**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 | To validate input data is to ensure that the data that’s being input matches what we’re expecting. We want to validate things like data type, format, and range. If we’re expecting a string, are we receiving a string for input? Is the date formatted the correct way? Is the data withing the expected dates? These are all questions that should be asked to validate data input. |
| 1. Heed Compiler Warnings | We don’t want unexpected outcomes when compiling our code. If the compiler warns of us inconsistencies or unexpected issues, we should listen. We want our logic to execute properly. We want our systems to run efficiently. |
| 1. Architect and Design for Security Policies | When writing our code, we must keep these security policies in mind. We must not ignore security flaws to meet a deadline. We must not design our application with known security flaws because it’s easier. We must strategically plan our projects to be withing our company's security policies and compliance. |
| 1. Keep It Simple | In order to ensure our systems can be easily understood, we must keep it simple! An overengineered system will be harder to collaborate on. An overly complex system can also be less secure in the long run. |
| 1. Default Deny | To stay in line with this policy is to deny all access to our system by default. We will only allow pre-approved traffic. This is to ensure minimal risk while working on our system. |
| 1. Adhere to the Principle of Least Privilege | To adhere to this principle, we will give any program, process, or user the minimum level of access or resources to complete their respective task. This both limits the amount of security risks and the impact those security risks can have. |
| 1. Sanitize Data Sent to Other Systems | We must remove any extraneous or potentially harmful data that is sent to other systems. If left in, this data could lead to possible attacks on our system or whatever system that receives this data. |
| 1. Practice Defense in Depth | We must have multiple layers of security. If one layer of defense is exploited or lost. Then, the other layers must remain intact to protect our system. |
| 1. Use Effective Quality Assurance Techniques | We must use techniques to ensure the quality of our code. These techniques include but are not limited to; all testing types (unit testing, integration testing, performance testing, etc.) code reviews, and continuous integration using tools such as Jenkins or CircleCI. |
| 1. Adopt a Secure Coding Standard | We must ALL adhere to these secure coding policies. This is the standard that is being set so that we can minimize risk. Minimizing risk from the start also ensures we are working efficiently because we won’t have to go back and fix our previously avoidable mistakes. |

### 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] | When being consistent with our data type usage, our projects will have a reduced number of errors. Our project will also perform better even if it does technically run with mismatched data types. |

| **Noncompliant Code** |
| --- |
| Here we are assigning the number one to the variable “num” as a character. Then we attempt to make an integer “num1” by adding the character “num” to the number 1. |
| char num = 1;  int num1 = num + 1; |

| **Compliant Code** |
| --- |
| Here we assign numbers to integers so we can do our simple arithmetic. |
| Int num = 1;  Int num1 = 1 + 1; |

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

| **Principles(s):** Validate input data and adhere to compiler warnings map to this standard. This is because we must ensure that the data types match what we’re expecting from the input and because the compiler with likely warn us if there are inconsistent data types. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.17 | type | Cppcheck type check looks for things like improper type conversions or improper type assignments. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | By using correct data values, we can ensure that our code is consistent. Consistency ensures the next person working on it will have an easy time making modifications. |

| **Noncompliant Code** |
| --- |
| Since a month number will be a whole number, making the variable a float could lead to inconsistencies. |
| Float monthNum = 12; |

| **Compliant Code** |
| --- |
| We assign month number to an integer since it will always be a whole number. |
| Int monthNum = 12; |

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

| **Principles(s):** Validate input data and adhere to compiler warnings map to this standard. This is because we must ensure that the data values make sense for the variable based on the inputs and because the compiler with likely warn us if there are inconsistent data values. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.17 | type | Type can also ensure that data values align with the standard. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Proper strings will lead to a more secure and maintainable code. Improper string usage could lead to SQL injection or just difficult to read code. |

| **Noncompliant Code** |
| --- |
| Here is a buffer that is likely too small for someone’s last name. This can lead to overflow. |
| Char lastName[4]; |

| **Compliant Code** |
| --- |
| By just assigning “lastName” to a string that takes user input, we solve any potential overflow risks. |
| Std::string lastName;  Std::cin >> lastName; |

