**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 | Ensuring that all input data is validated before processing prevents malicious data from entering the system. This includes checking for type, length, format, and range. Validating input data helps mitigate risks such as SQL injection, cross-site scripting (XSS), and buffer overflow attacks. |
| 1. Heed Compiler Warnings | Compiler warnings are indicators of potential issues in your code. Ignoring these warnings can lead to security vulnerabilities and unstable software. By addressing compiler warnings promptly, you can improve code quality and prevent potential security risks from being introduced into the codebase. |
| 1. Architect and Design for Security Policies | Security should be a fundamental part of the software design and architecture process. Incorporating security policies at the design stage ensures that security is built into the system from the ground up, rather than being an afterthought. This approach helps to identify and mitigate security risks early in the development lifecycle. |
| 1. Keep It Simple | Simple and straightforward code is easier to understand, test, and maintain. Complexity in code can lead to errors and obscure vulnerabilities. By keeping the design and implementation simple, you reduce the likelihood of introducing security flaws and make it easier to audit and review the code for security issues. |
| 1. Default Deny | The default deny principle ensures that access is restricted by default, and only explicitly permitted actions are allowed. This minimizes the attack surface by ensuring that only necessary permissions are granted, and all other actions are denied unless specifically allowed. It is a proactive approach to access control. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege involves granting users and systems only the minimum levels of access necessary to perform their functions. This reduces the risk of accidental or malicious misuse of privileges and limits the potential damage that can be caused by a security breach. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data before sending it to other systems ensures that potentially harmful data is cleaned or escaped to prevent injection attacks and other vulnerabilities. This is especially important when interacting with external systems, databases, or web services, where untrusted data could be exploited. |
| 1. Practice Defense in Depth | Defense in depth is a layered security strategy that employs multiple security controls to protect against threats. By using a combination of preventive, detective, and responsive measures, you create multiple barriers for attackers to overcome, thereby increasing the overall security of the system. |
| 1. Use Effective Quality Assurance Techniques | Applying rigorous quality assurance techniques, such as code reviews, static analysis, and automated testing, helps to identify and fix security issues early in the development process. These practices ensure that the code meets security standards and reduces the likelihood of vulnerabilities making it into production. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard provides developers with guidelines and best practices for writing secure code. By following established standards, you ensure consistency in the codebase and reduce the risk of introducing security vulnerabilities due to coding errors or unsafe practices. |

### 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 Appropriate Data Types** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Using appropriate data types ensures the correct handling of data, reducing the risk of overflows and type mismatches. |

| **Noncompliant Code** |
| --- |
| The use of an int type for a large number can cause overflow issues. |
| // Noncompliant  int length = 10000000000; // Potential overflow |

| **Compliant Code** |
| --- |
| Using long long ensures that large numbers are handled correctly. |
| // Compliant  long long length = 10000000000; // Correct type usage |

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

| **Principles(s):** **Validate Input Data**: Ensures correct handling of data types to prevent overflows and type mismatches. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 LTS | cpp | Detects potential overflow issues by ensuring appropriate data types are used. |
| Coverity | 2021.12 | TYPE\_OVERFLOW | Identifies and prevents overflow vulnerabilities by checking data types. |
| Fortify Static Code Analyzer | 21.1.2 | CTYPE\_OVERFLOW | Ensures that the appropriate data types are used to prevent overflows. |
| PVS-Studio | 7.12 | V1042 | Detects inappropriate data type usage that could lead to overflow. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Validate Data Values** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Validating data values ensures that only expected and safe data is processed, preventing invalid inputs from causing errors. |

| **Noncompliant Code** |
| --- |
| Not validating the input can result in invalid values being processed, leading to potential security issues or crashes. |
| // Noncompliant  int age = getInputAge(); // No validation |

| **Compliant Code** |
| --- |
| Validating the input ensures that it falls within a reasonable and expected range, thus preventing invalid values from causing issues. |
| // Compliant  int age = getInputAge();  if (age < 0 || age > 120) {  throw std::invalid\_argument("Invalid age value");  } |

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

| **Principles(s):** **Validate Input Data:** Ensures all input values are within expected ranges, preventing security issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 LTS | Cpp | Validates data ranges to ensure correct input values |
| Coverity | 2021.12 | INCONSISTENT\_RANGE | Identifies issues with data range validation |
| Fortify Static Code Analyzer | 21.1.2 | CVAL\_DATA | Ensures that data values are within expected ranges. |
| PVS-Studio | 7.12 | V601 | Detects improper validation of input values |

