**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

# Revised by Shawn Ciaurro

## 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 | To verify that all inputs coming from external sources are in the expected format, information type and not dangerous to the system. Checking input prevents errors and memory corruption whether it is intentional or unintentional. Specifically, input validation can stop injection attacks and buffer overflow breaches. |
| 1. Heed Compiler Warnings | Warnings given by compilers often highlight unsafe code. Heeding the warnings and making adequate adjustments prevent vulnerabilities. |
| 1. Architect and Design for Security Policies | Following security policies during development makes for stronger and more secure systems than those designed with security amended on as an afterthought. Core principals will be applied through all stages of development and design enforcing authentication, access controls, data protection and secure coding practices. |
| 1. Keep It Simple | Overly complex code is harder to develop, maintain, and identify security vulnerabilities. Unnecessary features or complexities introduces more possible points of breaches that may be more difficult to locate. |
| 1. Default Deny | The default to any access point is to deny access. Decisions should be made to grant access after checking for permissions or appropriate actions. |
| 1. Adhere to the Principle of Least Privilege | All components should be designed with the least amount of permissions necessary for that component. With minimal permissions, in the case of a breach, a hacker will have access limited to those of the compromised component. |
| 1. Sanitize Data Sent to Other Systems | To maintain the security standards of this system, any data received from other systems should be checked and altered if necessary to remove any unsafe characters, commands or formatting issues. This prevents injection attacks, data incompatibility, and data corruption. |
| 1. Practice Defense in Depth | Multiple levels of security features make it harder to breach a system. If one layer of protection is overcame by a nefarious actor, another is still in place to keep the system safe. Balance of amounts of layers of security and complexity and user experience is essential. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance should be used for ensuring security as much as it is for functionality. Testing techniques such as code review, product testing, and static and dynamic analysis during development should be incorporated to ensure secure code. |
| 1. Adopt a Secure Coding Standard | An established, well-defined, secure coding standard should be adopted to guide the development process. By using proven and trusted guidelines, such as SEI CERT C++, common coding errors that can introduce vulnerabilities will be avoided. |

### 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 for Variable Intent** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Choosing the correct data type for a variable’s application is essential to prevent behavioral and functionality issues, data loss, and memory access errors. The wrong data type selection opens the system up to multiple vulnerabilities, a few including overflow/underflow vulnerabilities, truncations, and memory access vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This code declares a fixed sized character array which can result in a buffer overflow if userInput is >9 characters. |
| char name[10];  strcpy(name, userInput); |

| **Compliant Code** |
| --- |
| Using string data type handles its own memory allocation and is the more appropriate data type in this situation. It will auto size to accommodate the input, preventing overflow errors and reducing the risk for memory leaks and dangling pointers. |
| std::string name = userInput; // auto-managed memory |

| **Principles(s):** 1. **Validate Input Data,** 2. **Heed Compiler Warnings**, and 10. **Adopt a Secure Coding Standard. These principles support choosing the appropriate data type to ensure safe input, prevent compiler warnings and address them when they arise, and keep with secure coding standards.** |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 LTS | cpp:S1007 | Checks for incorrect types and flags inappropriate usage. |
| Clang Tidy | 17 | Cert-str31-c | Flags misused data types. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Check Data Values Before Use** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Values must be validating and sanitized before being processed in the system to prevent logical failures, improper memory access, division by zero, and other vulnerabilities. |

| **Noncompliant Code** |
| --- |
| In the below example, the data value is not checked and validated before use. It is unsafe to use a raw user provided value without validating that it will not crash the program or introduce vulnerabilities. |
| int numPirates = getUserInput();  int result = totalLoot / numPirates; // may crash if denominator is 0 |

| **Compliant Code** |
| --- |
| To comply to the data value check standard, user input values should be passed through some kind of test to ensure they are valid. In this case, the value is not processed if it will lead to division by zero or breaks the logic of amount of a countable object (by using a negative number of pirates.) |
| int numPirates = getUserInput();  if (numPirates > 0) {  int result = totalLoot / numPirates;  std::cout << "Each pirate gets " << result << " gold coins.\n";  } else {  std::cerr << "Error: Number of pirates must be greater than zero.\n";  } |

| **Principles(s):** 1. **Validate Input Data,** 4. **Keep It Simple**, and 10. **Adopt a Secure Coding Standard. Logic correctness is maintained with validating values and avoiding overly complex logic reduces risk of errors. Secure coding standards are met when employing these principles.** |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.14 | divisionByZero | Checks lack of input validation to ensure unsafe operations are flagged. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Proper String Handling to Avoid Buffer Overflows** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Strings handled incorrectly can lead to buffer overflows providing an opportunity for attackers to exploit memory spaces. By keeping strings in expected bounds with safe string operations, unexpected behaviors are avoided. |

