. |
| int sum(int count, ...) {  va\_list args;  va\_start(args, count);  int total = 0;  for (int i = 0; i < count; ++i) {  total += va\_arg(args, int);  }  va\_end(args);  return total;  } |

| **Compliant Code** |
| --- |
| Using a C++ variadic template function. |
| template<typename... Args>  int sum(Args... args) {  return (args + ...);  } |

| **Principles(s):**   * Adopt a Secure Coding Standard - C-style variadic functions lack type safety, making them prone to errors. Secure coding standards discourage their use. * Heed Compiler Warnings - These functions are often flagged by compilers, and addressing these warnings is crucial for security. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | LANG.STRUCT.ELLIPSIS | Ellipsis Detection |

#### 

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Do not declare or define a reserved identifier - Reserved identifiers are meant for use by the implementation. Using them can lead to undefined behavior and conflicts with future versions of the language or standard library. |

| **Noncompliant Code** |
| --- |
| Defining a reserved identifier. |
| int \_\_reserved = 10; |

| **Compliant Code** |
| --- |
| Using a non-reserved identifier. |
| int reserved\_var = 10; |

| **Principles(s):**   * Default Deny - Using reserved identifiers can lead to conflicts and undefined behavior, similar to how default-deny restricts unauthorized access. * Adopt a Secure Coding Standard - This principle ensures that developers avoid reserved identifiers, maintaining compatibility and security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0. | LANG.ID.NU.MK  LANG.STRUCT.DECL.RESERVED | Reserved Identifier Detection |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Guarantee that storage for strings has sufficient space for character data and the null terminator - Insufficient storage for strings can lead to buffer overflows, which are a common source of security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Allocating insufficient space for the string. |
| char str[5];  strcpy(str, "hello"); |

| **Compliant Code** |
| --- |
| Allocating sufficient space for the string. |
| char str[6];  strcpy(str, "hello"); |

| **Principles(s):**   * Validate Input Data - Ensuring sufficient space for strings is crucial to avoid buffer overflows, a key aspect of input validation. * Adopt a Secure Coding Standard - Proper string handling is a core part of secure coding practices. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | MISC.MEM.NTERM  LANG.MEM.BO  LANG.MEM.TO | Null Terminator Absence  Buffer Overrun Detection  Type Overrun Detection |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Use valid references, pointers, and iterators to reference elements of a container - Using invalid references, pointers, or iterators can lead to undefined behavior and potential security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Using an invalid iterator. |
| std::vector<int> vec = {1, 2, 3};  auto it = vec.end();  int value = \*it; |

| **Compliant Code** |
| --- |
| Using a valid iterator. |
| std::vector<int> vec = {1, 2, 3};  auto it = vec.begin();  int value = \*it; |

| **Principles(s):**   * Sanitize Data Sent to Other Systems - Ensuring valid references and pointers helps in preventing invalid or malicious data from being processed. * Practice Defense in Depth - Using valid references adds another layer of protection, preventing undefined behavior that could lead to vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | ALLOC.UAF | Invalid Iterator Detection |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Do not access freed memory - Accessing freed memory can lead to undefined behavior, crashes, and potential security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Accessing freed memory. |
| int\* ptr = new int(10);  delete ptr;  int value = \*ptr; |

| **Compliant Code** |
| --- |
| Ensuring memory is not accessed after being freed. |
| int\* ptr = new int(10);  delete ptr;  ptr = nullptr; |

| **Principles(s):**   * Practice Defense in Depth - Preventing access to freed memory adds a layer of security by avoiding undefined behavior. * Heed Compiler Warnings - Compilers often warn about potential issues with memory management, which should be addressed to maintain security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | ALLOC.UAF | Freed Memory Access Detection |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Detect and handle memory allocation errors - Properly handling memory allocation errors prevents crashes and potential security vulnerabilities due to failed memory operations. |

| **Noncompliant Code** |
| --- |
| Not checking for memory allocation failure. |
| int\* ptr = new int[1000];  //No check for allocation success |

| **Compliant Code** |
| --- |
| Checking for memory allocation failure. |
| int\* ptr = new(std::nothrow) int[1000];  if (!ptr) {  // Handle allocation failure  std::cerr << "Memory allocation failed" << std::endl;  // Take appropriate action, such as cleaning up and exiting  exit(EXIT\_FAILURE);  } |