#### Coding Standard 3

| **Coding Standard** | **Label** | Use Safe String Functions |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Using safe string functions prevents buffer overflows and other string-related vulnerabilities. Unsafe string functions can lead to serious security issues, such as memory corruption and unauthorized access. |

| **Noncompliant Code** |
| --- |
| Using unsafe string functions like strcpy can result in buffer overflow if the source string exceeds the destination buffer size. |
| // Noncompliant  char buffer[10];  strcpy(buffer, "This is too long"); |

| **Compliant Code** |
| --- |
| Using safer string functions like strncpy and ensuring null-termination prevents buffer overflow and ensures the string fits within the buffer. |
| // Compliant  char buffer[10];  strncpy(buffer, "This is safe", sizeof(buffer) - 1);  buffer[9] = '\0'; // Ensure null-termination |

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

| **Principles(s):** **Adopt a Secure Coding Standard**. Prevents buffer overflows by using safe string functions. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Low | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify Static Code Analyzer | 21.1.2 | STR07-C | Detects unsafe string functions andrecommends safer alternatives |
| SonarQube | 8.9 LTS | Cpp | Identifies unsafe string functions like strcpy and recommends safer alternatives |
| Coverity | 2021.1.12 | BUFFER\_OVERFLOW | Ensures that string functions do not cause buffer overflows. |
| PVS-Studio | 7.12 | V512 | Detects unsafe string operations that could lead o buffer overflows |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Use Parameterized Queries** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Using parameterized queries prevents SQL injection attacks by safely handling user inputs. This ensures that user input is treated as data, not executable code, thereby preventing attackers from manipulating SQL queries. |

| **Noncompliant Code** |
| --- |
| Concatenating user input directly into SQL queries makes the code vulnerable to SQL injection. |
| // Noncompliant  std::string query = "SELECT \* FROM users WHERE username = '" + username + "'"; |

| **Compliant Code** |
| --- |
| Using parameterized queries ensures user input is treated as data, not code. |
| // Compliant  std::string query = "SELECT \* FROM users WHERE username = ?";  prepared\_statement->setString(1, username); |

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

| **Principles(s):** **Validate Input Data**: Prevents SQL injection by safely handling user inputs. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Checkmarx | 9.3 | CWE-89 | Identifies SQL injection vulnerabilities and recommends using parameterized queries. |
| Fortify Static Code Analizer | 21.1.2 | SQLI | Detects SQL injection vulnerabilities and ensures use of parameterized queries |
| SonarQube | 8.9 LTS | Cpp | Ensures the use of parameterized queries to prevent SQL injection. |
| Coverity | 2021.12 | SQL\_INJECTION | Identifies SQL injection risks and recommends secure coding practices |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Proper memory management prevents leaks and vulnerabilities related to improper memory use. By ensuring memory is allocated and freed correctly, we can avoid issues such as memory leaks and double free vulnerabilities. |

| **Noncompliant Code** |
| --- |

|  |
| --- |
| Double freeing memory can lead to undefined behavior and vulnerabilities. |
| // Noncompliant  char \*buffer = (char \*)malloc(10);  strcpy(buffer, "Hello");  free(buffer);  buffer = nullptr;  free(buffer); // Double free vulnerability |

| **Compliant Code** |
| --- |
| Ensuring the buffer is not freed twice prevents double free vulnerabilities. |
| // Compliant  char \*buffer = (char \*)malloc(10);  if (buffer != nullptr) {  strcpy(buffer, "Hello");  free(buffer);  buffer = nullptr;  } |

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

| **Principles(s):** **Sanitize Data Sent to Other Systems**: Ensures correct allocation and deallocation of memory. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgring | 3.17.0 | Memcheck | Detectcs memory management issues, including leaks and double free vulnerabilities |
| Coverity | 2021.12 | DOUBLE\_FREE | Identifies double free vulnerabilities and improper memory management |
| Fortify Static Code Analyzer | 21.1.2 | MEMPTR | Ensures proper memory allocation and deallocation |
| PVS-Studio | 7.12 | V730 | Detects improper memory management practices |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use Assertions to Enforce Assumptions** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Using assertions helps enforce assumptions and catch unexpected states during development. Assertions ensure that the program behaves as expected by validating critical assumptions at runtime. |

| **Noncompliant Code** |
| --- |
| Not validating the index can lead to out-of-bounds access. |
| // Noncompliant  int index = getIndex();  int array[10];  array[index] = 5; // No validation of index |

| **Compliant Code** |
| --- |
| The compliant code uses assertions to validate that the index is within bounds, preventing out-of-bounds access. |
| // Compliant  int index = getIndex();  assert(index >= 0 && index < 10);  int array[10];  array[index] = 5; |

