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



# Green Pace Secure Development Policy

# CHASE CARTER

## 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 from users and the other sources should never be relied on. Checking the data makes sure that it is in the correct outputs. This prevents risk from the inputs. |
| 1. Heed Compiler Warnings | Compiler warnings show weaknesses. Fixing all the warnings improves the safety and correctness. |
| 1. Architect and Design for Security Policies | Security should be planned and designed from the beginning of the development process. This makes sure that the system limits risk. |
| 1. Keep It Simple | Complex systems can make hidden weaknesses. Simple, well-documented code is easier to manage and maintain securely. |
| 1. Default Deny | Systems should stop all access by default and only allow explicitly permitted actions. This limits potential problems from a foreseen issue. |
| 1. Adhere to the Principle of Least Privilege | Every module, process, or user should use the minimum privileges necessary. This reduces the impact of exploits. |
| 1. Sanitize Data Sent to Other Systems | Outgoing data should be verified so that injection attacks or information leakage can be left at a minimum to the systems. |
| 1. Practice Defense in Depth | Layered security measures help make sure that failure in one area does not compromise the entire system. |
| 1. Use Effective Quality Assurance Techniques | Using unit testing, static analysis, peer reviews, and automated tools improves reliability and helps catch security issues early. |
| 1. Adopt a Secure Coding Standard | Using a recognized standard like SEI CERT C++ ensures consistent, safe practices across the development team, reducing the chance of vulnerabilities. |

### 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 | Using the right data types makes sure that the memory is used efficiently and prevents overflow/underflow vulnerabilities. |

| **Noncompliant Code** |
| --- |
| When the code uses an unsigned int it will have errors. |
| unsigned int balance = -50;  printf("Balance: %u\n", balance); |

| **Compliant Code** |
| --- |
| Uses a signed int the right way to represent negative values. |
| int balance = -50;  printf("Balance: %d\n", balance); |

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

| **Principles(s):**  Keep It Simple (Principle 4): Using the smallest adequate data type reduces complexity and attack surface.  Least Privilege (Principle 6): Choosing a type that only allows permitted values limits potential misuse. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | |  | | --- | |  |  |  | | --- | | 2.6 | | integerOverflow | Flags potential integer overflow and underflow errors |
| SonarSource C/C++ plugin | 8.9 | EXP34-C | Ensures use of appropriate signed/unsigned types to prevent under/overflow |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Checking the data to make sure that operations are performed on safe and expected values, lowering risks. |

| **Noncompliant Code** |
| --- |
| Does not check the value before dividing. |
| int x = 10, y = 0;  printf("Result: %d", x / y); |

| **Compliant Code** |
| --- |
| Adds validation to prevent division by zero. |
| int x = 10, y = 0;  if (y != 0) printf("Result: %d", x / y);  else printf("Invalid divisor\n"); |

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

| **Principles(s):**  Validate Input Data (Principle 1): Always verify values before use.  Defense in Depth (Principle 8): Multiple checks reduce single-point failures. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarSource C/C++ plugin | 8.9 | SIDD16-C | Finds divisions by zero and missing precondition checks |
| Cppcheck | 2.6 | divisionByZero | Shows when a divisor may be zero |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Wrong string handling makes the buffer overflows. Safe functions should be used instead. |

| **Noncompliant Code** |
| --- |
| Uses unsafe strcpy without bounds checking. |
| char dest[10];  strcpy(dest, "This is too long"); |

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

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

| **Principles(s):**  Validate Input Data (Principle 1): Verify string lengths and contents.  Quality Assurance (Principle 9): Use safe string APIs to prevent buffer overruns. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | |  | | --- | |  |  |  | | --- | | 2.6 | | bufferOverflow | Detects potential buffer overflows in string handling |
| SonarSource C/C++ plugin | 8.9 | S3527 | Flags unsafe string copy functions without bounds checks |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Stops SQL injection protects against unauthorized access to databases. |

| **Noncompliant Code** |
| --- |
| User input is concatenated directly into a SQL string. |
| string query = "SELECT \* FROM users WHERE name='" + input + "';"; |

| **Compliant Code** |
| --- |
| Uses parameterized statements to prevent SQL injection. |
| sqlite3\_stmt\* stmt;  sqlite3\_prepare\_v2(db, "SELECT \* FROM users WHERE name=?", -1, &stmt, NULL);  sqlite3\_bind\_text(stmt, 1, input.c\_str(), -1, SQLITE\_STATIC); |

