**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 | All data inputs must be treated as untrusted and need to be validated to make sure it is the expected format, type, and length. This will help stop attacks like buffer overflows, SQL injections, and unexpected behavior. |
| 1. Heed Compiler Warnings | Compiler warnings are great for pointing or potential bugs or problems in code. Developers should treat all warnings seriously. This includes enabling the highest warning levels and looking at everyone they come across to stop an issue from going into production. |
| 1. Architect and Design for Security Policies | Security should be thought about at all stages of system design including the earliest. This helps to solve problems before they arise, lower the cost of fixing these issues, and making our products security first. |
| 1. Keep It Simple | The simpler developers keep the code the easier it will be to understand, test, maintain, and fix. This wi8ll help reduce the likelihood of adding security flaws, and on the off chance we do it will make fixing them simpler. Overly complex code can hinder our abilities to find vulnerabilities or lead to unintended behavior. |
| 1. Default Deny | Software and systems designed around deny access by default mentality only allow access based on explicit permissions given out. This helps to lower the possibility of united access from users and makes it easier to manage access. |
| 1. Adhere to the Principle of Least Privilege | Processes and systems should work based on the minimum level of permissions required to do the task they are working on. Limiting privilege helps to stop the potential problems that would arise from a flaw in the code, phishing emails, or even breach. |
| 1. Sanitize Data Sent to Other Systems | All data sent out of our code or systems to other systems must be sanitized and validated. This will help to protect against Injections, or data getting stolen in transit. |
| 1. Practice Defense in Depth | Develop and maintain multiple overlapping security controls to protect data and systems. If one layer is hacked or compromised the other layers will help to minimize the impact. |
| 1. Use Effective Quality Assurance Techniques | Security needs to be added to the software testing lifecycle. This includes code reviews, development, and unit testing that test both functionality and security. Ussing effective quality assurance practices helps to detect flaws early, reduce risks, and ensure the code meets a standard. |
| 1. Adopt a Secure Coding Standard | Using a well-developed and recognized secure coding standard, like SEI CERT C++, helps to make sure our codebase has consistent implementations of best practices. This in turn reduces the risk of introducing known vulnerabilities. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Use Fixed-Width Integer Types** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Using fixed-width integer types like int32\_t or uint64\_t helps to make consistent behavior across platforms. Types like int or long can vary in size depending on the system, causing unexpected behavior, overflows, or compatibility issues. |

| **Noncompliant Code** |
| --- |
| The code uses a generic variable type, which can vary in size. |
| Int packetSize = 42918739812: |

| **Compliant Code** |
| --- |
| The code uses fixed-width integer types to make sure behavior is the same on different platforms. |
| # include <cstdint>  int32\_t packetSize = 2147483647: |

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

| **Principles(s):**  **2.** **Heed Compiler Warnings:** Using fixed width integer types can help avoid warnings for overflows, conversions, or truncations. Treating these warnings seriously can help reduce the change of unexpected behavior.  **10. Adopt a Secure Coding Standard:** Fixed-width integers are a key part of the SEI CERT C++ secure coding standard Adopting this practice ensures consistency and prevents known vulnerabilities tied to platform-dependent types. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.15 | type, portability | Detects type inconsistencies and portability issues. |
| Clang-Tidy | 21.0.0 | cert-int30-c, cert-int34-c | Flags platform-dependent types and unsafe conversions |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do Not Use Hard Coded Constants in Code** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Hardcoding values in code, also know as “magic numbers”, reduces the codes maintainability and readability. It can also hide unsafe boundaries like buffer size and increase the risk of adding security errors. Using named constants or setting up variables to use helps to improve clarity and allows for validation in one section of the code. |

| **Noncompliant Code** |
| --- |
| Hardcoded values added to make an array size, which can cause issues if it needed to be changed later in multiple sections. |
| char username[32]:  if (strlen(username)) > 31 {  // example code  } |

| **Compliant Code** |
| --- |
| Constant is defined in a single location and is easy to find and replace. |
| const sixe\_t USERNAME\_MAX\_LENGTH = 32 :  char username[USERNAME\_MAX\_LENGTH]:  if (strlen(username)) >= USERNAME\_MAX\_LENGTH {  // example code  } |

