**Green Pace Developer: Security Policy**



# 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 | This principle is about making sure any data that comes into a system is checked before it’s used. For example, when users type something into a form, the program should confirm that the input is the right type, size, or format. Doing this helps stop problems like people trying to sneak in harmful code or unexpected commands. basically a way to keep the system safe by only accepting data that makes sense. |
| 1. Heed Compiler Warnings | When coding, the compiler will sometimes throw warnings that point out things in your code that might not be safe or could cause errors later. It’s a good idea not to just skip over these. Even if the program still runs, the warning is usually there for a reason. Fixing them early can save you from bugs or security problems down the line. |
| 1. Architect and Design for Security Policies | When building software, security needs to be part of the plan from the very start, not something added at the end. This means thinking about things like how users log in, how data is stored, and who should have access to what. By setting clear security rules and building the system around them, it’s easier to avoid holes that hackers could take advantage of later. |
| 1. Keep It Simple | In security, simple designs are usually better because they’re easier to understand, use, and protect. If a system is too complicated, it’s easier to make mistakes or overlook weaknesses. By keeping things straightforward, it’s easier to spot problems, maintain the system, and make sure security measures actually work the way they’re supposed to. |
| 1. Default Deny | Default deny means blocking access to everything unless it’s been specifically allowed. Instead of assuming something is safe and then blocking bad cases, the system should assume nothing is safe until it’s proven. This way, only trusted users, programs, or actions are given permission, which makes it harder for attackers to slip through. |
| 1. Adhere to the Principle of Least Privilege | This principle means giving users or programs only the access they actually need to do their job, and nothing more. For example, if someone just needs to read a file, they shouldn’t also have permission to edit or delete it. Limiting access like this reduces the damage that can happen if an account is hacked or a mistake is made. |
| 1. Sanitize Data Sent to Other Systems | [Before sending data from one system to another, it should be cleaned and checked so it doesn’t carry anything harmful. For example, if your program passes data to a database or another application, you want to make sure no extra commands or malicious code are hidden in it. Sanitizing helps prevent attacks like SQL injection or cross-site scripting and keeps the receiving system safe. |
| 1. Practice Defense in Depth | Defense in depth means using multiple layers of security instead of relying on just one. If one layer fails, there are still others in place to protect the system. For example, even if a hacker gets past a firewall, they might still run into strong passwords, encryption, or monitoring tools. The idea is that stacking protections makes it much harder for an attack to succeed. |
| 1. Use Effective Quality Assurance Techniques | This principle is about testing your software thoroughly to catch problems before they become security risks. That includes things like code reviews, automated tests, and manual testing to make sure the program works as expected. By finding and fixing mistakes early, you can prevent bugs and vulnerabilities that attackers might try to exploit. |
| 1. Adopt a Secure Coding Standard | Using a secure coding standard means following a set of rules or best practices when writing code to reduce security risks. These standards guide things like how to handle input, manage errors, and store data safely. By sticking to them, developers are less likely to introduce vulnerabilities, and the code is easier for others to understand and maintain securely. |

### 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** | **Proper Data Types** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Using the correct data type ensures values are stored efficiently and reduces errors like overflows or unexpected behavior. |

| **Noncompliant Code** |
| --- |
| Using a float for a counter can cause rounding errors. |
| float counter = 0;  counter++; |

| **Compliant Code** |
| --- |
| Using an int avoids rounding issues and is safer for counting |
| int counter = 0;  counter++; |

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

| **Principles(s):** Validate Input Data, Adopt a Secure Coding Standard - These principles ensure code safety by enforcing type safety, validation, and consistency with secure coding practices to prevent injection or overflow vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| high | medium | medium | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.13 | CERT C++ Rules | Detects violations of secure coding standards and memory or input handling vulnerabilities automatically during CI builds. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Validate Input Data** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Checking input ensures only valid data enters the system, preventing errors or malicious data from causing issues. |

| **Noncompliant Code** |
| --- |
| Accepts user input without checking its validity. |
| int age;  std::cin >> age; |

| **Compliant Code** |
| --- |
| Input is validated to be within a reasonable range. |
| int age;  std::cin >> age;  if (age < 0 || age > 120) {  std::cout << "Invalid age entered.\n";  } |

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

| **Principles(s):** Validate Input Data, Heed Compiler Warnings, Adopt a Secure Coding - These principles ensure code safety by enforcing type safety, validation, and consistency with secure coding practices to prevent injection or overflow vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| high | medium | medium | high | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.13 | CERT C++ Rules | Detects violations of secure coding standards and memory or input handling vulnerabilities automatically during CI builds. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Safe String Handling** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Using unsafe string functions can lead to buffer overflows or memory corruption. |

| **Noncompliant Code** |
| --- |
| Using strcpy without checking string length can overflow the buffer. |
| char buffer[10];  strcpy(buffer, userInput); |

| **Compliant Code** |
| --- |
| Using strncpy with size limit prevents buffer overflow. |
| char buffer[10];  strncpy(buffer, userInput, sizeof(buffer) - 1);  buffer[9] = '\0'; |

