**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 | Validating input data is essential for ensuring that the data provided by users or other systems is both correct and secure. This involves checking the input against expected patterns or ranges, sanitizing to remove any malicious code, and ensuring that the input meets the required format. Proper validation prevents common vulnerabilities such as SQL injection, cross-site scripting (XSS), and buffer overflows. |
| 1. Heed Compiler Warnings | Compiler warnings often indicate potential vulnerabilities or problematic code practices that could lead to security issues. Developers should address these warnings promptly to ensure that the code adheres to best practices and does not contain flaws that could be exploited. Ignoring compiler warnings can lead to overlooked bugs that compromise the system's security. |
| 1. Architect and Design for Security Policies | Security must be integrated into the architecture and design phases of software development. This includes defining security policies that address data protection, access control, and threat mitigation. By incorporating security from the outset, systems can be built with robust defenses that anticipate and counteract potential attacks. |
| 1. Keep It Simple | Complex systems are more prone to security vulnerabilities due to the increased likelihood of bugs and the difficulty of thoroughly testing all components. Keeping the design and implementation simple makes it easier to identify, understand, and fix security issues. Simplicity also aids in maintaining and updating the system securely over time. |
| 1. Default Deny | The default deny principle states that access should be denied by default and only explicitly granted when necessary. This minimizes the attack surface by ensuring that only authorized users and processes can access sensitive resources. Implementing default deny requires careful management of permissions and regular reviews to ensure that access controls remain tight. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege mandates that users and systems are granted the minimum level of access necessary to perform their functions. This reduces the risk of malicious activity or accidental damage by limiting the actions that can be taken if an account or process is compromised. Regularly reviewing and adjusting permissions is crucial to maintaining least privilege. |
| 1. Sanitize Data Sent to Other Systems | When data is transmitted between systems, it should be sanitized to remove any potentially harmful content. This prevents attacks such as SQL injection or cross-site scripting when the data is processed by the receiving system. Ensuring that only clean, safe data is exchanged helps maintain the integrity and security of interconnected systems. |
| 1. Practice Defense in Depth | Defense in depth is a security strategy that employs multiple layers of defense to protect systems and data. By implementing a combination of preventive, detective, and responsive measures, organizations can create a more resilient security posture. If one layer is breached, additional layers provide continued protection, reducing the overall risk of a successful attack. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance techniques, such as code reviews, automated testing, and static analysis, help identify and resolve security vulnerabilities early in the development process. Ensuring that code meets high standards of quality and security reduces the likelihood of flaws being introduced and exploited in production environments. Continuous integration and testing further enhance security assurance. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard involves following established guidelines and best practices for writing secure code. This includes avoiding known vulnerabilities, using safe functions, and adhering to security principles throughout the development lifecycle. Secure coding standards help ensure that all developers within an organization produce code that is consistently secure and reliable. |

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Do not write syntactically ambiguous declarations |

| **Noncompliant Code** |
| --- |
| Using ambiguous declarations can lead to misunderstandings and potential errors in the code. This ambiguity can result in different interpretations by different compilers, leading to inconsistent behavior. |
| int \*p, q; |

| **Compliant Code** |
| --- |
| Declarations should be clear and unambiguous, making sure that the intended data types are easily understood. This clarity helps prevent errors and makes the code more maintainable. |
| int \*p;  int q; |

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

| **Principles(s):** Single Responsibility Principle: This principle emphasizes that every module or class should have responsibility over a single part of the functionality provided by the software. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12 | Style | Clang-Tidy is a clang-based C++ “linter” tool to help identify and fix typical programming errors. |
| CppCheck | 2.6 | Style | Cppcheck is an analysis tool for C/C++ code that detects various errors, including ambiguous declarations. |
| PVS-Studio | 7.15 | V807 | PVS-Studio is a static code analysis tool for detecting bugs and potential vulnerabilities in the source code of programs. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Do not access an object outside of its lifetime |

| **Noncompliant Code** |
| --- |
| Accessing an object outside of its lifetime can lead to undefined behavior, crashes, or security vulnerabilities. This happens when an object is used after it has been destroyed, leading to invalid memory access. |
| #include <iostream>  void example() {  int\* p = new int(42);  delete p;  std::cout << \*p << std::endl; // Accessing object outside its lifetime  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the pointer p is set to nullptr after the object is deleted to make sure it is not accessed again, preventing undefined behavior. |
| #include <iostream>  void example() {  int\* p = new int(42);  delete p;  p = nullptr;  if (p) {  std::cout << \*p << std::endl;  }  } |