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

| **Principles(s):**  **4. Keep it Simple:** Using named constants makes the code cleaner, easier to understand, and simpler to maintain. This helps developers avoid using or duplicating “Magic numbers”, which can help lead to a reduced issue with logic errors.  **9. Use Effective Quality Assurance Techniques:** Named constants make it easier to test boundary conditions using automation. It also helps static analysis tools to verify values, |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.15 | style | Flags unclear values that should be constant. |
| Clang-Tidy | 21.0.0 | readability-magic-numbers  <https://clang.llvm.org/extra/clang-tidy/checks/readability/magic-numbers.html> | Warns when hard coded literals are used instead of constants. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Properly Null Terminate Strings and Avoid Buffer Overflows** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | String manipulation in C++ is very prone to errors when using functions that do not null terminate buffers or check boundaries. Failure to check these can lead to buffer overflows, unexpected behavior, or even security vulnerabilities. Always use a safer alternative that enforces these rules. |

| **Noncompliant Code** |
| --- |
| This code uses strcpy which does not perform boundary checking and may cause a buffer overflow if the string is to long. |
| Char example[10];  Strcpy(example , userInput) |

| **Compliant Code** |
| --- |
| [Compliant description] |
| Char example[10];  Strcpy(example , userInput , sizeof(example)-1);  example[sizeof(example)-1]=’\0’ |

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

| **Principles(s):**  **1. Validate Input Data:** Using safe string functions or validating length helps prevent untrusted input from exceeding the buffer size.  **8. Practice DID:** Even if input validation occurs in other sections of the code, adding multiple checks is always good. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.15 | bufferAccessOutOfBounds, style | Detects unsafe use of string functions and improper null termination. |
| Clang-Tidy | 21.0.0 | clang-analyzer-security.insecureAPI.strcpy <https://clang.llvm.org/extra/clang-tidy/checks/clang-analyzer/security.insecureAPI.strcpy.html> | Warns when functions access buffers without bounds checking. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Use Prepared Statements** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Making SQL queries through string concatenation adds several security risks, especially if user input is involved. This can lead to SQL injection attacks, causing users to have more access than they should. Using parameterized queries or premade queries can help make sure input is treated as data and not executable code. |

| **Noncompliant Code** |
| --- |
| This code shows an example of a SQL query made by concatenating user inputs directly into the query. |
| Std::string query = “SELECT \* FROM users WHERE username = ‘“ + username: “’;”;  db.execute(query) |

| **Compliant Code** |
| --- |
|  |
| PreparedStatement\* stmt = conn -> prepareStatement(  “SELECT \* FROM users WHERE username = ?”  );  Stmt->setString(1,username);  ResultSet\* res = stmt->executeQuery(); |

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

| **Principles(s):**  **1. Validate Input Data:** Using prepared statements ensures user input is validated and handles literal data. **7. Sent to Sanitize Data To Other Systems:** When sending user data from the database, you must sanitize it from executable instructions. Using parameterized queries enforces this. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify SCA | 23.2 | SQL Injection: Database | Advanced analysis for database queries for injections. |
| Cppcheck | 2.15 | security | Provides basic checks for unsafe external input handling. |
| Clang-Tidy | 21.0.0 | optin.taint.GenericTaint | Analyzes use of insecure query construction APIs. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Free Dynamically Allocated Memory Once** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Failing to manage dynamically allocated memory properly can lead to vulnerabilities like memory leaks, double frees, or use-after-free errors. These kinds of bugs can lead to crashes, data corruption, or exploitable states. Programs must make sure that memory is released once and not references after. |

| **Noncompliant Code** |
| --- |
| The following code shows memory being free twice which can lead to issues. |
| Int\* data = new int[100];  delete[] data;  delete[] data; |

| **Compliant Code** |
| --- |
| [This code frees the memory and then nullifies the pointer to stop and use after the deletion. |
| Int\* data = new int[100];  delete[] data;  Data = nullptr; |

