**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 | Always check and clean any data that comes into a system. This helps prevent attacks like SQL injections and cross-site scripting. Proper validation keeps the system safe and working correctly. |
| 1. Heed Compiler Warnings | Compiler warnings can point out possible security problems in the code. Ignoring them can lead to serious errors or weaknesses. Fixing them makes the code safer and more reliable. |
| 1. Architect and Design for Security Policies | Security should be part of the system’s design from the start. Using rules like least privilege and secure defaults makes systems harder to hack. A well-planned system keeps data safe and reduces risks. |
| 1. Keep It Simple | Complicated code is harder to protect and more likely to have mistakes. Simple, clear designs are easier to fix and keep secure. Less complexity means fewer security problems. |
| 1. Default Deny | The system should block access to everything unless permission is given. This helps prevent unauthorized people from getting sensitive data. Only trusted users should have access. |
| 1. Adhere to the Principle of Least Privilege | Users and programs should only get the minimum access they need. This stops hackers from gaining too much control if they break in. Limiting access keeps important data safer. |
| 1. Sanitize Data Sent to Other Systems | Any data sent to other systems should be cleaned and checked. This prevents dangerous attacks like data injections. Making sure data is safe protects the system from harm. |
| 1. Practice Defense in Depth | Using many layers of security makes it harder for hackers to break in. Firewalls, passwords, and encryption work together to protect data. If one layer fails, others still provide security. |
| 1. Use Effective Quality Assurance Techniques | Testing and reviewing code help find security problems before they cause harm. Catching and fixing issues early makes the system stronger. Regular security checks keep the software safe. |
| 1. Adopt a Secure Coding Standard | Using safe coding rules helps prevent common security problems. These rules guide developers in writing safer programs. Sticking to security standards reduces risks. |

### 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-DAT | Use appropriate data types for variables. |

| **Noncompliant Code** |
| --- |
| In this case a float number is initialized as an integer. |
| int var1 = 1.8; |

| **Compliant Code** |
| --- |
| Variable initialized as float. |
| float var1 = 1.8; |

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

| **Principles(s):** Data Integrity, ensures that the correct data type is used for variables. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.17.1 | Type Checking | Analyses C++ code for errors. |
| Clang-Tidy | 13.0.0 | Modernize Type | This tool checks for improper or inconsistent data types used in the code. |
| SonarQube | 9.5 | Type Issues | Identifies improper variables. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-DVL | Do not use object representations to compare floating-point values |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, memcmp() is used to compare two structures for equality. However, since the structure contains a floating-point object, this code may not behave as the programmer intended. |
| #include <stdbool.h>  #include <string.h>    struct S {    int i;    float f;  };    bool are\_equal(const struct S \*s1, const struct S \*s2) {    if (!s1 && !s2)      return true;    else if (!s1 || !s2)      return false;    return 0 == memcmp(s1, s2, sizeof(struct S));  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the structure members are compared individually. |
| #include <stdbool.h>  #include <string.h>    struct S {    int i;    float f;  };    bool are\_equal(const struct S \*s1, const struct S \*s2) {    if (!s1 && !s2)      return true;    else if (!s1 || !s2)      return false;    return s1->i == s2->i &&           s1->f == s2->f;  } |

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

| **Principles(s):** Correctness, ensures that comparisons are done in a way that guarantees accurate results. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.17.1 | Uninitialized | Analyses C++ code for errors. |
| Clang-Tidy | 13.0.0 | Readability-avoid-const-params | This tool checks for improper or inconsistent data types used in the code. |
| SonarQube | 2025.1 | Type Issues | Identifies improper variables. |
| Flowfinder | 2.0 | Floating point comparison | Looks for vulnerabilities in C++ code. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-STC | Do not attempt to modify string literals. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the char pointer str is initialized to the address of a string literal. Attempting to modify the string literal is undefined behavior. |
| char \*str  = "string literal";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| As an array initializer, a string literal specifies the initial values of characters in an array as well as the size of the array. This code creates a copy of the string literal in the space allocated to the character array str. The string stored in str can be modified safely. |
| char str[] = "string literal";  str[0] = 'S'; |