| **Noncompliant Code** |
| --- |
| This code has a fixed sized buffer but does not check for length. This can lead to overflow if the user enters an unexpectedly long amount of characters. This type of action can lead to program crashes and open the door for code injection and memory corruption. |
| char shipName[10];  strcpy(shipName, userInput); // risk of overflow if userInput > 9 chars |

| **Compliant Code** |
| --- |
| A compliant version of the ship name operation uses strncpy to safely copy the name while accounting for the buffer size. Only 9 character will be copied with an explicitly set terminator guaranteeing success. |
| char shipName[10];  strncpy(shipName, userInput, sizeof(shipName) - 1);  shipName[9] = '\0'; // manually null-terminate |

| **Principles(s):** 1. **Validate Input Data,** 8. Practice Defense in Depth, and 10. **Adopt a Secure Coding Standard. Overflows are avoided by validating input and safe string handling while adding a layer of defense. Secure coding standards are met when employing these principles.** |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 17 | security.insecureAPI.strcpy | Detects operations that can lead to overflow. |
| Fortify SCA | 23.1 | Buffer Overflow | Static analysis for buffer overrun paths through input vectors. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Parameterized Queries to Prevent SQL Injection** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Using raw user input in situations where input is used for SQL queries invites injection attacks. To avoid this, prepared statements should be used to treat input as data rather than executable SQL. |

| **Noncompliant Code** |
| --- |
| The below code allows users to modify the query at their will. This is dangerous because with input such as “Jack'; DROP TABLE TreasureLogs;--” the query becomes:  SELECT \* FROM TreasureLogs WHERE Captain = 'Jack'; DROP TABLE TreasureLogs;--'  Resulting in an injection attack, dumping the table. |
| std::string pirateName = getUserInput();  std::string query = "SELECT \* FROM TreasureLogs WHERE Captain = '" + pirateName + "';";  db.execute(query); |

| **Compliant Code** |
| --- |
| The below code complies with the standard by using a prepared statement to safely bind the user input with a placeholder rather than interpreting it as code. This prevents malicious code from being executed. |
| std::string pirateName = getUserInput();  PreparedStatement stmt = db.prepare("SELECT \* FROM TreasureLogs WHERE Captain = ?");  stmt.bind(1, pirateName);  stmt.execute(); |

| **Principles(s):** 1. **Validate Input Data,** 7. **Sanitize Data Sent to Other Systems**, and 10. **Adopt a Secure Coding Standard. The biggest defense in preventing SQL injections is input validation and sanitation. Secure coding standards are met when employing these principles.** |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify SCA | 23.1 | SQL Injection | Tracks input as SQL statements. |
| SonarQube | 9.9 LTS | Cpp:S2077 | Warns in dangerous dynamic SQL queries. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Preventing Memory Leaks and Handling Free Memory** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | To properly manage memory, it must be ensured that freed memory is handled in a way to prevent reuse to avoid exploits of dangling pointers. |

| **Noncompliant Code** |
| --- |
| The code below deletes treasureMap but the declared pointer is still pointing at the freed memory space. Using it again can cause undefined behavior. |
| char\* treasureMap = new char[100];  strcpy(treasureMap, "X marks the spot!");  delete[] treasureMap;  std::cout << treasureMap << std::endl; // use-after-free |

| **Compliant Code** |
| --- |
| The compliant version of this process frees the memory and sets the pointer to nullptr, preventing accidental reuse. |
| char\* treasureMap = new char[100];  strcpy(treasureMap, "X marks the spot!");  delete[] treasureMap;  treasureMap = nullptr; // pointer nullified |

| **Principles(s):** 3. Architect and Design for Security Policies**,** 6. **Adhere to the Principle of Least Privilege**, and 10. **Adopt a Secure Coding Standard. Designing with memory safety in mind will limit privilege misuses, layer protection and secure memory handling. Secure coding standards are met when employing these principles.** |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.21 | Memcheck | Detects memory leaks and similar memory problems at runtime. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Enforce Preconditions and Invariants with Assertions** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Use of assertions helps identify logic errors by enforcing assumptions about the program’s state. This helps signal bugs before they can become vulnerabilities. It is important to note that this does not replace input validation but helps along side it. |

| **Noncompliant Code** |
| --- |
| The below code accepts the user’s input for a heading but doesn’t confirm that it is valid. |
| int compass[4] = {0, 90, 180, 270}; // North, East, South, West  int direction = getUserInput(); // User enters direction index  int heading = compass[direction]; |

