**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 | Allows for a program to ensure that it is receiving input within certain parameters. This can prevent many vulnerabilities such as SQL injection attacks. |
| 1. Heed Compiler Warnings | Treat warnings as errors so that vulnerabilities are less likely. This allows for issues to be addressed much faster in the development process. |
| 1. Architect and Design for Security Policies | A program should be designed around security. Security models and threat assessments should guide the structure and interaction of system components to minimize risk and exposure. |
| 1. Keep It Simple | Simple programs are easier to keep secure. Less opportunities for vulnerabilities. Simple, well understood designs are easier to maintain, audit, and secure over time. |
| 1. Default Deny | Access should be denied by default to prevent potentially giving away too much access. Further access can be given later if needed. |
| 1. Adhere to the Principle of Least Privilege | Each user or process should operate using the minimum privileges necessary to complete its task. This can help keep the damage isolated if a vulnerability was exploited. |
| 1. Sanitize Data Sent to Other Systems | Data exchanged between systems should be sanitized to prevent injection attacks and data leakage. If anything, important is intercepted during the exchange it could potentially be exploited. |
| 1. Practice Defense in Depth | Involves using multiple layers of security controls throughout an information system. If one layer fails, others continue to provide protection. This strategy includes firewalls, intrusion detection systems, secure coding, encryption, and user training. |
| 1. Use Effective Quality Assurance Techniques | Robust quality assurance practices, such as code reviews, automated testing, and static analysis, help detect and eliminate vulnerabilities early. Continuous support will allow for further protection. |
| 1. Adopt a Secure Coding Standard | Pick a coding standard for the language your program will be written in. This will act as a guidebook to eliminate common security issues and allow for a more robust program. S |

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Use Fixed-Width Integer Types |

| **Noncompliant Code** |
| --- |
| Uses plain int which varies in size on different systems. |
| int count = 100000;  int total = count \* 1000; |

| **Compliant Code** |
| --- |
| Uses int32\_t for consistency and safety |
| #include <cstdint>  int32\_t count = 100000;  int64\_t total = (int64\_t)count \* 1000; |

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

| #2 Heed Compiler Warnings – Prevents compiler-dependent behavior.  **#10 Adopt a Secure Coding Standard** – Promotes portability and reliability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.9 | performanceTypeMismatch | Detects unsafe use of varying-width integers |
| Clang-Tidy | 17.0 | cppcoreguidelines-\* | Helps enforce modern C++ best practices |
| SonarQube | 10.2 | cpp:S100 | Warns on type mismatches, potential overflows |
|  |  |  |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Check for Integer Overflows |

| **Noncompliant Code** |
| --- |
| Multiplies large numbers without checking for overflow. |
| int result = a \* b; |

| **Compliant Code** |
| --- |
| Uses std::numeric\_limits to detect overflow. |
| #include <limits>  if (a > 0 && b > std::numeric\_limits<int>::max() / a) {  // Handle error  } else {  int result = a \* b;  } |

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

| **#1 Validate Input Data** – Ensures inputs don’t exceed expected bounds.  **#10 Adopt a Secure Coding Standard** – Prevents undefined behavior from unchecked math operations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.9 | warning: possible overflow | Checks for risky arithmetic operations |
| Clang-Tidy | 17.0 | clang-analyzer-core.UndefinedBinaryOperatorResult | Detects overflow and undefined behavior |
|  |  |  |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Avoid Dangerous String Functions |

| **Noncompliant Code** |
| --- |
| Uses strcpy with no length check. |
| char buffer[10];  strcpy(buffer, input); |

| **Compliant Code** |
| --- |
| Uses strncpy with length limit. |
| char buffer[10];  strncpy(buffer, input, sizeof(buffer) - 1);  buffer[9] = '\0'; |