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

| **Principles(s):** Consistency, avoiding undefined behavior by preventing direct modification of string literals. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.0 | V605 | Detects modification of string literals as part of static analysis. |
| Coverity | 2022.0 | C++ | Analyzes issues with modifying immutable string literals. |
| CppCheck | 2.17.1 | Strings | Identifies potential modifications to string literals. |
| ClangTidy | 13.0.0 | pro-type-const-cast | Ensures that string literals are not modified. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-SQL | Use bound parameters to safely pass user input to SQL queries. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, user inputs (username and password) are directly inserted into the SQL query string using string formatting (f""). This leaves the application vulnerable to SQL injection. |
| query = f"SELECT \* FROM users WHERE username='{username}' AND password='{password}'" |

| **Compliant Code** |
| --- |
| In the compliant code, a parameterized query is used to safely pass user inputs into the SQL query. The “?” placeholders in the query represent where the user inputs should go. |
| 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):** Security First, protect sensitive data to ensure data integrity. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.17.1 | Custom rules | Checks for string operations or raw query building. |
| SonarQube | 2025.1 | cpp:S2077 | Detects SQL injection vulnerabilities in C++ codebases. |
| Coverity Scan | 2022.12.2 | BAD\_SQL | Detects unsafe SQL construction patterns. |
| CodeQL | 3.28.15 | cpp/sql-injection | Checks for C++ to find injection risks. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-MPR | Declare objects with appropriate storage durations. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the address of the variable c\_str with automatic storage duration is assigned to the variable p, which has static storage duration. The assignment itself is valid, but it is invalid for c\_str to go out of scope while p holds its address, as happens at the end of dont\_do\_this(). |
| #include <stdio.h>    const char \*p;  void dont\_do\_this(void) {    const char c\_str[] = "This will change";    p = c\_str; /\* Dangerous \*/  }    void innocuous(void) {    printf("%s\n", p);  }    int main(void) {    dont\_do\_this();    innocuous();    return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, p is declared with the same storage duration as c\_str, preventing p from taking on an [indeterminate value](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-indeterminatevalue) outside of this\_is\_OK(). |
| void this\_is\_OK(void) {    const char c\_str[] = "Everything OK";    const char \*p = c\_str;    /\* ... \*/  }  /\* p is inaccessible outside the scope of string c\_str \*/ |

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

| **Principles(s):** Fail-safe defaults, ensures correct storage duration. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| **Cppcheck** | 2.17.1 | uninitvar, memleak | Detects memory issues and uninitialized variables. |
| **Clang-Tidy** | 13.0.0 | clang-analyzer-core | Analyzes memory lifetime and static analysis. |
| **Coverity Scan** | 2022.12.2 | USE\_AFTER\_RETURN | Detects usage of memory after it has gone out of scope. |
| **SonarQube** | 2025.1 | cpp:S5164, cpp:S1523 | Identifies risky lifetime issues and incorrect memory use. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-ASR | Assertions should be used for conditions that should never happen in a running program. |

| **Noncompliant Code** |
| --- |
| This assertion checks if a is not NULL, but it doesn't provide any explanation if the assertion fails. |
| assert(a != NULL); |

| **Compliant Code** |
| --- |
| In the compliant code, the assertion includes an error message that explains why the check is in place. This helps developers understand the issue and debug it more easily when the assertion fails. |
| assert(a != NULL && "Pointer 'a' should not be NULL"); |

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

| **Principles(s):** Fail-fast design, prevents undefined behavior and helps catch logic errors before the application goes into production. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Low | P4 | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 13.0.0 | clang-analyzer-core | Checks use of undefined or null pointers |
| Coverity Scan | 2022.12.2 | ASSERT\_SIDE\_EFFECT | Flags invalid or risky assertion patterns |
| SonarQube | 2025.1 | cpp:S5806 | Suggests better assertion messages |
| CodeQL | 3.28.15 | cpp/assertion-failure | Highlights problematic or risky assertions |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-EXP | Handle all exceptions. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called. |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    int main() {    f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    int main() {    try {      f();    } catch (...) {      // Handle error    }  } |