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

| **Principles(s):** **Practice Defense in Depth**: Validates assumptions to prevent unexpected states. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.12 | V1006 | Detects code that can cause out-of-bounds access and ensures assertions are used properly. |
| Coverity | 2021.1.2 | ASSERTION\_FAILURE | Ensures that assertions are properly used to validate assumptions. |
| SonarQube | 8.9 LTS | Cpp | Identifies missing assertions in critical parts of the code |
| Fortify Static Code Analyzer | 21.1.2 | ASSERT | Validates the use of assertions to enforce assumption |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Use Exceptions for Error Handling** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Using exceptions ensures robust error handling and prevents the propagation of errors. Exceptions provide a structured way to handle errors and exceptional conditions in the program. |

| **Noncompliant Code** |
| --- |
| This code uses a return value to indicate an error, which can be easily ignored by the caller, leading to improper error handling. |
| // Noncompliant  int divide(int a, int b) {  if (b == 0) return -1; // Improper error handling  return a / b;  } |

| **Compliant Code** |
| --- |
| Using exceptions ensures proper handling of division by zero errors. |
| // Compliant  int divide(int a, int b) {  if (b == 0) throw std::invalid\_argument("Division by zero");  return a / b;  } |

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

| **Principles(s):** **Adopt a Secure Coding Standard**: Ensures robust error handling and prevents error propagation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverityy | 2021.12 | FORWARD\_NULL | Identifies improper error handling and recommends the use of exceptions |
| Fortify Static Code Analyzer | 21.1.2 | ERR\_HANDLING | Ensures proper use of exceptions for error handling. |
| SonarQube | 8.9 LTS | Cpp | Detects improper error handling practices nd recommends exceptions. |
| PVS-Studio | 7.12 | V1043 | Ensures robust error handing using exceptions |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Prevent Integer Overflow** |
| --- | --- | --- |
| Integer Overflow | STD-008-CPP | Preventing integer overflow ensures the integrity of calculations and prevents vulnerabilities. Integer overflow can lead to incorrect calculations and potential security issues. |

| **Noncompliant Code** |
| --- |
| This code does not check for overflow, which can result in incorrect calculations if the result exceeds the maximum value for an integer. |
| // Noncompliant  int result = a \* b; // Potential overflow |

| **Compliant Code** |
| --- |
| The compliant code checks for overflow using built-in functions, ensuring that calculations are safe and correct. |
| // Compliant  int result;  if (\_\_builtin\_mul\_overflow(a, b, &result)) {  throw std::overflow\_error("Integer overflow");  } |

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

| **Principles(s):** **Validate Input Data**: Prevents vulnerabilities by ensuring calculations are safe and correct. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 LTS | Cpp | Detects potential integer overflow issues and recommends proper handling. |
| Coverity | 2021.12 | INTEGER\_OVERFLOW | Identifies potential integer overflow vulnerabilities. |
| Fortify Static Code Analyzer | 21.1.2 | INT\_OVER | Ensures calculations are safe from integer overflow. |
| PVS-studio | 7.12 | V617 | Detects potential integer overflow issues in code. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Secure File I/O Operations** |
| --- | --- | --- |
| I/O Operations | STD-009-CPP | Secure file I/O operations prevent unauthorized access and ensure data integrity. Proper file handling ensures that files are accessed and modified securely, preventing data breaches. |

| **Noncompliant Code** |
| --- |
| This code does not check whether the file operation was successful, which can lead to data being written to an incorrect location or not being written at all. |
| // Noncompliant  std::ofstream file("data.txt");  file << "Sensitive data"; // No validation  file.close(); |

| **Compliant Code** |
| --- |
| The compliant code checks whether the file was successfully opened before performing write operations, ensuring that data is handled securely. |
| // Compliant  std::ofstream file("data.txt", std::ios::out | std::ios::app);  if (!file.is\_open()) {  throw std::runtime\_error("Cannot open file");  }  file << "Sensitive data";  file.close(); |