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

| **Principles(s):**  Validate Input Data (Principle 1): Reject or parameterize untrusted input.  Defense in Depth (Principle 8): Database layer must enforce its own checks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarSource C/C++ plugin | 8.9 | S3643 | Detects SQL string concatenation without parameter binding |
| Cppcheck | 2.6 | sqlInjection | Warns when SQL is composed from untrusted data |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Good memory management stops leaks and problems like buffer overflows or use-after-free. |

| **Noncompliant Code** |
| --- |
| Fails to free dynamically allocated memory. |
| int\* data = new int[100]; |

| **Compliant Code** |
| --- |
| Deletes memory to avoid leaks. |
| int\* data = new int[100];  delete[] data; |

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

| **Principles(s):**  Keep It Simple (Principle 4): Explicit allocation/deallocation reduces surprises.  Defense in Depth (Principle 8): Proper memory hygiene prevents exploitation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.6 | memleak | Finds memory leaks when pointers go out of scope |
| SonarSource C/C++ plugin | 8.9 | S3518 | |  | | --- | |  |  |  | | --- | | Reports missing deallocation of heap memory | |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions catch logic errors in development and help document problems. |

| **Noncompliant Code** |
| --- |
| Does not check expected input. |
| int divide(int a, int b) {  return a / b;  } |

| **Compliant Code** |
| --- |
| Adds checks to make sure b is not zero. |
| #include <cassert>  int divide(int a, int b) {  assert(b != 0);  return a / b;  } |

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

| **Principles(s):**  Quality Assurance (Principle 9): Use assertions to catch developer errors early.  Keep It Simple (Principle 4): Fail-fast on unexpected states. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarSource C/C++ plugin | 8.9 | |  | | --- | |  |  |  | | --- | | S3000 | | |  | | --- | |  |  |  | | --- | | Encourages use of assert() for  invariant checks | |
| Cppcheck | |  | | --- | |  |  |  | | --- | | 2.6 | | assertWithoutInclude | Warns when assert is used without including <cassert> |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Exception safety makes sure errors are handled and prevents program crashes. |

| **Noncompliant Code** |
| --- |
| No try-catch block around risky operation. |
| [int num = std::stoi("abc"); |

| **Compliant Code** |
| --- |
| Wraps conversion in a try-catch block. |
| [try {  int num = std::stoi("abc");  } catch (const std::invalid\_argument& e) {  std::cerr << "Invalid input\n";  } |

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

| **Principles(s):**  Defense in Depth (Principle 8): Catch and handle exceptions at all layers.  Keep It Simple (Principle 4): Avoid uncaught exceptions that crash applications. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarSource C/C++ plugin | 8.9 | S125 | Flags functions that throw but don’t catch exceptions |
| Cppcheck | 2.6 | exceptionThrownAndNotCaught | Detects thrown exceptions that are not caught |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Pointer Initialization | STD-008-CPP | Starting the pointers can make the code not act accordingly. Always start pointers before use. |

| **Noncompliant Code** |
| --- |
| Uses pointer that does not start. |
| int\* ptr;  \*ptr = 42; |

| **Compliant Code** |
| --- |
| Starts pointer before use. |
| int value = 42;  int\* ptr = &value; |

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

| **Principles(s):**  Validate Input Data (Principle 1): Uninitialized pointers are untrusted.  Keep It Simple (Principle 4): Always initialize to a known value. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarSource C/C++ plugin | 8.9 | S4528 | Reports use of uninitialized pointers |
| Cppcheck | 2.6 | uninitvar | Flags variables (including pointers) used before initialization |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Lock Management | STD-009-CPP | Not releasing locks can cause deadlocks or to stop all resources. |

| **Noncompliant Code** |
| --- |
| Lock will not release because early return. |
| std::mutex m;  m.lock();  if (someCondition) return;  m.unlock(); |

| **Compliant Code** |
| --- |
| Uses RAII to make sure lock will always released. |
| std::mutex m;  std::lock\_guard<std::mutex> guard(m);  if (someCondition) return; |

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

| **Principles(s):**  Defense in Depth (Principle 8): Ensure locks are always released, even on error.  Quality Assurance (Principle 9): Use RAII to guarantee correct cleanup. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarSource C/C++ plugin | 8.9 | S297 | Flags manual lock()/unlock() calls that could leak on early exit |
| Cppcheck | 2.6 | lockManagement | Detects mismatched lock/unlock usage |