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

| **Principles(s):** Reliability, ensures the program remains stable and can recover gracefully from unexpected errors. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P1 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.17.1 | unhandledException | Warns about possible unhandled exceptions |
| Clang-Tidy | 13.0.0 | bugprone-exception-escape | Identifies functions that may throw but aren't marked or caught |
| Coverity Scan | 2022.12.2 | UNCAUGHT\_EXCEPT | Detects unhandled exceptions paths |
| SonarQube | 2025.1 | cpp:S3626 | Ensures that exceptions are caught properly |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Validation | STD-008-VAL | Always validate user inputs to prevent invalid data or security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, user input is not validated before being used in a calculation, which could lead to errors or security issues. |
| #include <iostream>  using namespace std;  int main() {  string userInput;  cout << "Enter age: ";  cin >> userInput;  int age = stoi(userInput); // No validation, could cause an exception if input is invalid  cout << "Age is: " << age << endl;  return 0;  } |

| **Compliant Code** |
| --- |
| In the compliant solution, the user input is validated to ensure it is a valid integer. |
| #include <iostream>  #include <string>  #include <cctype>  using namespace std;  // Function to validate if the input is a number  bool isValidAge(const string& input) {  for (char ch : input) {  if (!isdigit(ch)) {  return false; // Input is invalid if it contains non-digit characters  }  }  return true;  }  int main() {  string userInput;  cout << "Enter age: ";  cin >> userInput;  if (isValidAge(userInput)) {  int age = stoi(userInput);  if (age < 0) {  cout << "Age cannot be negative." << endl;  } else {  cout << "Age is: " << age << endl;  }  } else {  cout << "Invalid input! Please enter a valid number for age." << endl;  }  return 0;  } |

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

| **Principles(s):** Input validation, validate external input to ensure it meets expected formats and ranges. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | P1 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.17.1 | invalidscanf, uninitvar | Flags invalid input handling and uninitialized variables |
| Clang-Tidy | 13.0.0 | cert-err34-c, bugprone-\* | Detects unchecked conversions and misuse of inputs |
| Coverity Scan | 2022.12.2 | TAINTED\_SCALAR, UNVALIDATED\_INPUT | Identifies unvalidated user input that may be tainted |
| SonarQube | 2025.1 | cpp:S3664, cpp:S5782 | Ensures input validation is performed before use |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Variable Naming | STD-009-VAR | Use meaningful variable names. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, variables have non-descriptive names that make the code less readable and maintainable. |
| #include <iostream>  int main() {  int a = 10;  int b = 20;  int c = a + b;  std::cout << c << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| In this code the variable names indicate a meaningful description. |
| #include <iostream>  int main() {  int heightOne = 10;  int heigthTwo = 20;  int sum = heightOne + heigthTwo;  std::cout << sum << std::endl;  return 0;  } |

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

| **Principles(s):** Code readability and maintainability, ensures clear and descriptive variable names makes the code easier to understand, debug, and update in the future. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Likely | Low | P2 | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.17.1 | style: variableScope | Detects poor or unclear variable naming and scope usage |
| Clang-Tidy | 13.0.0 | readability-identifier-naming | Flags unclear or inconsistent naming patterns |
| SonarQube | 2025.1 | cpp:S100, cpp:S1121 | Checks for meaningful names and avoids ambiguous terms |
| CodeQL | 3.28.15 | cpp/ambiguous-variable-name | Identifies variables with unclear or misleading names |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Functions | STD-010-FUN | Use Functions Instead of Repetitive Code. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, the logic for calculating the area of a circle is repeated three times in the main function. |
| #include <iostream>  int main() {  // First Circle Area Calculation  int radius = 5;  double circleArea = 3.14 \* radius \* radius;  std::cout << "Area of circle 1: " << circleArea << std::endl;  // Second Circle Area Calculation  radius = 7;  circleArea = 3.14 \* radius \* radius;  std::cout << "Area of circle 2: " << circleArea << std::endl;  // Third Circle Area Calculation  radius = 10;  circleArea = 3.14 \* radius \* radius;  std::cout << "Area of circle 3: " << circleArea << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| In the noncompliant code, the function for calculating the area of a circle is called when needed. |
| #include <iostream>  // Function to calculate the area of a circle  double calculateCircleArea(int radius) {  return 3.14 \* radius \* radius;  }  int main() {  // Using the function to calculate the area of circles  int radius = 5;  std::cout << "Area of circle 1: " << calculateCircleArea(radius) << std::endl;  radius = 7;  std::cout << "Area of circle 2: " << calculateCircleArea(radius) << std::endl;  radius = 10;  std::cout << "Area of circle 3: " << calculateCircleArea(radius) << std::endl;  return 0;  } |