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

| **Principles(s):** **Sanitize Data Sent to Other Systems**: Ensures file operations are secure and validated. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify Static Code Analyzer | 21.1.2 | CWE-252 | Ensures all file operations are properly checked for success |
| Coverity | 2021.12 | FILE\_IO | Identifies insecure file I/O operations. |
| SonarQube | 8.9 LTS | Cpp | Ensures file handling is done securely |
| PVS-studio | 7.12 | V2006 | Detects insecure file I/O operations. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Use Secure Network Protocols** |
| --- | --- | --- |
| Network Communication | STD-010-CPP | Using secure network protocols ensures the confidentiality and integrity of data transmitted over the network. Secure communication prevents data from being intercepted and tampered with during transmission. |

| **Noncompliant Code** |
| --- |
| This code sends data over the network without encryption, making it susceptible to interception and eavesdropping. |
| // Noncompliant  send(socket, data, strlen(data), 0); // Unencrypted communication |

| **Compliant Code** |
| --- |
| The compliant code uses SSL/TLS to encrypt data before sending it over the network, ensuring secure communication. |
| // Compliant  SSL\_write(ssl, data, strlen(data)); // Encrypted communication |

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

| **Principles(s):** **Use Effective Quality Assurance Techniques**: Ensures network communication is secure. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Checkmarx | 9.3 | CWE-319 | Ensures the use of secure network protocols and identifies unencrypted communications |
| Fortify Static Code Analyzer | 21.1.2 | NET\_SECURITY | Ensures that network communications use secure protocols. |
| SonarQube | 8.9 LTS | Cpp | Identifies unencrypted communication. |
| Coverity | 2021.12 | NETWORK\_PROTOCOL | Ensures that network protocols are secure |

### 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.

Guidance on modifying the existing DevOps process to automate the enforcement of the standards in this policy includes integrating automated code analysis tools like SonarQube, Fortify, Checkmarx, and PVS-Studio into various stages of the DevSecOps pipeline. During the coding phase, these tools can be integrated into the IDE to detect vulnerabilities early. In the verification phase, static and dynamic analysis tools can identify potential issues, and in pre-production, resilience testing tools like Chaos Monkey and input fuzzing tools can test for edge cases and input validation issues. Finally, during the release and monitoring phases, tools like RASP and network monitoring solutions can provide real-time protection and detection of 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 | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Medium | Low | High | 4 |
| STD-003-CPP | High | High | Low | High | 5 |
| STD-004-CPP | High | High | Medium | High | 5 |
| STD-005-CPP | High | Medium | Medium | High | 4 |
| STD-006-CPP | Medium | Medium | Low | Medium | 3 |
| STD-007-CPP | High | Medium | Medium | High | 4 |
| STD-008-CPP | High | Medium | Medium | High | 4 |
| STD-009-CPP | High | Medium | Low | High | 4 |
| STD-010-CPP | High | High | Medium | High | 5 |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest protects data stored on disks or other storage mediums from unauthorized access. It ensures that data is encrypted when stored, making it unreadable without the decryption key. All sensitive data stored on Green Pace systems must be encrypted at rest using AES-256 encryption. This applies to databases, file systems, and backups. Encrypting data at rest mitigates the risk of data breaches and ensures compliance with data protection regulations. |
| Encryption in flight | Encryption in flight secures data transmitted over networks, preventing interception and eavesdropping. It ensures that data is encrypted during transmission, protecting its confidentiality and integrity. All data transmitted over public and private networks must be encrypted using TLS 1.2 or higher. This includes communications between internal systems and with external parties. Encrypting data in flight protects sensitive information from being intercepted during transmission, maintaining its confidentiality and integrity. |
| Encryption in use | Encryption in use protects data being processed or used by applications, ensuring it remains encrypted even while in memory. Sensitive data processed by applications must be encrypted in memory using runtime encryption technologies. This includes data accessed and processed by services and applications. Encrypting data in use provides an additional layer of security, protecting sensitive information from being exposed while being processed. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication ensures that users are who they claim to be before they are granted access to systems and data. All users must be authenticated using multi-factor authentication (MFA) before accessing Green Pace systems. This includes password-based authentication combined with a secondary factor such as a security token or biometric verification. Implementing MFA strengthens security by requiring multiple forms of verification, reducing the risk of unauthorized access. |
| Authorization | Authorization determines what an authenticated user is allowed to do within the system. Access controls must be enforced to ensure users have access only to the resources necessary for their role. Role-based access control (RBAC) should be implemented to manage permissions. Enforcing authorization policies ensures that users have the appropriate level of access based on their roles, reducing the risk of data exposure and unauthorized actions. |
| Accounting | Accounting involves tracking and logging user activities within the system to provide an audit trail for security and compliance purposes. All user actions must be logged, including logins, data access, and changes to the system. Logs should be reviewed regularly for suspicious activity. Maintaining detailed logs of user activities helps detect and respond to security incidents, ensuring accountability and compliance with regulations. |

**\***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 | 5/25/2024 | Updated Coding Standards | Blake Schmitt |  |
| 1.2 | 6/16/2024 | Updated Rest of Document | Blake Schmitt |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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