| **Principles(s):**   * Use Effective Quality Assurance Techniques - Properly handling memory allocation errors is a key aspect of ensuring code quality and preventing security issues. * Adopt a Secure Coding Standard - This principle emphasizes the importance of handling all memory-related operations securely. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | 45 D | Memory Allocation Error Detection |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle all exceptions - Properly handling exceptions prevents crashes and potential security vulnerabilities due to unhandled exceptional conditions. |

| **Noncompliant Code** |
| --- |
| Not handling exceptions. |
| void func() {  throw std::runtime\_error("Error");  }  void caller() {  func(); // No exception handling  } |

| **Compliant Code** |
| --- |
| Handling exceptions. |
| void func() {  throw std::runtime\_error("Error");  }  void caller() {  try {  func();  } catch (const std::runtime\_error& e) {  // Handle exception  }  } |

| **Principles(s):**   * Use Effective Quality Assurance Techniques - Proper exception handling is critical for ensuring that software behaves reliably under all conditions. * Adopt a Secure Coding Standard - Handling exceptions is a fundamental part of secure coding practices. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | LANG.STRUCT.UCTCH | Exception Handling Detection |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory Protection | [STD-008-CPP] | Do not delete an array through a pointer of the incorrect type - Deleting an array through a pointer of the incorrect type can lead to undefined behavior and potential memory corruption. |

| **Noncompliant Code** |
| --- |
| Deleting an array through a pointer of the incorrect type. |
| int\* arr = new int[10];  delete arr; // Incorrect deletion |

| **Compliant Code** |
| --- |
| Deleting an array through a pointer of the correct type. |
| int\* arr = new int[10];  delete[] arr; // Correct deletion |

| **Principles(s):**   * Practice Defense in Depth - Properly deleting arrays through the correct pointer type is part of ensuring memory safety. * Heed Compiler Warnings - Misuse of delete operations may trigger compiler warnings, which should be addressed to prevent undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | ALLOC.TM | Array Deletion Type Mismatch Detection |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Data Type | [STD-009-CPP] | Do not access an object outside of its lifetime - Accessing an object outside of its lifetime can lead to undefined behavior and potential security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Accessing an object outside of its lifetime. |
| int\* ptr = nullptr;  {  int val = 10;  ptr = &val;  }  int value = \*ptr; // Accessing out-of-scope object |

| **Compliant Code** |
| --- |
| Ensuring object access within its lifetime. |
| int value = 0;  {  int val = 10;  value = val; // Access within scope  } |

| **Principles(s):**   * Validate Input Data - Ensuring that objects are only accessed within their lifetime is crucial to prevent undefined behavior and data corruption. * Practice Defense in Depth - This principle supports ensuring that objects are handled safely at all times, adding another layer of security. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | High | Medium | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | IO.UAC  ALLOC.UAF | Static Object Initialization Cycle Detection  Unordered Initialization Detection |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Declarations and Initializations | [STD-010-CPP] | Avoid cycles during initialization of static objects - Cycles during the initialization of static objects can lead to undefined behavior and potential security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Creating initialization cycles. |
| struct A {  static B b;  };  struct B {  static A a;  };  A A::b;  B B::a; |

| **Compliant Code** |
| --- |
| Avoiding initialization cycles. |
| struct A {  static A& instance() {  static A instance;  return instance;  }  };  struct B {  static B& instance() {  static B instance;  return instance;  }  }; |

| **Principles(s):**   * Architect and Design for Security Policies - Avoiding initialization cycles is a key part of designing secure and stable software architectures. * Heed Compiler Warnings - Initialization cycles often result in compiler warnings, which should be addressed to ensure secure initialization processes. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | LANG.STRUCT.INIT.CYCLE  LANG.STRUCT.INIT.UNORDERED | Initialization Cycle  Unordered Initialization |

### 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.

### Automation

Provide a written explanation using the image provided.



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

Automation will be integral to enforcing and ensuring compliance with the coding standards defined in this policy. As Green Pace transitions to a DevSecOps model, automation will become even more critical. Specifically, the integration of CodeSonar and the LDRA tool suite into the existing DevOps pipeline will provide comprehensive security and compliance coverage. CodeSonar, known for its advanced static analysis capabilities, will be implemented early in the development process to automatically detect and report on potential vulnerabilities in the codebase, ensuring that any issues are addressed before they progress further down the pipeline. This tool will help maintain adherence to secure coding standards by providing real-time feedback to developers within their IDEs and during the code verification stages.