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

| **Principles(s):** Keep it simple maps to this standard. This is because we must not avoid overcomplicating our strings. There is no reason to get cute with taking characters as input. We must code securely so as to avoid overflow. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.17 | Buffer Access | Buffer access checks for improper buffer usage |
| cppcheck | 2.17 | String Hunting | String hunting checks for unsafe string manipulation |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | By following SQL injection standards, we can ensure that our program is free from this specific attack. Keeping our data safe. |

| **Noncompliant Code** |
| --- |
| This query is made directly using user input. SQL code could be injected into this by the password input being, “' OR '1'='1” This would make the query always true. |
| std::cin >> username;  std::cin >> password;  Std::string query = "SELECT \* FROM users WHERE username = '" + username + "' AND password = '" + password + "'" |

| **Compliant Code** |
| --- |
| By using placeholder symbols, we ensure that user input isn’t being used directly in the SQL query. This means we can bind the user input to those placeholders so that they won’t be part of the actual query. |
| Std::string query = "SELECT \* FROM users WHERE username = ! AND password = !" |

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

| **Principles(s):** Architect and design for security policies and use effective quality insurance techniques map to this standard. We must design our code in a way that isn’t vulnerable to SQL injection by adhering to our security policies. Then, we must test our code to make sure it isn’t vulnerable to SQL injection. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.17 | Unsafe Function | Unsafe function can help detect if a query is susceptible. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Improper memory protection will lead to things like loss of data, memory leak, or buffer overflows. |

| **Noncompliant Code** |
| --- |
| This can lead to memory leak because we are dynamically allocating new memory without clearing it. |
| Char\* lastName = new char[50];  std::cout << lastName << std::endl; |

| **Compliant Code** |
| --- |
| By using delete[] we deallocate the memory we assigned using new[] |
| Char\* lastName = new char[50];  std::cout << lastName << std::endl;  Delete[] lastName; |

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

| **Principles(s):** Adhere to principle of least privilege use effective quality assurance techniques map to this standard. We must make sure that there isn’t access to databases that things shouldn’t have. We then, must ensure that our code is tested for overflows and memory leaks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.17 | Memory Leak | Memory Leak can detect memory. However, there is lttle documentation on their site how it does this. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Properly using assertions means we are using them for debugging and not error handling. Assertions should also not modify the program. Assertions should be used to catch issues we have made in programming. This way our code is more correct and maintainable. |

| **Noncompliant Code** |
| --- |
| In this bad example we use an assertion to both check if the number of cars is positive and then make it zero if it is negative. |
| string numOfCars(int inventory) {  Assert(inventory >= 0);  If (inventory < 0) {  inventory = 0;  }  } |

| **Compliant Code** |
| --- |
| Here we are just checking to see if the number of cars is positive. |
| Int numOfCars(int inventory) {  Assert(inventory >= 0);  } |

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

| **Principles(s):** Use effective quality assurance techniques map to this standard. This is because if we aren’t properly using assertions then we aren’t using quality assurance techniques properly. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.17 | assert | Assert warns if there are unintended side effects of an assertion. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | By properly employing industry standards when it comes to exception handling, we ensure the code runs more predictably. We also help those down the line by placing mechanisms to where it is easier for them to manage the code we created. |

| **Noncompliant Code** |
| --- |
| Catch (...) will catch every exception. We want to try specific things so we can catch specific errors. |
| Class() {  try { anything()  }  catch (...) {  }  } |

| **Compliant Code** |
| --- |
| This is proper “try” and “catch” logic. We will try something. If there is an error, the exception will be caught. |
| Class() {  try {  }  catch () {  }  } |

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

| **Principles(s):** Use effective quality assurance techniques and validate input data map to this standard. This is because we must properly catch exceptions to have good quality assurance. Validating input should also be part of the exception handling process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.17 | Exception safety | Exception safety checks things like if we’re throwing exceptions in destructors or throwing an exception in a valid state. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Unit Testing | [STD-008-CPP] | To confirm that our code is working as intended, it is important to create good unit tests. Bad unit testing can lead to difficult to reuse code as well as slower debugging |

| **Noncompliant Code** |
| --- |
| We are jamming multiple assertions into one unit test. |
| Void testaddition() {  Assert(addition(1, 1) == 2);  Assert(subtraction(2, 1) == 1);  } |

| **Compliant Code** |
| --- |
| Multiple tests for multiple assertions. |
| Void testaddition) {  Assert(addition(1, 1) == 2);  }  Void testsubtraction() {  Assert(subtraction(2, 1) == 1);  } |