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

| **#1 Validate Input Data** – Prevents input overflows.  **#7 Sanitize Data Sent to Other Systems** – Blocks unsafe inputs before output. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Flawfinder | 2.0.19 | strcpy | Identifies dangerous string manipulation |
| SonarQube | 10.2 | cpp:S5867 | Warns on unsafe C-style string handling |
|  |  |  |  |
|  |  |  |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Use Parameterized Queries |

| **Noncompliant Code** |
| --- |
| Concatenates user input into SQL string. |
| std::string query = "SELECT \* FROM users WHERE name = '" + userInput + "'"; |

| **Compliant Code** |
| --- |
| Uses prepared statements to separate code and data. |
| sqlite3\_stmt\* stmt;  sqlite3\_prepare\_v2(db, "SELECT \* FROM users WHERE name = ?", -1, &stmt, nullptr);  sqlite3\_bind\_text(stmt, 1, userInput.c\_str(), -1, SQLITE\_STATIC); |

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

| **#1 Validate Input Data** – Prevents injection via unsafe inputs.  **#7 Sanitize Data Sent to Other Systems** – Protects external systems. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.2 | cpp:S3649 | Detects unsafe SQL query building |
| Bandit | 1.7.5 | B608 | Flags unsafe SQL statements |
|  |  |  |  |
|  |  |  |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Avoid Buffer Overflows |

| **Noncompliant Code** |
| --- |
| Writes beyond array size. |
| int arr[5];  arr[5] = 10; // out-of-bounds |

| **Compliant Code** |
| --- |
| Validates array index before access. |
| int arr[5];  if (index >= 0 && index < 5) {  arr[index] = 10;  } |

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

| **#5 Default Deny** – Prevents assumption of valid input.  **#8 Practice Defense in Depth** – Protects even if logic fails. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.21.0 | memcheck | Detects out-of-bounds memory access |
| Cppcheck | 2.9 | bufferOverrun | Finds unsafe array access |
|  |  |  |  |
|  |  |  |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Do Not Use Assertions for Runtime Checks |

| **Noncompliant Code** |
| --- |
| Uses assert to check user input. |
| assert(userInput != nullptr); |

| **Compliant Code** |
| --- |
| Uses proper error handling. |
| if (userInput == nullptr) {  throw std::invalid\_argument("Null input not allowed");  } |

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

| **#4 Keep It Simple** – Avoids logic that can be compiled out.  **#9 Use Effective Quality Assurance Techniques** – Promotes reliable testing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17.0 | misc-static-assert | Ensures static asserts used correctly |
| SonarQube | 10.2 | cpp:S4124 | Warns against misuse of runtime assertions |
|  |  |  |  |
|  |  |  |  |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Catch Exceptions by Reference |

| **Noncompliant Code** |
| --- |
| Catches exception by value. |
| try {  throw std::runtime\_error("error");  } catch (std::runtime\_error e) {  // error  } |

| **Compliant Code** |
| --- |
| Catches exception by reference. |
| try {  throw std::runtime\_error("error");  } catch (const std::runtime\_error& e) {  // handle error  } |

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

| **#10 Adopt a Secure Coding Standard** – Prevents slicing and improves efficiency. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17.0 | cppcoreguidelines-catch-by-reference | Warns against catching by value |
| SonarQube | 10.2 | cpp:S1181 | Checks for exception safety |
|  |  |  |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Pointer Safety | [STD-008-CPP] | Avoid Dereferencing Null Pointers |

| **Noncompliant Code** |
| --- |
| Dereferences a pointer without checking for null. |
| int\* ptr = nullptr;  int value = \*ptr; // undefined behavior |

| **Compliant Code** |
| --- |
| Checks the pointer before dereferencing. |
| int\* ptr = getPointer();  if (ptr != nullptr) {  int value = \*ptr;  } |