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

| **Principles(s):** Avoid code repetition, reusing functions avoids code duplication and simplifies maintenance if changes are needed later. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | P3 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.17.1 | style: duplicateCode | Detects duplicate logic and suggests using functions |
| Clang-Tidy | 13.0.0 | readability-function-cognitive-complexity | Highlights repeated logic and complex main functions |
| SonarQube | 2025.1 | cpp:S134, cpp:S138 | Detects duplicate or redundant code blocks |
| PMD C++ | 6.55.0 | AvoidDuplicateLiterals | Identifies repeated logic that could be abstracted |

### 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



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 our software more secure, we are adding a process called DevSecOps. This means we will include security checks in every step of our software process. From the beginning, when we plan a new project, we will think about how to keep it safe. While developers are writing code, tools will check it right away to find mistakes or security issues. Before the code is turned into software, we will use tools like SonarQube and Cppcheck to scan it for errors and unsafe code.

When we test the software, we will run automatic checks to make sure everything is working correctly and safely. Before we release the software, we will do another round of checks to be sure it is secure. When we install the software for users, we will make sure it is set up properly to avoid problems. After that, we will keep watching how the software runs to catch any issues early. We will also collect logs and reports to help us find and fix any problems that come up.

Tools like Jenkins, SonarQube, and Docker will help us by running these checks automatically. If they find something wrong, the process will stop and tell us what needs to be fixed before we can continue. This helps us make sure our software is safe at every step.

### 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-DAT | Medium | Likely | Low | P1 | 1 |
| STD-002-DVL | High | Medium | Medium | P1 | 1 |
| STD-003-STC | High | Unlikely | Low | P1 | 2 |
| STD-004-SQL | High | Likely | Low | P1 | 1 |
| STD-005-MPR | High | Likely | Low | P1 | 1 |
| STD-006-ASR | Medium | Unlikely | Low | P4 | 2 |
| STD-007-EXP | High | Likely | Medium | P1 | 3 |
| STD-008-VAL | High | Likely | Low | P1 | 3 |
| STD-009-VAR | Medium | Likely | Low | P2 | 2 |
| STD-010-FUN | Low | Likely | Low | P3 | 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 | When data is stored on a device, like a computer or a server, it is at rest. Green Pace can use XOR encryption to protect files that are not being used at the moment. For example, before saving a file to a hard drive, the system can use XOR with a secret key to scramble the data. If someone steals the file, they will not be able to read it unless they have the key. |
| Encryption in flight | Encryption in flight means protecting data while it is moving from one place to another, like when a user sends information through a website or when systems talk to each other. We use TLS (Transport Layer Security) or SSL (Secure Sockets Layer) to keep that data safe. This stops hackers from seeing or changing the data while it is moving across the internet or a network. At Green Pace, we will require TLS/SSL for all communication between users and our services, and between different parts of our system. |
| Encryption in use | Encryption in use means keeping data safe even while it is being processed or used in a computer’s memory. This is harder to do, but it is important when we work with very sensitive information. We can use things like secure enclaves or trusted hardware to protect the data. When we are working with high-risk information like personal or financial data, we will use these methods to stop attackers from spying on the data while it is being used. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of checking that someone is who they say they are. It involves a username and password, but we can also add extra steps like sending a code to a phone like a multi-factor authentication or MFA). At Green Pace, all users must log in with secure passwords and use MFA when working with sensitive systems. This helps stop hackers from breaking in using stolen passwords. |
| Authorization | Authorization controls what a user is allowed to do after they log in. For example, a regular employee might be able to view certain files, but only a manager can change them. At Green Pace, we will set permissions based on job roles so that each person only has access to what they need. This reduces the chance of mistakes or misuse. |
| Accounting | Accounting means keeping track of what users do while they are in the system. This includes logging in, when users log in, what files they open, and what changes they make. At Green Pace, we will use automated logs to record this information. These records help us find out what happened if something goes wrong and are important for audits or investigations. |

**\***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 | 03/23/2025 | Improved Policy | Aaron Tellez | John Doe |
| 3.0 | 04/13/2025 | Final Policy | Aaron Tellez | Jonh Doe |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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