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

| **Principles(s):** Resource Management Principle: This principle emphasizes proper management of resources, such as memory, ensuring they are allocated and deallocated correctly. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0 | misc-dangling-handle | - |
| CppCheck | 2.6 | memoryLeak | - |
| PVS-Studio | 7.15 | V611 | - |
| Valgrind | 3.17 | Memcheck | Valgrind is a programming tool for memory debugging, memory leak detection, and profiling. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Range check element access |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an element is accessed without checking if the index is within the valid range of the vector, which may be leading to out-of-bounds access. |
| #include <vector>  #include <iostream>  void printElement(const std::vector<int>& vec, size\_t index) {  std::cout << vec[index] << std::endl; // Potential out-of-bounds access  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the element access is performed after a range check to make sure the index is within the valid range of the vector, preventing out-of-bounds access. |
| #include <vector>  #include <iostream>  void printElement(const std::vector<int>& vec, size\_t index) {  if (index < vec.size()) {  std::cout << vec.at(index) << std::endl;  } else {  std::cerr << "Index out of range" << std::endl;  }  } |

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

| **Principles(s):** Defensive Programming Principle: This principle emphasizes writing code that is robust and can handle unexpected inputs or situations gracefully. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0 | cert-oop11-cpp | - |
| CppCheck | 2.6 | Bounds | - |
| PVS-Studio | 7.15 | V519 | - |
| AddressSanitizer | 11.0 | address | AddressSanitizer is a fast memory error detector which can find out-of-bounds accesses. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Prevent SQL Injection |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, user input is directly concatenated into an SQL query, making it vulnerable to SQL injection attacks. |
| std::string username = getRequestString("username");  std::string password = getRequestString("password");  std::string query = "SELECT \* FROM Users WHERE Name = '" + username + "' AND Pass = '" + password + "'"; |

| **Compliant Code** |
| --- |
| In this compliant solution, a prepared statement is used to safely insert user input into the SQL query, preventing SQL injection attacks. |
| PreparedStatement\* stmt = conn->prepareStatement("SELECT \* FROM Users WHERE Name = ? AND Pass = ?");  stmt->setString(1, username);  stmt->setString(2, password);  stmt->executeQuery(); |

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

| **Principles(s):** Principle of Least Privilege: This principle emphasizes minimizing the number of privileges given to a user or process, reducing the potential impact of a security breach. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | Sql-injection | SonarQube is an open-source platform for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs, code smells, and security vulnerabilities. |
| Fortify Static Code Analyzer | 21.1.1 | SQL Injection | Fortify SCA is a static code analyzer that identifies vulnerabilities in source code, including SQL injection risks. |
| Coverity | 2022.12 | SQL Injection | Coverity is a static analysis tool that finds critical defects and security vulnerabilities in code, including SQL injection. |
| OWASP ZAP | 2.10 | SQL Injection Scanner | OWASP ZAP is an open-source web application security scanner, which helps to find security vulnerabilities in web applications, including SQL injection vulnerabilities. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Detect and handle memory allocation errors |

| **Noncompliant Code** |
| --- |
| Without checking for memory allocation errors, the program may dereference a null pointer, leading to undefined behavior. |
| int\* data = new int[1000];  memset(data, 0, 1000 \* sizeof(int)); // Potential null dereference if allocation fails |

| **Compliant Code** |
| --- |
| The compliant solution checks for successful memory allocation and handles errors appropriately. |
| int\* data = new (std::nothrow) int[1000];  if (!data) {  // Handle allocation failure according to SOP  } |