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

| **Principles(s):** Use effective quality assurance techniques maps to this standard. This is because we must have focused unit tests for what we want to accomplish. Multiple assertions may not prove an immediate threat to security, but they will make coding more difficult for the next person with unclear tests or results. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.17 | Exception safety | Exception safety can also check if an exception is caught via value. |
| Cppcheck | 2.17 | Check function usage | Check function usage can detect if a function usage is discouraged. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Naming Conventions | [STD-009-CPP] | Poor naming conventions can lead to misleading or confusing code. Improper naming conventions can also lead to improper use of functions because a programmer might’ve though a function did something else based off a bad name |

| **Noncompliant Code** |
| --- |
| What is user1 or user2? It is unclear what these strings are if they were in a DB. |
| Std::string user = JohnnyK;  Std::string user1 = 123456798;  Std::string user2 = Premium |

| **Compliant Code** |
| --- |
| Here we can see clearly what the strings mean. |
| Std::string username = JohnnyK;  Std::string userID = 123456798;  Std::string userStatus = Premium |

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

| **Principles(s):** The principle that maps to this standard is keep it simple. Although, the noncompliant example is simple, it actually overcomplicates things with how vague the variable names are. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.17 | string | String can detect the misuse of strings. I’m not sure this would help with naming conventions, though. |
| cppcheck | 2.17 | Unused var | Unused var checks for things like unused variables or unread variables |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Error Handling | [STD-010-CPP] | We must put mechanisms in place to ensure we know what is happening if our code doesn’t behave as expected. This way we can work to correct it. |

| **Noncompliant Code** |
| --- |
| Here, if the file doesn’t open then the programmer won’t know it’s not working. |
| Std::ifstream file(filename);  std::stringstream buffer;    buffer << file.rdbuf();  return buffer.str(); |

| **Compliant Code** |
| --- |
| If the file fails to open, then the programmer is alerted with an error. |
| if (!file) {  throw std::runtime\_error(“File: ” + filename + “, failed to open”);  }  Else {  Std::ifstream file(filename);  std::stringstream buffer;    buffer << file.rdbuf();  return buffer.str();  } |

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

| **Principles(s):** Use effective quality assurance techniques and validate input data map to this standard. This is because we are both validating input and effectively testing out code by throwing an exception if something unexpected occurs. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.17 | Exception safety | Exception safety can also help rethrow exceptions if they’re missed |

### 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.

An automation tool like cppcheck with its built-in checkers can be implemented in the pre-production phase. Specifically, in the verify and test phase, we can use assert and exception safety checks to help run our tests. We don’t necessarily have to modify the DevOps process. We need to add steps to already existing parts of the cycle. In build, we can use cppcheck’s bounds checking check to ensure that our arrays will remain in bounds and that we won’t experience buffer overflow. In design, we can use the class or function usage checks to ensure that our app is designed both securely and efficiently. In production, we can use continuous integration tools such as Jenkins or CircleCI to update our apps security and functionality.

### 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 | Unlikely | Medium | High | 2 |
| STD-003-CPP | High | Medium | Medium | High | 2 |
| STD-004-CPP | Critical | Medium | High | Very High | 4 |
| STD-005-CPP | Critical | Medium | High | Very High | 4 |
| STD-006-CPP | Medium | Medium | Low | Medium | 2 |
| STD-007-CPP | Medium | High | Medium | High | 2 |
| STD-008-CPP | Low | Medium | Low | Low | 1 |
| STD-009-CPP | Medium | Medium | Low | Medium | 1 |
| STD-010-CPP | Medium | Medium | Medium | Medium | 1 |

### 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 is when data that is stored physically is encrypted. This is to ensure that even if the device the data is stored on is compromised, the data won’t be. This is a physical part of defense in depth. A layer of added protection if a bad actor infiltrates a facility. |
| Encryption in flight | Encryption in flight is when data is encrypted while it’s being transferred over networks. While this is an added layer of defense versus social engineering and man in the middle attacks. Encryption in flight is also part of complying with laws that require certain sensitive data to be encrypted while it is in transit. |
| Encryption in use | Encryption in use refers to the process of encrypting data while it is being used. This encryption is important because if a device is compromised, it will help prevent all the data that is being worked on from being compromised. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies a user’s identity. It ensures a user’s login isn’t compromised. A compromised user’s login could access data they’re not supposed to or attempt to fish for more info from other authenticated users. This is also how we can add new users. If we’re able to properly authenticate who they are, that is. |
| Authorization | Authorization makes sure that a now authenticated user has the proper access to the databases they need and no more. By adhering to the principle of least privilege, we can ensure that a compromised login does minimal damage. |
| Accounting | Accounting is the process of tracking users once they are in the network. With proper accounting techniques we can see all of what a user is doing. We can see when they logged in. When they logged out. What they accessed or changed. This part of triple A acts as a reactive measure in a potential security breach. This is because we can audit all of the user’s actions to see if their account was compromised. |

**\***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 | 04/19/2025 | Updated all required sections for final project. | John Kangiser | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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