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

| **Principles(s):**  6. Adhere to the Principles of Least Privilege: Poor memory handling can lead to privilege escalations or uncontrolled access to memory that should not be accessed. Freeing memory properly ensures this can’t happen. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.15 | memleak, useAfterFree | Identifies memory not released and potential use-after-free issues. |
| Clang-Tidy | 21.0.0 | clang-analyzer-cplusplus.NewDeleteLeaks | Identifies memory that’s not freed, or freed more than once. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **DO Not Rely On Assertions For Runtime behavior** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions are meant for catching logic errors in code when developing, not for handling runtime condition errors. Since assertions can be disabled in production builds, relying on them to check conditions can add vulnerabilities. Instead add proper error handling and input validation. |

| **Noncompliant Code** |
| --- |
| This code uses an assertion to try and enforce a runtime condition.This can be ignored if assertions are disabled to release builds. |
| assert(userInput != nullptr);  useInput(userInput); |

| **Compliant Code** |
| --- |
| This code uses a check during runtime and handles an errors itself. |
| if (userInput == nullptr){  logError(“Null pointer recived”)  return;  }  useInput(userInput); |

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

| **Principles(s):**  **3. Architect and Design for Security Policies:** Runtime conditions should be apart of the system design and not just left up to development tools like assertions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.15 | style, cert | Warns user when the code uses assert for non-testing. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Catch Exceptions by Reference, Not Value** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Catching exceptions by value can result in unnecessary copying or object slicing, which can cause a loss of information or performance issues. Catching by reference ensure this does not happen. |

| **Noncompliant Code** |
| --- |
| The following code catches the exception using its value. |
| Try{  Throw std::runtime\_error(“Error occured”);  } catch (std::exception e){  Std::cerr << e.what() ,, std::endl;  } |

| **Compliant Code** |
| --- |
| This code catches the exception by reference , keeping the full type information and avoiding any unneeded copies. |
| Throw std::runtime\_error(“Error occured”);  } catch (const std::exception e){  Std::cerr << e.what() ,, std::endl;  } |

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

| **Principles(s):**  **4. Keep it Simple:** Catching exceptions by reference makes exception handling simpler by preventing unexpected behavior and maintains the object.  **10. Adopt a Secure Coding Standard:** This best practice is directly reflected in SEI CERT C++ rules, ERR61-CPP, which says to catch exceptions by reference to avoid object slicing and maintain error detail. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.15 | style, cert | Warns about bad exception practices like slicing. |
| Clang-Tidy | 21.0.0 | cert-err61-cpp, misc-throw-by-value-catch-by-reference | Catches exceptions that are caught by value instead of by reference. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Initialize Variable Before Use** |
| --- | --- | --- |
| Expressions | STD-008-CPP | Using variables before they are initialized can lead to unwanted behavior, crashes, or security problems. Always explicitly initialize them to know values before using them. |

| **Noncompliant Code** |
| --- |
| This code uses an uninitialized variable, which can have garbage data in it and produce weird results. |
| int total;  std::cout << “”Total is: “ >>total << std::endl; |

| **Compliant Code** |
| --- |
| This code initializes the variable with a value before it is used. |
| int total = 0;  std::cout << “”Total is: “ >>total << std::endl; |

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

| **Principles(s):**  **2. Heed Compiler Warnings:** Compilers will often warn when variables are uninitialized. Using these warnings can help prevent runtime issues and security vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.15 | uninitvar | Detects uninitialized variables |
| Clang-Tidy | 21.0.0 | clang-analyzer-core.uninitialized.Assign | Detects use of uninitialized variables in logic or assignments. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Use Nullptr Instead of Null or 0 For Pointers** |
| --- | --- | --- |
| Expressions | STD-009-CPP | Using Nullptr improves type safety compared to using NULL or 0, which can cause problems. |

| **Noncompliant Code** |
| --- |
| This code uses Bull, which can lead to confusion or mistaken for an integer. |
| Int\* ptr = NULL; |

| **Compliant Code** |
| --- |
| This code uses nullptr making sure it is clear and helps with type safety. |
| Int\* ptr = nullptr; |