**Coding Standard 10**

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Loop Bounds | STD-010-CPP | Wrong loop code can lead to buffer overflows or infinite loops. |

| **Noncompliant Code** |
| --- |
| Accesses out-of-bounds array index. |
| int arr[5];  for (int i = 0; i <= 5; i++) arr[i] = i; |

| **Compliant Code** |
| --- |
| Make sure loop bounds stay within array size. |
| int arr[5];  for (int i = 0; i < 5; i++) arr[i] = i; |

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

| **Principles(s):**  Validate Input Data (Principle 1): Prevent out-of-range indexing.  Defense in Depth (Principle 8): Multiple checks against buffer overruns. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarSource C/C++ plugin | 8.9 | S4386 | Flags array indexing that may go out of bounds |
| Cppcheck | 2.6 | arrayIndexOutOfBounds | Detects loops or index operations that exceed container limits |

### 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.

To make these rules automatically work we will add code-checking tools at key points in our development process. First in the Create phase each developer’s code editor will have Cppcheck and SonarLint installed. These tools will catch problems before any code is saved. Next in the Verify phase SonarQube will scan every code change. The build will stop if it finds any serious issues.

In Pre-Production we will run fuzz tests in a safe test area. These tests will find buffer overflow errors and SQL injection issues. At Release and Prevent we will require a clean scan before signing and publishing the software. We will also include a list of all software parts and scripts that check file safety. Finally, in Detect and Respond tools that protect the running app and tools that watch user and system activity will look for attacks. Any findings will feed back into our risk planning so we can keep improving our defenses.

### 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 | Possible | Low | High | 4 |
| STD-003-CPP | High | Likely | Medium | High | 5 |
| STD-004-CPP | High | Likely | Medium | High | 5 |
| STD-005-CPP | Medium | Possible | Medium | Medium | 3 |
| STD-006-CPP | Low | Unlikely | Low | Low | 1 |
| STD-007-CPP | Medium | Possible | Medium | Medium | 3 |
| STD-008-CPP | Medium | Possible | Low | Medium | 2 |
| STD-009-CPP | High | Likely | Low | High | 4 |
| STD-010-CPP | 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.** |
| --- | --- |
| Rest | |  | | --- | |  |  |  | | --- | | All sensitive data stored in databases or file systems. All must be encrypted using  AES-256 to protect data. | |
| Flight | All network communications with information must use TLS 1.2 or higher to defend against attacks. |
| Use | For highly sensitive processing use hardware-backed memory encryption like intel SGX to guard against memory-scraping. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Require multi-factor authentication for all user and service logins. Verify and identity before giving any access. |
| Authorization | Role-based access control (RBAC) and least privilege. Make sure users and services can only perform explicitly approved actions. |
| Accounting | Allow and retain audit logs of user logins, database changes, new user additions, access levels, and file accesses. Ship logs to a SIEM for analysis. |

**\***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. STD-001-CPP 4, 6 Simple minimal types make code easier to read. Using the smallest type keeps the attack surface small.
2. STD-002-CPP 1, 8 Checking values stops wrong data from getting in. Multiple checks add extra layers of defense.
3. STD-003-CPP 1, 9 We check string lengths before using them. We use safe string functions to avoid buffer errors.
4. STD-004-CPP 1, 8 We reject any SQL built from raw input. We also enforce checks on the database side.
5. STD-005-CPP 4, 8 Manually managing memory keeps code clear. Extra checks help catch leaks and buffer attacks.
6. STD-006-CPP 4, 9 Asserting key conditions stops bugs early. This makes code simpler and more reliable.
7. STD-007-CPP 4, 8 We handle exceptions in a central place. This prevents crashes and makes flow easier to follow.
8. STD-008-CPP 1, 4 We start pointers before use. This avoids crashes from undefined memory.
9. STD-009-CPP 8, 9 RAII locks manage resources safely. This ensures tests catch all lock errors.
10. STD-010-CPP 1, 8 We check loop before running them. Extra checks stop out-of-range errors.

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 | 05/25/2025 | Initial Template | Chase Carter |  |
| 1.1 | 06/10/2025 | Completed coding standards, risk assessments, automation, encryption, Triple-A | Chase Carter |  |
|  |  |  |  |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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