**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 | Ensuring that all input is properly validated helps prevent various forms of injection attacks, such as SQL or XSS. Input validation involves checking that the data meets specific criteria before processing it, ensuring that it conforms to expected formats or constraints. |
| 1. Heed Compiler Warnings | Compiler warnings will often indicate potential issues in code that could lead to vulnerabilities. Paying attention to and addressing these concerns can help catch and resolve issues early in the development process. |
| 1. Architect and Design for Security Policies | Security should be integrated into the design and architecture phases of software development. This involves defining security requirements, creating models, and designing systems that enforce these policies. By integrating security considerations from the start of development, systems can be more resilient against attacks. |
| 1. Keep It Simple | Simple designs are easier to understand, test, and secure. Complexity can introduce more potential for vulnerabilities, and by keeping the design straightforward, it can be easier to spot and fix security concerns. |
| 1. Default Deny | This principle ensures that access to resources is denied by default and only granted when necessary. This reduces the attacker’s ability to get into systems that they shouldn’t have access to by guarding sensitive resources. |
| 1. Adhere to the Principle of Least Privilege | Giving users and systems the minimal amount of access necessary to perform their functions reduces the potential damage that a compromised account can wreak on the system at large. By limiting privileges, a hacker’s ability is significantly constrained. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data ensures that data shared with outside systems does not contain malicious content that could exploit vulnerabilities. This means that cleaning and validating data to ensure it adheres to expected formats and doesn’t contain harmful code before sending to other systems is crucial. |
| 1. Practice Defense in Depth | DiD involves implementing multi-layered security controls and defenses to protect systems. This approach means that even if one defense mechanism is bypassed, others remain to provide protection. |
| 1. Use Effective Quality Assurance Techniques | Using rigorous QA techniques such as code reviews, automated testing, and static analysis helps identify and address security vulnerabilities early into the development process. Ensuring high-quality code reduces the likelihood of security flaws and adds to the overall security of the application. |
| 1. Adopt a Secure Coding Standard | Following a secure standard provides guidelines and best practices for writing secure code. These standards help developers avoid common pitfalls and vulnerabilities. This ensures that the codebase is strong against common types of attacks and adheres to established security principles. |

### 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** | **Use appropriate data types** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Using appropriate data types ensures that the program uses memory efficiently and works as intended. |

| **Noncompliant Code** |
| --- |
| An int variable is used to store a floating-point value leading to potential data loss |
| Int age = 25.5; |

| **Compliant Code** |
| --- |
| Using the correct data types prevents loss of data and ensures proper and intended functionality. |
| Float age = 25.5 |

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

| **Principles(s):** **Ensure Data Integrity and Memory Efficiency:** This principle maps to the standard by emphasizing the importance of using the correct data types for the job to avoid data loss and optimize memory usage. This helps prevent unexpected behavior and ensure that data is stored accurately, and resources are allocated efficiently. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.3 | RSPEC-1199 | Detects inappropriate use of data types in C++ code. |
| Cppcheck | 2.7 | TypeMismatch | Detects mismatched data types and potential data loss issues. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Validate data values** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Validating data ensures that variables hold expected and valid values preventing unexpected behavior. |

| **Noncompliant Code** |
| --- |
| The variable is assigned an invalid value, leading to potential logic errors or undefined behavior. |
| Int age = -5  If (age < 0) {  Code continues without proper handling  } |

| **Compliant Code** |
| --- |
| Age is reassigned with a valid value, ensuring the code functions properly. |
| Int age = -5;  If (age < 0) {  Age = 0;  //code  } |

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

| **Principles(s):** **Ensure Data Integrity and Code Readability:** This principle maps to this standard by emphasizing the importance of data validation to avoid logic errors and undefined behavior. Proper validation helps maintain accuracy and ensures the program operates as intended. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.3 | RSPEC-2757 | Detects missing validation checks in C++ code |
| Cppcheck | 2.7 | ValueAnalysis | Detects missing validation checks |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Ensure String Correctness** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Ensuring string correctness prevents buffer overflow, data corruption, and other vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The string copied into buffer exceeds the allocated size, leading to overflow. |
| Char buffer[10];  Strcpy(buffer, “This is a very long string.”); |

| **Compliant Code** |
| --- |
| Using strncpy with limits ensures null-termination prevents overflow. |
| Char buffer[10];  Sttrncpy(buffer, “This is a very long string.”, sizeof(buffer) – 1);  Buffer[sizeof(buffer) - 1] = ‘\0’; |