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

| **Principles(s):** Validate Input Data, Heed Compiler Warnings, Adopt a Secure Coding Standard - These principles ensure code safety by enforcing type safety, validation, and consistency with secure coding practices to prevent injection or overflow vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| high | medium | medium | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Visual Studio Code Analyzer | 2022 | Rule STR31-C | Flags unsafe C functions like strcpy or sprintf and suggests safer alternatives (strncpy, snprintf). |
| Clang Static Analyzer | 22.0 | Rule STR31-C | Flags unsafe C functions like strcpy or sprintf and suggests safer alternatives (strncpy, snprintf). |

#### Coding Standard 4

| **Coding Standard** | **Label** | **SQL Injection Prevention** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Directly inserting user input into SQL queries can be exploited. Use parameterized queries to prevent attacks. |

| **Noncompliant Code** |
| --- |
| User input is inserted directly into the query. |
| std::string query = "SELECT \* FROM users WHERE username = '" + userInput + "'";  executeQuery(query); |

| **Compliant Code** |
| --- |
| Prepared statements treat input as data, not as code. |
| std::string query = "SELECT \* FROM users WHERE username = ?";  preparedStatement.setString(1, userInput);  preparedStatement.execute(); |

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

| **Principles(s):** Validate Input Data, Default Deny - SQL injection attacks exploit unchecked user input. Using parameterized queries or stored procedures ensures data is treated as values, not executable code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10 | CERT Rule IDS00-J | Detects dynamic query concatenation and flags unsafe query construction patterns. |
| Fortify | 23.2 | CERT Rule IDS00-J | Detects dynamic query concatenation and flags unsafe query construction patterns. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Avoid Dangling Pointers** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Using pointers after freeing memory can crash programs or allow attacks. |

| **Noncompliant Code** |
| --- |
| Pointer used after being deleted. |
| int\* ptr = new int(5);  delete ptr;  \*ptr = 10; |

| **Compliant Code** |
| --- |
| Pointer set to nullptr after deletion to prevent use. |
| int\* ptr = new int(5);  delete ptr;  ptr = nullptr; |

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

| **Principles(s):** Keep It Simple, Practice Defense in Depth - A dangling pointer points to deallocated memory, leading to crashes or data corruption. Nullifying pointers after deletion prevents accidental reuse and mitigates memory corruption risks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2023 | MEM35-C | Detects double frees, invalid memory accesses, and use after free issues. |
| Cppcheck | 2.13 | MEM35-C | Detects double frees, invalid memory accesses, and use after free issues. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Using Assertions for Critical Checks** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions catch programming errors early during development. |

| **Noncompliant Code** |
| --- |
| No checks on critical assumptions. |
| int value = getValue();  int result = 100 / value; |

| **Compliant Code** |
| --- |
| Assertion ensures safe values before using them. |
| int value = getValue();  assert(value != 0);  int result = 100 / value; |

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

| **Principles(s):** Use Effective Quality Assurance Techniques, Heed Compiler Warnings - Assertions are used to catch errors early during development by verifying assumptions about code. This improves reliability and ensures invalid states don’t progress to production. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 22.0 | Rule MSC30-C | Identifies missing assertions or misuse of assertions during testing phases. |
| Visual Studio Code Analysis | 2022 | Rule MSC30-C | Identifies missing assertions or misuse of assertions during testing phases. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Proper Exception Handling** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Catching exceptions properly prevents program crashes and handles errors safely. |

| **Noncompliant Code** |
| --- |
| Exceptions ignored or left unhandled. |
| void openFile() {  std::ifstream file("data.txt");  // no error handling  } |

| **Compliant Code** |
| --- |
| Exceptions are caught and handled. |
| void openFile() {  try {  std::ifstream file("data.txt");  if (!file) throw std::runtime\_error("File not found");  } catch (const std::exception& e) {  std::cout << e.what() << "\n";  }  } |

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

| **Principles(s):** Architect and Design for Security, Adhere to Least Privilege - Handling exceptions correctly prevents crashes and unpredictable behavior. Structured error handling allows safe recovery and avoids leaking sensitive information in error messages. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10 | Rule ERR33-C | Finds empty or missing catch blocks and verifies exceptions are handled safely. |
| Coverity | 2023 | Rule ERR33-C | Finds empty or missing catch blocks and verifies exceptions are handled safely. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Safe Pointer Use** |
| --- | --- | --- |
| Pointer Safety | STD-008-CPP | Uninitialized or null pointers can crash a program or create security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Pointer used without initialization, which can cause undefined behavior. |
| cpp int\* ptr; \*ptr = 10; |

| **Compliant Code** |
| --- |
| Pointer initialized before use to ensure safety. |
| cpp int value = 10; int\* ptr = &value; \*ptr = 20; |