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

| **#1 Validate Input Data** – Checks for null before use.  **#8 Practice Defense in Depth** – Protects against failure propagation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.21.0 | memcheck | Detects null pointer dereferencing |
| Clang Static Analyzer | 17.0 | core.NullDereference | Identifies potential null pointer usage |
|  |  |  |  |
|  |  |  |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File Handling | [STD-009-CPP] | Always Check File Operations for Success |

| **Noncompliant Code** |
| --- |
| Assumes file opens successfully. |
| std::ifstream file("data.txt");  file >> data; |

| **Compliant Code** |
| --- |
| Checks if the file was successfully opened. |
| std::ifstream file("data.txt");  if (file.is\_open()) {  file >> data;  } else {  // Handle error  } |

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

| **#6 Least Privilege** – Prevents assuming access without validation.  **#9 Use Effective Quality Assurance Techniques** – Encourages proper handling of file operations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.9 | resourceLeak | Warns when file handles are not properly checked or closed |
| SonarQube | 10.2 | cpp:S2674 | Detects missing checks for file I/O operations |
|  |  |  |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Loop Safety | [STD-0010-CPP] | Prevent Infinite Loops with Proper Conditions |

| **Noncompliant Code** |
| --- |
| Infinite loop due to no update condition. |
| int i = 0;  while (i < 10) {  std::cout << i;  // Missing i++  } |

| **Compliant Code** |
| --- |
| Loop variable is properly updated. |
| int i = 0;  while (i < 10) {  std::cout << i;  i++;  } |

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

| **#4 Keep It Simple** – Reduces complexity and avoids accidental infinite loops.  **#9 Use Effective Quality Assurance Techniques** – Supports testable, safe loop design. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.2 | cpp:S2189 | Detects loops with constant conditions or no update |
| Clang-Tidy | 17.0 | bugprone-infinite-loop | Flags infinite loops without a proper termination condition |

Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



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

Automation at Green Pace will be embedded throughout the DevSecOps pipeline to enforce compliance with secure coding standards. Static analysis tools like Cppcheck and SonarQube will automatically scan code during development, while CI/CD pipelines will block deployments that violate security policies. In production, monitoring tools will track system behavior, log user actions, and trigger alerts for any suspicious activity. This integration ensures that security is continuously enforced without slowing down development, aligning with Green Pace’s commitment to secure, efficient software delivery.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Likely | Medium | High | 4 |
| STD-003-CPP | High | Likely | Low | High | 5 |
| STD-004-CPP | High | Likely | Medium | High | 5 |
| STD-005-CPP | High | Likely | High | Critical | 5 |
| STD-006-CPP | Medium | Unlikely | Low | Low | 1 |
| STD-007-CPP | Medium | Unlikely | Low | Medium | 2 |
| STD-008-CPP | High | Possible | Medium | High | 4 |
| STD-009-CPP | Medium | Possible | Low | Medium | 3 |
| STD-010-CPP | Medium | Possible | Medium | Medium | 3 |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |  |

### 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 ensures that all sensitive data is encrypted using AES-256 or higher regardless if its physical or virtual media. |
| Encryption in flight | Encryption in flight secures data during transmission over networks. This includes HTTPS (TLS 1.2 or above), SSH, or VPN tunnels. This Encryption renders any intercepted data virtually useless to attackers. |
| Encryption in use | Encryption in use protects data currently being processed in memory. Techniques include hardware-based enclaves (e.g., Intel SGX), virtual machine isolation, or application-level encryption. This can protect data even if it is actively being accessed and used. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication ensures only privileged users can access certain systems or data. This policy often requires MFA for all users at every level of access especially at higher levels. MFA for all users, integration with enterprise identity providers (using OAuth2.0 or SAML), and secure password storage (e.g., bcrypt). |
| Authorization | Authorization determines what authenticated users can access. Role-based access control (RBAC) is enforced, where each user is granted minimum required privileges (least privilege). This includes restricting file access, user account changes, and database modifications based on assigned roles. |
| Accounting | Accounting can be a great tool in retrospective to see if any users have been showing signs of suspicious behavior. Logging all user actions is vital to ensuring the integrity of important systems. Logs must be stored securely and reviewed regularly. This ensures auditability and helps identify malicious activity or policy violations. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 6/18/25 | Finished Security Policy List | James Galbreath | [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 |