| **Compliant Code** |
| --- |
| In compliant code, the assertion is set what to expect from the user so that it can be made clearly known when an invalid value is entered. |
| int compass[4] = {0, 90, 180, 270}; // North, East, South, West  int direction = getUserInput(); // User enters direction index  assert(direction >= 0 && direction < 4); // declares expected vals  int heading = compass[direction]; |

| **Principles(s):** 1. **Validate Input Data,** 9. Use Effective Quality Assurance Techniques, and 10. **Adopt a Secure Coding Standard. Input validity is enforced by assertions which improves the QA of the product. Secure coding standards are met when employing these principles.** |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Tidy | 17 | cert-oop54-cpp | Maintains logic invariants and ensures inputs are in expected bounds. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Use Targeted Exception Catching for Safe Error Recovery** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Proper exception handling is important to preserve the control flow and maintain stability. General exception catching using catch(…)does not reveal the root cause of the error. Better diagnostics can be achieved by using specific exception types. |

| **Noncompliant Code** |
| --- |
| In this code example, a catch all exception is used giving no context to what went wrong. |
| try {  loadTreasureMap("map.txt");  } catch (...) {  std::cerr << "Something went wrong. Abandon ship!\n";  } |

| **Compliant Code** |
| --- |
| The correct version of the code catches specific error types and outputs a message that details what went wrong. |
| try {  loadTreasureMap("map.txt");  } catch (const std::ifstream::failure& e) {  std::cerr << "Map file could not be loaded: " << e.what() << '\n';  } catch (const std::runtime\_error& e) {  std::cerr << "Runtime error while decoding treasure coordinates: " << e.what() << '\n';  } |

| **Principles(s):** 3. Architect and Design for Security Policies**,** 9. **Use Effective Quality Assurance Techniques**, and 10. **Adopt a Secure Coding Standard. Structured exception handling should be built which will also help to identify bugs and is part of standard secure coding.** |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 LTS | Cpp:S3626 | Warns when generic exceptions are used instead of specific. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Close Files and Free Resources to Prevent Leaks** |
| --- | --- | --- |
| File and Resource Handling | STD-008-CPP | Memory leaks can be caused by the failure to close files. Other issues can also occur such as data corruption and file locks. Proper cleanup ensures the availability of resources for future uses. |

| **Noncompliant Code** |
| --- |
| Here the file is never closed introducing the possibility of corruption and memory leaks. |
| FILE\* log = fopen("ledger.txt", "r");  if (log) {  // Read from the log...  // does not fclose(log)  } |

| **Compliant Code** |
| --- |
| The file is properly closed using fclose(), ensuring the issues of a file left open are avoided. |
| FILE\* log = fopen("ledger.txt", "r");  if (log) {  // Read from the log...  fclose(log); // properly close log  } |

| **Principles(s):** 5. **Default Deny,** 6. **Adhere to the Principle of Least Privilege**, and 10. **Adopt a Secure Coding Standard. When not in use, resources permission should be set to deny, minimizing unnecessary access privileges. Secure coding standards are met when employing these principles.** |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.14 | resourceLeak | A static check for unclosed files/unfreed memory. |
| Coverity | 2024.06 | RESOURCE\_LEAK | Detects unclosed files. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Limit Use of System Calls and Shell Execution** |
| --- | --- | --- |
| System Calls and Shell Execution | STD-009-CPP | System calls and shell execution are risky practices due to the possibility of functions such as system(), popen(), and exec() being vulnerable to exploits via malicious shell command injection. Safer alternatives can be used such as platform specific APIs like std::filesystem and file streams. |

| **Noncompliant Code** |
| --- |
| Passing user input into the command presents the opportunity for injection. |
| std::string filename = getUserInput();  std::string command = "cat " + filename;  system(command.c\_str()); |

| **Compliant Code** |
| --- |
| In the safer version of this code, the user input is passed through ifstream(), removing the shell involvement and direct commands. This prevents injection opportunities. |
| std::string filename = getUserInput();  std::ifstream file(filename);  if (file.is\_open()) {  std::string line;  while (std::getline(file, line)) {  std::cout << line << '\n';  }  } else {  std::cerr << "Couldn’t open the logbook!\n";  } |

| **Principles(s):** 3. Architect and Design for Security Policies**,** 4. **Keep It Simple**, 7. Sanitize Data Sent to Other Systems, and 10. **Adopt a Secure Coding Standard. By avoiding shell calls, code is kept simple and prevents injections, keeping with secure code practices.** |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify SCA | 23.1 | Command Injection | Checks for unsanitized strings passed into the system/shell |
| SonarQube | 9.9 LTS | Cpp:S2674 | Warns against unvalidated input used in command system. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Clear Out Sensitive Data After Use** |
| --- | --- | --- |
| Sensitive Data Handling | STD-010-CPP | Sensitive data like passwords or security keys may remain in memory after use, leaving them vulnerable to unauthorized access. Clearing or overwriting these values will make for more secure handling of the sensitive data. |