The LDRA tool suite will complement CodeSonar by offering both static and dynamic analysis, as well as automated testing capabilities. It will be particularly effective during the verification and pre-production stages, where it will enforce coding standards through rigorous testing and code coverage analysis. By automating these processes with CodeSonar and LDRA, Green Pace will ensure continuous enforcement of security standards and maintain compliance across all stages of development, from initial coding to final production, without compromising the efficiency of its established DevOps workflow.

### 

### Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Medium | Medium | High | 4 |
| STD-002-CPP | Low | Low | Low | Low | 1 |
| STD-003-CPP | High | High | Medium | High | 5 |
| STD-004-CPP | High | Medium | High | High | 5 |
| STD-OO5-CPP | High | High | Medium | High | 5 |
| STD-006-CPP | High | High | Medium | High | 5 |
| STD-007-CPP | Low | Medium | Medium | Medium | 3 |
| STD-008-CPP | Low | Low | Medium | Medium | 2 |
| STD-009-CPP | High | Medium | High | Medium | 4 |
| STD-010-CPP | Low | Low | Medium | Low | 3 |

### 

### Create Policies for Encryption and Triple A

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | All sensitive data stored within Green Pace’s systems must be encrypted at rest using industry-standard encryption algorithms, such as AES-256. This includes data stored in databases, file systems, and backups. Encryption keys must be managed securely, with regular rotation and access controls to prevent unauthorized access.  This policy applies to all data repositories within the organization, including production, staging, and development environments. Encrypting data at rest ensures that even if physical storage devices are compromised, the data remains protected. |
| Encryption in flight | Data in transit between systems, whether internal or external, must be encrypted using strong transport protocols, such as TLS 1.2 or higher. This includes data exchanged over networks, APIs, and web services. Certificates and encryption keys must be managed securely to ensure the integrity of encrypted communication.  This policy applies to all communications between Green Pace systems, including between servers, clients, and external partners. Encrypting data in flight prevents interception and unauthorized access during transmission. |
| Encryption in use | Sensitive data being processed or accessed by applications must be protected using techniques such as in-memory encryption, hardware-based security modules (HSMs), or secure enclaves. Applications must be designed to minimize exposure of unencrypted sensitive data in memory or during processing.  This policy applies to all applications that handle sensitive data, particularly in scenarios where data is decrypted for processing. Encrypting data in use reduces the risk of data exposure through memory dumps, unauthorized memory access, or side-channel attacks. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | All user and system access to Green Pace systems must be authenticated using strong, multi-factor authentication methods. This includes user logins, which must be protected with complex passwords and additional authentication factors such as biometrics or one-time passwords (OTPs). Access tokens and certificates used for system authentication must be securely managed to prevent unauthorized access.  This policy applies to all user logins and system accesses, ensuring that only verified individuals and systems can interact with Green Pace resources. Regular reviews and updates to authentication mechanisms are essential to maintain security against evolving threats. |
| Authorization | Access to Green Pace resources must be granted based on the principle of least privilege. This includes setting and reviewing user levels of access, ensuring that users only have permissions necessary for their roles. Any addition of new users or changes to existing user permissions, such as changes to the database or access to sensitive files, must be carefully controlled and audited.  This policy governs the authorization of all user actions, including database changes, the addition of new users, and access to critical files and systems. By implementing strict access controls and regularly reviewing them, Green Pace can minimize the risk of unauthorized actions that could compromise security. |
| Accounting | All actions within Green Pace systems, including user logins, changes to the database, the addition of new users, and access to files by users, must be logged and auditable. These logs should include detailed information such as user identity, the nature of the action, the time of access, and the specific resources affected. Logging mechanisms must be secured against tampering, and logs should be regularly reviewed to detect any suspicious activities.  This policy applies to all systems and processes within Green Pace, ensuring that every action taken by users or systems is traceable. Comprehensive logging and regular audits are critical for identifying and responding to potential security incidents, providing accountability and enhancing the overall security posture. |

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 | 07/28/2024 | Milestone | Kelly Sebastiani | Passed |
| 3.0 | 08/11/2024 | Project One | Kelly Sebastiani |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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