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

| **Principles(s):** Fail-Safe Defaults Principle: This principle states that systems should default to a safe state in case of failure. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0 | Cert-err34-c | - |
| CppCheck | 2.6 | Memory Leak | - |
| PVS-Studio | 7.15 | V668 | - |
| Valgrind | 3.17 | memcheck | - |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Use a static assertion to test the value of a constant expression |

| **Noncompliant Code** |
| --- |
| Without using static assertions, the code might silently fail if a constant expression does not meet certain conditions, which can cause undefined behavior at runtime. |
| #define ARRAY\_SIZE 10  int arr[ARRAY\_SIZE];  if (ARRAY\_SIZE <= 0) {  // Handle error at runtime  } |

| **Compliant Code** |
| --- |
| The compliant solution uses a static assertion to check that a constant expression meets specific conditions at compile time. |
| #define ARRAY\_SIZE 10  static\_assert(ARRAY\_SIZE > 0, "Array size must be greater than 0");  int arr[ARRAY\_SIZE]; |

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

| **Principles(s):** Fail-Fast Principle: This principle emphasizes detecting and addressing errors as early as possible in the development process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0 | Misc-static-assert | - |
| CppCheck | 2.6 | unusedFunction | - |
| PVS-Studio | 7.15 | V1234 | - |
| GCC | 10.2 | Wstatic-assert | GCC is a compiler that supports various warnings and static assertions to help detect issues at compile time. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Do not abruptly terminate the program |

| **Noncompliant Code** |
| --- |
| Abruptly terminating a program using exit or similar functions can lead to resource leaks, inconsistent states, and loss of data. |
| #include <iostream>  #include <cstdlib>  void processInput(int input) {  if (input < 0) {  std::cerr << "Error: Negative input\n";  std::exit(EXIT\_FAILURE);  }  // Process input  } |

| **Compliant Code** |
| --- |
| The compliant solution handles errors gracefully by cleaning up resources and providing a useable error messages before terminating the program. |
| #include <iostream>  #include <stdexcept>  void processInput(int input) {  if (input < 0) {  throw std::invalid\_argument("Negative input");  }  // Process input  } |

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

| **Principles(s):** Principle of Robustness: This principle emphasizes the importance of handling errors and unexpected conditions gracefully, ensuring that the program can recover or terminate in a controlled manner. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0 | Cert-err33-c | - |
| CppCheck | 2.6 | Exit | - |
| PVS-Studio | 7.15 | V1051 | - |
| Coverity | 2022.12 | CERT C ERR33-C | - |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Output | STD-008-CPP | Close files when they are no longer needed |

| **Noncompliant Code** |
| --- |
| Failing to close files when they are no longer needed can lead to resource leaks, potential file corruption, and issues with system resources not being freed. |
| #include <fstream>  void writeToFile(const std::string& filename, const std::string& data) {  std::ofstream outfile(filename);  if (outfile.is\_open()) {  outfile << data;  // File not closed  }  } |

| **Compliant Code** |
| --- |
| The compliant solution ensures that files are properly closed after use, preventing resource leaks and ensuring data integrity. |
| #include <fstream>  void writeToFile(const std::string& filename, const std::string& data) {  std::ofstream outfile(filename);  if (outfile.is\_open()) {  outfile << data;  outfile.close(); // File closed properly  }  } |

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

| **Principles(s):** Resource Management Principle |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0 | clang-analyzer-cplusplus.CloseFile | - |
| CppCheck | 2.6 | resourceLeak | - |
| PVS-Studio | 7.15 | V773 | - |
| Coverity | 2022.12 | RESOURCE\_LEAK | - |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Miscellaneous | STD-009-CPP | Do not use std::rand() for generating pseudorandom numbers |

| **Noncompliant Code** |
| --- |
| Using std::rand() for generating pseudorandom numbers can lead to issues like non-uniform distribution and low-quality randomness. |
| #include <cstdlib>  #include <iostream>  int main() {  int random\_number = std::rand() % 100; // Non-uniform distribution  std::cout << "Random number: " << random\_number << "\n";  return 0;  } |

| **Compliant Code** |
| --- |
| The compliant solution uses the <random> library to generate pseudorandom numbers, which provides better control over the quality and distribution of random numbers. |
| #include <iostream>  #include <random>  int main() {  std::random\_device rd;  std::mt19937 gen(rd());  std::uniform\_int\_distribution<> dis(1, 100); // Uniform distribution  int random\_number = dis(gen);  std::cout << "Random number: " << random\_number << "\n";  return 0;  } |