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

| **Principles(s):** **Maintain Security and Stability:** The principle of maintaining security and stability maps to this standard by emphasizing the importance of managing strings correctly to avoid overflow and data corruption, which can lead to security and stability issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.3 | RSPEC-820 | Detects improper string handling in C++ code |
| Cppcheck | 2.7 | BufferOverrun | Detects improper string handling and buffer overflow issues |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevent SQL Injection** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Preventing SQL injection ensures user inputs cannot compromise database security by executing unintended queries. |

| **Noncompliant Code** |
| --- |
| Directly concatenating user input into queries can lead to injection attacks. |
| Std::string query = “SELECT \* FROM users WHERE username = ‘” + username + “’;”; |

| **Compliant Code** |
| --- |
| Using a prepared statement with parameterized queries prevents injection by handling inputs safely |
| Std::string query = “SELECT \* FROM users WHERE username = ?”;  Std::prepared stmt = conn.prepareStatement(query);  Stmt.setString(1, username); |

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

| **Principles(s):** **Ensure Database Security and Integrity:** This maps to our standard by highlighting the importance of preventing SQL injection attacks. The proper handling of inputs through parameterized queries and prepared statements helps protect from manipulation and unauthorized access. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.3 | RSPEC-3649 | Detects injection vulnerabilities |
| Cppcheck | 2.7 | SQLInjection | Detects potential SQL injection vulnerabilities and unsafe SQL query construction |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Ensure Memory Protection** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Proper memory protection prevents unauthorized access and memory corruption. |

| **Noncompliant Code** |
| --- |
| Deleting a pointer to an array using delete instead of delete[] can lead to corruption |
| Int\* prt = new int[10];  Delete ptr; |

| **Compliant Code** |
| --- |
| Using the proper function deallocates memory allotted for an array, thus preventing corruption |
| Int\* ptr = new int[10];  Delete[] ptr; |

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

| **Principles(s):** **Ensure System Stability and Security:** This maps on by showcasing how important memory management is in preventing corruption and unauthorized access. Proper deallocation helps maintain stability and security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.3 | RSPEC-4036 | Detects improper memory management |
| Cppcheck | 2.7 | MemoryManagement | Detects improper management and potential corruption issues |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Assertions for Debugging** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions can help catch logic errors and invalid assumptions during development |

| **Noncompliant Code** |
| --- |
| The code assumes ptr is not null without checking, which could lead to a null pointer dereference |
| Void process(int\* ptr){  //assumes not null  \*ptr = 32;  } |

| **Compliant Code** |
| --- |
| Using assert ensures that ptr is not null during development, potentially catching errors early on. |
| #include <cassert>  Void process(int\* ptr){  Assert(ptr != nullptr);  \*ptr = 42;  } |

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

| **Principles(s):** **Ensure Code Reliability and Robustness:** Using assertions allows us to catch logic errors and invalid assumptions early in development. This helps identify and fix issues before they manifest in production. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.3 | RSPEC-3626 | Detects missing assertions |
| Cppcheck | 2.7 | Assertions | Detects missing assertions and potential logic errors |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Proper Exception Handling** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Proper exception handling ensures errors are managed gracefully, preventing crashes |

| **Noncompliant Code** |
| --- |
| Catching all exceptions without proper handling can hide errors and make debugging difficult |
| Try{  //code that may throw an exception  } catch (…) {  //catches all exceptions  } |

| **Compliant Code** |
| --- |
| Catching specific exceptions and logging error messages makes for easier debugging |
| Try {  //code that may throw exception  } catch (const std::exception& e) {  Std::cerr << “Error: ” << e.what() << std::endl;  } |

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

| **Principles(s):** **Ensure Error Resilience and Debuggability:** Allows for graceful error management, preventing crashes, and making debugging easier by providing detailed error information. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.3 | RSPEC-4790 | Detects improper exception handling |
| Cppcheck | 2.7 | ExceptionSafety | Detects improper handling and potential issues with exception management |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Use Smart Pointers** |
| --- | --- | --- |
| **Pointers** | [STD-008-CPP] | Using smart pointers automatically manages the lifetime of objects, preventing memory leaks and dangling pointers. |

| **Noncompliant Code** |
| --- |
| Using raw pointers requires manual memory management, leading to potential memory leaks. |
| Int\* ptr = new int(5);  Delete ptr; |

| **Compliant Code** |
| --- |
| Using smart pointers ensures that memory is automatically managed and released when the pointer leaves scope |
| Std::unique\_ptr<int> ptr = std::make\_unique<int>(5); |

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

| **Principles(s):** **Ensure Memory Safety and Resource Management:** Prevents memory leaks and dangling pointers, ensuring that resources and released when no longer needed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.3 | RSPEC-4699 | Detects the use of raw pointers |
| Cppcheck | 2.7 | SmartPointerUsage | Detects raw pointer usage and suggests smart pointers |