| **Noncompliant Code** |
| --- |
| The following code takes a sensitive piece of data and deletes it after use but fails to clear it from memory. |
| char\* secret = new char[100];  strcpy(secret, "Treasure location: under the tree on Smoky Beach");  // …code using secret  delete[] secret; // secret remains in memory |

| **Compliant Code** |
| --- |
| This code overwrites the secret so that even if the memory location can be accessed, the sensitive data is gone. |
| char\* secret = new char[100];  strcpy(secret, " Treasure location: under the tree on Smoky Beach");  // …code using secret  std::fill(secret, secret + 100, 0); // clear sensitive data  delete[] secret; |

| **Principles(s):** 3. Architect and Design for Security Policies**,** 6. **Adhere to the Principle of Least Privilege**, 8. Practice Defense in Depth, and 10. **Adopt a Secure Coding Standard. Sensitive data should be handled in a secure manner, only accessible to roles necessary, and cleared when not needed.** |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2024.06 | SECURE\_TEMP, UNINIT | Checks for sensitive data still in memory and not deleted. |

### Defense-in-Depth Illustration

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



## Project One

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

### Revise the C/C++ Standards

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

### Risk Assessment

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

### Automated Detection

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

### Automation



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy.

Using the automated tools such as SonarQube, Cppcheck, Clang-Tidy, Fortify SCA, etc., makes it so that issues are caught early and throughout all of the stages of the development lifecycle. Catching possible risky or unsafe code early reduces the chances of the errors being built upon or making their way into the final product. During development of a project, automated plugins and IDE warnings provide instant feedback on developing code when then can be static tested with cppcheck or similar automated tests. Through the pre-production phases, Valgrind, Fortify SCA and similar tools can detect vulnerabilities by simulating runtime conditions. On the production side, security continues with tools that check that tamper-free code is being run safely. Integrity checkers and policy enforcement tools automatically check that the running system matches the predefined configurations. When tools detect problems, the system need to respond and adapt to the issue with tools like WAF shields and automatic rollbacks triggered by dangerous events. Scans made at regular intervals will collect data that informs new updates and produces a more secure system over time.

### 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 | Likely | Low | High | 4 |
| STD-003-CPP | High | Likely | Medium | High | 5 |
| STD-004-CPP | High | Likely | Low | High | 5 |
| STD-005-CPP | Medium | Likely | Medium | Medium | 3 |
| STD-006-CPP | Medium | Likely | Low | Medium | 3 |
| STD-007-CPP | Medium | Likely | Low | Medium | 3 |
| STD-008-CPP | Medium | Likely | Low | Medium | 3 |
| STD-009-CPP | High | Likely | Low | High | 5 |
| STD-010-CPP | High | Unlikely | Medium | High | 4 |

### 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 refers to securing stored data whether it be in databases or on hard drives. This is important because a bad actor can gain access to a database or a physical means of storage but still not be able to access the sensitive data without the decryption key. AES-256 encryption will be the Green Pace standard for encryption for all data stored at rest. |
| Encryption in flight | This type of encryption applies to data in transit from one system to another. Protecting data in transit by TLS 1.2 or higher will protect inter-system communications from attacks such as eavesdropping and man in the middle attacks. Places this will be necessary include login forms, third-party integrations and transmissions of sensitive data. |
| Encryption in use | In use encryption deals with data that is in use in RAM. Policy should be that data is immediately cleared after use to prevent exposure of data through process inspection. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is used to verify the identity of a party attempting to gain access to a system. Policy for Green Pace systems will be multi-factor authentication for user logins and an additional layer of authentication for admin roles to access sensitive parts of the system. Logging all login attempts is used to audit activity. |
| Authorization | Authorization is the concept of levels of user access for controls and authentication walls in a system. Role based access controls will be the Green Pace policy paired with the principle of least privileges for each role. Access to administrative functions, files and databases (CRUD functions) will be reserved for roles that require access with changes in user roles and new user addition going through an approval process. |
| Accounting | Accounting is the process of tracking activity and access to audit and inspect to ensure compliance to security standards. This process is important to check that adherence to the standards is kept and to review actions to improve security in updates. |

**\***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 | 07/20/2025 | Milestone 1 | Shawn Ciaurro |  |
| 1.2 | 08/10/2025 | Finalized Form | Shawn Ciaurro |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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