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

| **Principles(s):** Principle of High-Quality Randomness: This principle emphasizes the need for generating high-quality random numbers, which are crucial for applications requiring randomness, such as simulations, cryptographic operations, and gaming. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0 | Modernize-avoid-c-arrays | - |
| CppCheck | 2.6 | Misra-cpp2008-9\_3 | - |
| PVS-Studio | 7.15 | V2567 | - |
| Coverity | 2022.12 | CERT C++ MSC50-CPP | - |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Declarations and Initialization | STD-010-CPP | Obey the one-definition rule |

| **Noncompliant Code** |
| --- |
| Violating the one-definition rule can lead to linker errors and undefined behavior, especially when multiple definitions of the same entity exist across different areas. |
| // a.cpp  struct S {  int a;  };    // b.cpp  class S {  public:  int a;  }; |

| **Compliant Code** |
| --- |
| The compliant solution checks that entities are defined exactly once within a program, typically by declaring them in headers and defining them in implementation files. |
| // 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):** Principle of Single Definition: This principle ensures that each entity (such as a class, struct, or variable) is defined only once within a program to prevent conflicts, linker errors, and undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 12.0 | Misc-one-definition-rule | - |
| CppCheck | 2.6 | redundantAssignment | - |
| PVS-Studio | 7.15 | V3033 | - |
| Coverity | 2022.12 | MISRA C++ 2008 3-3-1 | - |

### 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.

By embedding security practices throughout the DevOps lifecycle, it evolves into DevSecOps. During the "Design" phase, threat modeling is introduced, and in the "Build" phase, both IDE and deployment security are prioritized. The "Verify & Test" phase incorporates static application security testing and automated security scans alongside unit, integration, and other testing activities.

Once the application is in production, automated testing continues with integrity checks and layered security measures to prevent breaches. Continuous threat detection is maintained through methods such as network monitoring, penetration testing, and performance log analysis. Just as QA testing is conducted early and frequently, security testing should follow the same principle.

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | Medium | High | Low | High | 2 |
| STD-003-CPP | Medium | High | Low | High | 2 |
| STD-004-CPP | High | Medium | Medium | High | 2 |
| STD-005-CPP | High | Medium | Medium | High | 2 |
| STD-006-CPP | High | Low | Low | Medium | 3 |
| STD-007-CPP | High | High | Medium | High | 3 |
| STD-008-CPP | Medium | Medium | Low | Medium | 3 |
| STD-009-CPP | Medium | High | Low | High | 2 |
| STD-010-CPP | High | Medium | Medium | High | 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 | Encryption in rest protects stored data. This may include hard drives, phones, computers, and cloud assets, among others. Protection of this data can be done through encryption tools, disk encryption and security for mobile devices and computers. |
| Encryption in flight | Encryption at flight is about protecting data that is moving. This can be between two devices within a network, or moving outside of a network. This can be protected through examples such as email encryption, DLP solutions, and solid network security features, such as firewalls and authentication. It is also important to consider the path data may be taking, and the security of this path. |
| Encryption in use | Encryption in use protects data that is created, edited, or otherwise defined as in-use. Protection of this data can be done by ensuring data control and protection exists prior to use, and in place in the first place. Managing access rights and identity will also minimize risk to this data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the act of confirming one’s identity. This can cover a range of types, but often are examples such as static passwords, one-time passwords, certifications, and biometric credentials. These forms of identification work to ensure a person is who they claim to be. |
| Authorization | Authorization specifies the access rights and privileges of a user, and are an important part of information and computer security. Where authentication confirms an identity, authorization determines what a user can and cannot access in the first place, limiting possible vulnerabilities when someone may interact with sensitive data they may not need to access, or the permissions one may have during access. |
| Accounting | Accounting is the process of keeping track of activity while interacting with a system, showing timestamps, accessed resources, and data transfer information. This is valuable in both creating a “bread crumb trail” in user activity, and also for the purposes of forensic analysis and investigation, should it be required. |

**\***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 | 8/08/2024 | Final Project One | Caio Mauro |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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