**Chris Marrs: 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 | Test inputs ensuring only appropriate and secure information enters an information system; any suspicious or harmful input must be rejected at this stage. |
| 1. Heed Compiler Warnings | Compiler warnings are designed to alert a developer about any errors or potential security vulnerabilities within their code, prompting changes aimed at eliminating it altogether. Warnings don't preclude compilation, while errors prevent it completely, yet both types are equally essential in mitigating potential security risks in code modification efforts. |
| 1. Architect and Design for Security Policies | Architecture and design can play an essential role when it comes to implementing security policies, such as splitting a system into sub-systems with differing levels of authorization or privilege. |
| 1. Keep It Simple | By keeping designs as straightforward and straightforward as possible, you reduce the likelihood of both programming errors and user mistakes, and also possibly decrease security complexity requirements. |
| 1. Default Deny | By default, access is denied and can only be allowed under certain conditions of the protection scheme in use. |
| 1. Adhere to the Principle of Least Privilege | Processing should run with only those privileges necessary, while elevated privileges should only be utilized as needed and for as short a duration as possible. This reduces the chance that an attacker could exploit higher privileges to execute code with elevated permissions. |
| 1. Sanitize Data Sent to Other Systems | When passing data between systems, it should always be checked for possible issues before passing it along - including functions that are called out of context, such as SQL injection attacks. Sanitizing your data prior to invoking new systems checks these potential threats before their invocation occurs. |
| 1. Practice Defense in Depth | By Employing multiple layers of defense, it is possible to reduce any possible exploits or damages should one layer become vulnerable. |
| 1. Use Effective Quality Assurance Techniques | To ensure an effective quality assurance program, proper testing techniques such as fuzz and penetration testing as well as audits against code can play an essential part. Furthermore, security reviews from both internal and external parties may help identify and address potential problems more quickly and efficiently. |
| 1. Adopt a Secure Coding Standard | For an assured start, adopt secure coding standards in the language and platform of your choice to protect your code from day one. |

### 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** | **Obey the one-definition rule** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Ensures consistency and avoids undefined behavior by requiring that every non-inline function or variable that is odr-used (used in a way that requires a definition) in a program has exactly one definition. |

| **Noncompliant Code** |
| --- |
| Two different translation units define a class of the same name with differing definitions. |
| // a.cpp  struct S {  int a;  };    // b.cpp  class S {  public:  int a;  }; |

| **Compliant Code** |
| --- |
| Use a header file to introduce the object into both translation units. |
| // S.h  struct S {  int a;  };    // a.cpp  #include "S.h"    // b.cpp  #include "S.h" |

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

| **Principles(s):** 3. Architect and Design for Security Policies - Ensures exceptions are properly handled.  4. Keep It Simple - Avoids complexity by ensuring valid enumerations.  10. Adopt a Secure Coding Standard – Minimizes the opportunity for errors to be left in. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2023.6.2 | ODR | Ensures no multiple definitions across translation units. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do not depend on the order of evaluation for side effects** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Do not allow the same scalar object to appear in side effects or value computations in both halves of an unsequenced or indeterminately sequenced operation. |

| **Noncompliant Code** |
| --- |
| Here, i is evaluated more than once in an unsequenced manner, so the behavior of the expression is undefined. |
| void f(int i, const int \*b) {  int a = i + b[++i];  // ...  } |

| **Compliant Code** |
| --- |
| This example is independent of the order of evaluation of the operands and can each be interpreted in only one way |
| void f(int i, const int \*b) {  ++i;  int a = i + b[i];  // ...  } |

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

| **Principles(s):** 2. Heed Compiler Warnings - Avoids undefined behavior due to order dependencies. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang | 19.0.0 | OrderOfEvaluation | Detects dependencies on order of evaluation. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Guarantee that storage for strings has sufficient space for character data and the null terminator** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Ensure that the destination is of sufficient size to hold the data to be copied. |

| **Noncompliant Code** |
| --- |
| Because the input is unbounded, the following code could lead to a buffer overflow. |
| #include <iostream>    void f() {  char buf[12];  std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| The best solution for ensuring that data is not truncated and for guarding against buffer overflows is to use std::string instead of a bounded array, as in this compliant solution. |
| #include <iostream>  #include <string>    void f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):** 7. Sanitize Data Sent to Other Systems - Prevents buffer overflow vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang | LLVM 19.0.0 | cplusplus.StringChecker (C++) | Checks for NULL termination. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevent SQL injection** |
| --- | --- | --- |
| **SQL Injection** | STD-004-JAV | SQL injection vulnerabilities a present in applications where elements of a SQL query originate from an untrusted source. The untrusted data may maliciously alter the query, resulting in information leaks or data modification. |

| **Noncompliant Code** |
| --- |
| this code example permits a SQL injection attack by incorporating the unsanitized input argument username into the SQL command, allowing an attacker to inject validuser' OR '1'='1. |
| String sqlString = "SELECT \* FROM db\_user WHERE username = '"  + username +  "' AND password = '" + pwd + "'"; |

| **Compliant Code** |
| --- |
| [Compliant description] |
| String sqlString =  "select \* from db\_user where username=? and password=?"; PreparedStatement stmt = connection.prepareStatement(sqlString); |