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

| **Principles(s):**   1. **Adopt a Secure Coding Standard:** Using nullptr is aligned with modern secure coding practices and is recommended in SEI CERT, EXP54-CPP, which says we shouldn’t use NULL or 0. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.15 | style, modernize | Warns on NULL usage |
| Clang-Tidy | 21.0.0 | modernize-use-nullptr | Recommends and replaces NULL or 0 with nullptr. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Do Not Modify Constant Objects** |
| --- | --- | --- |
| Declaration and Initializations | STD-010-CPP | Trying to modify objects declared as consts by casting them or going around the const leads to unexpected behavior. This goes around the whole point of an immutable object and cause serious bugs or security issues. |

| **Noncompliant Code** |
| --- |
| This code attempts to cast away a const and modify the object. |
| Const int value = 1;  int\* ptr =const\_cast<int\*>(&value);  \*ptr = 20; |

| **Compliant Code** |
| --- |
| The following code does not try to modify the cont and uses it as is. |
| Const int value = 1;  std::out << “value: “ << value << std::endl; |

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

| **Principles(s):**  **3. Architect and Design for Security Policies:** Marking values as const is a design choice to prevent unintended changes. Going around that choice breaks the safety model and adds unexpected behavior. **4. Keep it Simple:** Respecting a constants declaration simplifies its use in the code and makes managing it not weird. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.15 | constStatement, style | Warns when a const value is modified or cast. |
| Clang-Tidy | 21.0.0 | cppcoreguidelines-avoid-c-arrays | Warns when a const value is modified or cast. |

### 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.

Green Pace will use automation throughout the DevSecOps process to make sure all the coding standards listed above are followed. In the pre-production phase, we will use tools like Cppcheck and Clang-Tidy built into our development env and Ci Pipelines. These tools will automatically scan our code for issues like uninitialized variables, unsafe pointers, or SQL injection risk points. During the Design and Bild phase, IDE plugins and static analysis will help developers write secure code to start off with. Automated testing tools will also run checks for vulnerabilities and make sure the code meets both functionality requirements and our new security requirements.

In the production phase, we can use tools like a SIEM (for Security Information and Event Management) system and runtime protection can monitor for attacks or weird behavior. If something goes wrong, automated response systems can isolate the issue, warn an admin, and possible rollback to safe system state. If this were to be layered with other security measures this would help Green pace enforce secure coding policies, reduce human error, and catch security issues early on in the development process.

### 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 | Medium | Likely | Low | High | 2 |
| STD-002-CPP | Medium | Likely | Low | High | 2 |
| STD-003-CPP | High | Likely | Medium | High | 1 |
| STD-004-CPP | High | Likely | Medium | Critical | 1 |
| STD-005-CPP | High | Likely | Medium | High | 1 |
| STD-006-CPP | Medium | Possible | low | Medium | 3 |
| STD-007-CPP | Medium | Possible | Low | Medium | 3 |
| STD-008-CPP | Medium | Likely | Low | High | 2 |
| STD-009-CPP | Low | Likely | Low | Low | 3 |
| STD-010-CPP | High | Unlikely | Low | High | 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 means protecting data stored on a device, like a hard drive. We use strong encryption, like AES-256, so if someone gets physical access to our storage, they can’t just read the data. This policy would apply to all data we store including but not limited to logs, files, databases, and backups. |
| Encryption in flight | This protects data while it is being sent over a network. Using Protocols Like HTTPS keeps this data safe from interceptions or man in the middle attacks. This policy should apply to all user and system communications. |
| Encryption in use | This protects data while it is being used or processed in memory. Making sure the data is only available for the time it is needed is important. Clearing memory after the data is no longer needed is very important. This should apply to anytime data is used. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Verifying who the user is. Users must log in using strong passwords or multi-factor authentication (MFA). This protects against unauthorized access and should apply when anyone tries to log into the system. |
| Authorization | Controlling what a user is allowed to do. By default, all users should have the least amount of access to do their job. This applies when managing user roles, adding new users, and assigning access levels to files or databases. |
| Accounting | Track user activity. The system should log things like user logins, file access, and change to the database. This helps detect suspicious activity and supports audits. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.2 | 6/15/2025 | Project 1 | Devin Wheeler | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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