#### 

#### Coding Standard 9

| **Coding Standard** | **Label** | **Use nullptr instead of NULL or 0** |
| --- | --- | --- |
| **Null Pointer** | [STD-009-CPP] | Using nullptr makes pointer initialization and comparison more type-safe avoiding ambiguity and errors |

| **Noncompliant Code** |
| --- |
| Using NULL or 0 can lead to ambiguity and type safety issues |
| Int\* ptr = NULL;  If (ptr == 0) {  //handle null pointer  } |

| **Compliant Code** |
| --- |
| Using nullptr improves type safety and clarity |
| Int\* ptr = nullptr;  If (ptr == nullptr) {  //handle null pointer  } |

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

| **Principles(s):** **Ensure Type Safety and Code Clarity:** Avoids the ambiguity associated with NULL or 0, leading to more robust and maintainable code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.3 | RSPEC-1787 | Detects the use of NULL or 0 in code and suggests nullptr |
| Cppcheck | 2.7 | UseNullptr | Detects use of NULL or 0 and suggest nullptr |

#### 

#### Coding Standard 10

| **Coding Standard** | **Label** | **Prefer Range-Based For Loops** |
| --- | --- | --- |
| **For Loops** | [STD-010-CPP] | Range-based for loops simplify iteration over collections, reducing potential errors over traditional for loops. |

| **Noncompliant Code** |
| --- |
| Using a traditional for loop involves manual index management, which is error prone |
| Std::vector<int> values = {1, 2, 3, 4, 5};  For (size\_t I = 0; I < values.size(); i++){  Std::cout << values[i] << std::endl;  } |

| **Compliant Code** |
| --- |
| Using a range based for loop removes the need for index management, limiting potential human error |
| Std::vector<int> values = {1, 2, 3, 4, 5};  For (int value : values){  Std::cout << value << std::endl;  } |

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

| **Principles(s):** **Simplify Code & Reduce Errors:** Streamlines iteration and minimizes manual index management, leading to less errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.3 | RSPEC-3626 | Detects traditional for loops and suggests range-based loops |
| Cppcheck | 2.7 | RangeBasedForLoop | Detects traditional for loops and suggests range-based loops |

### 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 should be integrated into the lifecycle to enforce coding standards and ensure compliance. In the pre-production phase, we can use tools such as SonarQube and Cppcheck for constant vulnerability scanning and use static analysis during the design and build stages. Automation of vulnerability scanning and comprehensive testing is crucial to catching issues early. During the transition to production, we should also automate security configs and penetration testing. On top of this, we can implement automated log collection and intrusion detection for constant monitoring and real-times alerts. Developing automated responses will allow us to handle incidents quickly, as well as block attacks and implement rollbacks for a compromised service. Regular assessments and stability checks of the system against security baselines allows us to maintain a secure and compliant environment.

### 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 | Medium | Low | High | 4 |
| STD-002-CPP | High | High | Low | High | 5 |
| STD-003-CPP | High | Medium | Medium | High | 4 |
| STD-004-CPP | Critical | High | Medium | Critical | 5 |
| STD-005-CPP | High | Medium | Medium | High | 4 |
| STD-006-CPP | Medium | Medium | Low | Medium | 3 |
| STD-007-CPP | High | Medium | Medium | High | 4 |
| STD-008-CPP | High | Medium | Medium | High | 4 |
| STD-009-CPP | Medium | Medium | Low | Medium | 3 |
| STD-010-CPP | Medium | Medium | Low | Medium | 3 |

### 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 of stored data on disks, databases, or any storage media. Uses technologies like AES to secure. This policy protects sensitive data from being accessed if the storage media is stolen or accessed illegitimately. |
| Encryption in flight | This refers to the encryption of data while it is being transmitted over networks. Common protocols such as TLS/SSL are used to secure communications. This ensures data privacy and integrity during transmission, protecting from interception and eavesdropping. |
| Encryption in use | Encryption of data while it is being processed or used in memory. Techniques include homomorphic encryption and secure enclaves. This ensures privacy during processing, preventing exposure to unauthorized users. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Verifying the identity of users and devices before granting access. Methods may include passwords, biometrics, and 2FA. This ensures that only authorized users or devices can access sensitive data. |
| Authorization | Determines the permissions and access levels of an authenticated user or device. Typically uses role-based access control or access control lists. |
| Accounting | Tracking and logging user and device activities. Utilizes logging and audit trails. Provides a record of access and actions allowing for monitoring, compliance, and analysis. Applied constantly to monitor usage, detect abnormalities, and ensure accountability. |

**\***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 |  |
| 1.1 | 08/07/2024 | Finished Document | Bryce Burroughs | [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 |