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

| **Principles(s):** 1. Validate Input Data – prevents injection of code from the UI.  7. Sanitize Data Sent to Other Systems – prevents sending bad data to components/systems.  10. Adopt a Secure Coding Standard - Ensures SQL queries are safe from injection attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| sonarqube | 10.5 | CWE-89-SQL Injection | Taint analysis to detect SQL Injection |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Do not access freed memory** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Dangling pointers are memory pointers that have been re-allocated. By accessing a dangling pointer, vulnerabilities can be exploited. |

| **Noncompliant Code** |
| --- |
| Here, s is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| The dynamically allocated memory is not deallocated until it is no longer required. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  s->f();  delete s;  } |

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

| **Principles(s):** 2. Heed Compiler Warnings – warning could be pointing to dead memory.  5. Default Deny – by using only known memory, code cannot be injected and used outside of that space.  6. Adhere to the Principle of Least Privilege – only valid access and only in controlled memory  9. Use Effective Quality Assurance Techniques – prevention is key to prevent creating openings. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang | 3.9 | clang-analyzer-cplusplus.NewDelete clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use a static assertion to test the value of a constant expression** |
| --- | --- | --- |
| **Assertions** | STD-006-CLG | Assertions are an important diagnostic tool to find and eliminate software defects which may lead to vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The assert() macro here asserts a property concerning a memory-mapped structure that is essential for the code to behave correctly. |
| #include <assert.h>    struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    int func(void) {  assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| For assertions involving only constant expressions, a preprocessor conditional statement may be used, as in this compliant solution. |
| struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))  #error "Structure must not have any padding"  #endif |

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

| **Principles(s):** 8. Practice Defense in Depth - Ensures constant expressions are correct and secure. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang | 3.9 | misc-static-assert | Checked by clang-tidy |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Any exception thrown by an application should be caught by a matching exception handler and addressed gracefully, even if recovery cannot occur immediately. Using such a handler ensures the stack unwinds correctly while providing an opportunity to manage external resources before terminating the process. |

| **Noncompliant Code** |
| --- |
| Here, 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** |
| --- |
| 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):** 3. Architect and Design for Security Policies - Ensures exceptions are properly handled. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.12 | ExceptionHandling | Detects unhandled exceptions. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Close files when they are no longer needed** |
| --- | --- | --- |
| [Student Choice] | STD-008-CPP | A call to the std::basic\_filebuf<T>::open() function must be matched with a call to std::basic\_filebuf<T>::close() before the lifetime of the last pointer that stores the return value of the call has ended or before normal program termination, whichever occurs first. |

| **Noncompliant Code** |
| --- |
| A std::fstream object file is constructed. The constructor for std::fstream calls std::basic\_filebuf<T>::open(), and the default std::terminate\_handler called by std::terminate() is std::abort(), which does not call destructors. Consequently, the underlying std::basic\_filebuf<T> object maintained by the object is not properly closed. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ...  std::terminate();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, std::fstream::close() is called before std::terminate() is called, ensuring that the file resources are properly closed. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ...  file.close();  if (file.fail()) {  // Handle error  }  std::terminate();  } |

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

| **Principles(s):** 5. Default Deny - Ensures resources are properly managed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2023.6.2 | FileHandling | Ensures files are properly closed. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do not cast to an out-of-range enumeration value** |
| --- | --- | --- |
| [Student Choice] | STD-009-CPP | For safe operation, any value cast must fall within the range of values the enumeration can represent. When checking dynamically for out-of-range values, checking must take place prior to performing casting expressions. |

| **Noncompliant Code** |
| --- |
| Here is an attempt to check whether a given value is within the range of acceptable enumeration values. However, it is doing so after casting to the enumeration type, which may not be able to represent the given integer value. On a two's complement system, the valid range of values that can be represented by EnumType are [0..3], so if a value outside of that range were passed to f(), the cast to EnumType would result in an unspecified value, and using that value within the if statement results in unspecified behavior. |
| enum EnumType {  First,  Second,  Third  };    void f(int intVar) {  EnumType enumVar = static\_cast<EnumType>(intVar);    if (enumVar < First || enumVar > Third) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| Check that the value can be represented by the enumeration type before performing the conversion to guarantee the conversion does not result in an unspecified value. It does this by restricting the converted value to one for which there is a specific enumerator value. |
| enum EnumType {  First,  Second,  Third  };    void f(int intVar) {  if (intVar < First || intVar > Third) {  // Handle error  }  EnumType enumVar = static\_cast<EnumType>(intVar);  } |