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

| **Principles(s):** Keep It Simple, Practice Defense in Depth - Using uninitialized or null pointers causes undefined behavior and can lead to privilege escalation or system crashes. Always initialize pointers and validate them before use. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2023 | MEM30-C | Scans pointer usage patterns to ensure safe initialization and access. |
| Clang Static Analyzer | 22.0 | MEM30-C | Scans pointer usage patterns to ensure safe initialization and access. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Validate External Input** |
| --- | --- | --- |
| Input Validation | STD-009-CPP | Any input from users, files, or network sources should be checked to make sure it’s safe and expected. This prevents crashes, invalid data, and attacks like code injection. |

| **Noncompliant Code** |
| --- |
| Reads a number from the user without checking its type. |
| cpp int num; std::cin >> num; |

| **Compliant Code** |
| --- |
| Checks that input is a valid integer and handles invalid input. |
| cpp int num; if (!(std::cin >> num)) { std::cout << "Invalid input\n"; std::cin.clear(); std::cin.ignore(1000, '\n'); } |

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

| **Principles(s):** Validate Input, Sanitize Data - External inputs from files, APIs, or networks can contain malformed or malicious data. Proper validation prevents injection attacks and data corruption. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10 | FIO30-C | Detects missing input validation and unsafe conversions for external data sources. |
| Cppcheck | 2.13 | FIO30-C | Detects missing input validation and unsafe conversions for external data sources. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Secure Logging** |
| --- | --- | --- |
| Logging | STD-010-CPP | Logging sensitive information, like passwords or personal data, can create security risks. Logs should only contain information safe for public or internal review. |

| **Noncompliant Code** |
| --- |
| Logs user passwords directly, which can be stolen or exposed. |
| cpp std::cout << "User password: " << password << "\n"; |

| **Compliant Code** |
| --- |
| Logs only safe information, avoiding sensitive data. |
| cpp std::cout << "User logged in successfully\n"; |

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

| **Principles(s):** Default Deny, Adopt a Secure Coding Standard- Avoid logging sensitive data such as passwords or tokens. Logs should contain only safe diagnostic or audit information to prevent data leaks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10 | CERT Rule MSC32-C | Scans log statements to ensure no sensitive data is exposed. |
| Splunk Enterprise Security | 8.2.3 | CERT Rule MSC32-C | Scans log statements to ensure no sensitive data is exposed. |

### 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.

[Insert your written explanations here.]

### 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 | Medium | High | 2 |
| STD-002-CPP | High | Medium | Medium | High | 2 |
| STD-003-CPP | High | Medium | High | High | 3 |
| STD-004-CPP | Critical | High | Medium | Critical | 5 |
| STD-005-CPP | High | Medium | High | High | 3 |
| STD-006-CPP | Medium | Low | Low | Medium | 2 |
| STD-007-CPP | Medium | Medium | Medium | Medium | 3 |
| STD-008-CPP | High | Medium | Medium | High | 3 |
| STD-009-CPP | High | High | Medium | High | 4 |
| STD-010-CPP | Medium | Low | Low | Medium | 2 |

### 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 | This protects data that’s stored on hard drives or databases. All sensitive info like passwords, financial data, and records should be encrypted using AES-256 or another strong method. It keeps data safe even if someone gets physical access to the device or server. This applies any time data is stored or backed up. |
| Encryption in flight | This protects data that’s being sent across a network. Using secure methods like TLS 1.3 or HTTPS makes sure no one can read or change the data while it’s moving between systems. This applies to emails, web traffic, and API calls to keep private info safe while it’s being transferred. |
| Encryption in use | This protects data that’s being used in memory while a program is running. Using secure hardware features or system encryption keeps attackers from reading info in RAM. It applies to servers and apps that handle sensitive stuff like passwords or credit card numbers. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | This checks who’s trying to get into a system. Everyone must log in using strong passwords and multi factor authentication (MFA) to prove who they are. It helps stop hackers or outsiders from pretending to be real users. |
| Authorization | This controls what someone can do after they log in. Users should only have access to what they actually need for their job. Using roles and permissions helps prevent mistakes or abuse. This rule applies to all systems and databases. |
| Accounting | This means tracking what users do in the system. Logs should record logins, changes, and access to data so security teams can review them later if something goes wrong. It helps with investigations and shows who did what. |

**\***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 | 10/16/2025 | Project 1 Final | Ulises Almaguer | N/A |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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