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

| **Principles(s):** 3. Architect and Design for Security Policies – prevent access to unmanaged memory.  4. Keep It Simple - Avoids complexity by ensuring valid enumerations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang | 3.9 | EnumRange | Checks for out-of-range enumeration casts. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Guarantee that container indices and iterators are within the valid range** |
| --- | --- | --- |
| [Student Choice] | STD-010-CPP | Programmers are entirely responsible for ensuring array references fall within their array's bounds. When using vectors from standard template libraries they should ensure integer indexes fall within these bounds as well. |

| **Noncompliant Code** |
| --- |
| A function, insert\_in\_table(), that has two int parameters, pos and value, both of which can be influenced by data originating from untrusted sources. The function performs a range check to ensure that pos does not exceed the upper bound of the array, specified by tableSize, but fails to check the lower bound. Because pos is declared as a (signed) int, this parameter can assume a negative value, resulting in a write outside the bounds of the memory referenced by table. |
| #include <cstddef>    void insert\_in\_table(int \*table, std::size\_t tableSize, int pos, int value) {  if (pos >= tableSize) {  // Handle error  return;  }  table[pos] = value;  } |

| **Compliant Code** |
| --- |
| The parameter pos is declared as size\_t, which prevents the passing of negative arguments. |
| #include <cstddef>    void insert\_in\_table(int \*table, std::size\_t tableSize, std::size\_t pos, int value) {  if (pos >= tableSize) {  // Handle error  return;  }  table[pos] = value;  } |

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

| **Principles(s):** 6. Adhere to the Principle of Least Privilege - Ensures indices are within valid ranges, preventing unauthorized access. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify | 24.2 | IndexOutOfBounds | Ensures array and vector indices are within bounds. |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

Automation will be utilized for the enforcement and compliance with standards specified in this policy. Green Pace already boasts an established DevOps process and infrastructure. Automated security tools will be integrated into both continuous integration (CI) and continuous deployment (CD) pipelines to ensure code is analyzed for compliance with security standards throughout its development cycle. Coverity, Clang, Fortify, SonarQube, Valgrind and PVS-Studio will be used to perform static analysis, detect vulnerabilities and enforce coding standards. They will run during code check-ins, build processes and pre-deployment stages so issues are quickly identified and addressed; and their results reviewed regularly to ensure ongoing compliance and adapt policies as required based on new vulnerabilities or security threats.

### 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 | Low | Medium | High | 2 |
| STD-002-CPP | Medium | High | Low | High | 3 |
| STD-003-CPP | High | Medium | Medium | High | 2 |
| STD-004-JAV | High | High | Low | High | 1 |
| STD-005-CPP | High | Medium | Medium | High | 3 |
| STD-006-CLG | Medium | Low | Low | Medium | 4 |
| STD-007-CPP | High | Medium | Medium | High | 2 |
| STD-008-CPP | Medium | Medium | Low | Medium | 3 |
| STD-009-CPP | Medium | Low | Low | Medium | 4 |
| STD-010-CPP | High | Medium | Medium | 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***.***

Explain each type of encryption, how it is used, and why and when the policy applies.

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 refers to protecting data stored on physical media. This policy ensures that all sensitive data is encrypted when stored to prevent unauthorized access in the event of physical theft or unauthorized access to the storage medium. The policy applies to databases, file storage, and backups. |
| Encryption in flight | Encryption in use protects data while it is being processed or used in memory. This policy requires the use of secure enclaves or memory encryption technologies to ensure that sensitive data remains protected during processing. The policy applies to applications and systems handling sensitive information. |
| Encryption in use | Encryption in use protects data while it is being processed or used in memory. This policy requires the use of secure enclaves or memory encryption technologies to ensure that sensitive data remains protected during processing. The policy applies to applications and systems handling sensitive information. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication ensures that only authorized users can access systems and data. This policy requires the use of strong, multi-factor authentication methods to verify user identities. The policy applies to all access points, including user logins, new user account creation, API access, and administrative interfaces. |
| Authorization | Authorization controls the access levels and permissions granted to users. This policy mandates the implementation of role-based access control (RBAC) to ensure that users only have access to the resources necessary for their roles. The policy applies to all systems and applications. |
| Accounting | Accounting involves tracking and logging user activities. This policy requires comprehensive logging of user actions, including logins, files and data access, and changes made to the system. The logs should be reviewed regularly to detect and respond to suspicious activities. The policy applies to all systems and applications. |

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

User logins

Changes to the database

Addition of new users

User level of access

Files accessed by users

### Map the Principles

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

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

Operating system logs

Firewall logs

Anti-malware logs

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

## Audit Controls and Management

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

Evidence will include the following:

Code compliance to standards

Well-documented access-control strategies, with sampled evidence of compliance

Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use

Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

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

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

## Exceptions Process

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

Business or technical rationale

Risk impact analysis

Risk mitigation analysis

Plan to come into compliance

Date for when the plan to come into compliance will be completed

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 06/15/2024 | Updated for Project One | Chris Marrs |  |
| [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 |