**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 | Always check user input to make sure it’s safe and in the correct format. This helps prevent security issues like hackers injecting harmful code. For example, validating that an email input only contains allowed characters. |
| 1. Heed Compiler Warnings | Pay attention to warnings from the compiler, as they often point out possible security problems. Fixing these warnings can help prevent bugs and vulnerabilities. |
| 1. Architect and Design for Security Policies | Build security into the system from the start instead of adding it later. A well-planned security approach helps keep data safe. |
| 1. Keep It Simple | The more complex a system is, the easier it is to have security flaws. Keeping things simple makes it easier to spot and fix problems. For instance, using a single method for authentication rather than multiple, redundant methods reduces errors. |
| 1. Default Deny | By default, block all access unless it is specifically allowed. This ensures that only approved users and processes can access sensitive data. |
| 1. Adhere to the Principle of Least Privilege | Only give users and applications the minimum access they need to do their job. This limits damage if an account is hacked. For example, a web server should only have access to the files it needs and not to the entire system. |
| 1. Sanitize Data Sent to Other Systems | Before sending data to another system, clean it to remove anything harmful. This prevents attacks like code injections. |
| 1. Practice Defense in Depth | Use multiple layers of security so that if one fails, others still protect the system. This makes it harder for attackers to break in. |
| 1. Use Effective Quality Assurance Techniques | Test software carefully using tools and reviews to find security flaws before they become a problem. |
| 1. Adopt a Secure Coding Standard | Follow well-known coding rules that focus on security. This helps create safer and more reliable software. |

### 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 | Choosing the right data type is essential to avoid unexpected behavior, overflow, or underflow. Using an incorrect data type can result in data loss, security vulnerabilities, or crashes. By selecting the appropriate data type, developers ensure the system handles data safely and efficiently. |

| **Noncompliant Code** |
| --- |
| Incorrect use of an int to store a large number that exceeds its limit. |
| int value = 3000000000; // Value exceeds maximum range of int |

| **Compliant Code** |
| --- |
| Using a long long to store large values safely. |
| long long value = 3000000000LL; // Correct data type to store large values |

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

| **Principles(s):** Validate Input Data – Picking the right data type helps to make sure the input is safe and will not break the program. Preventing numbers being too big or not fitting properly for example. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High – If the wrong data type is used, it can lead to crashes, overflows, or serious security bugs. | Unlikely – It doesn’t happen all the time, but it can easily be missed in large codebases. | Medium – Not terrible, would require some time to find and fix across the code. | High – Should be fixed early to prevent bigger issues later. | 2 – SEI CERT rates this as a level 2 threat based on risk. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3 | cpp:S121 | Detects suspicious type conversions that can cause data loss or overflow. Helps ensure the correct data types are used during assignments and operations. |
| clang-tidy | 17 | clang-analyzer-core.CallAndMessage | Identifies unsafe function calls and data handling issues, including incorrect type usage and out-of-bounds access. Useful for catching violations of proper data type use and range safety. |

#### 

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Validating data values ensures that inputs remain within expected ranges. Without proper validation, out-of-range values can lead to unexpected behavior, crashes, or security vulnerabilities. Ensuring data is checked against defined limits protects system stability and security. |

| **Noncompliant Code** |
| --- |
| This code tries to write data beyond the allocated size of the array, which can lead to buffer overflow or unpredictable program behavior. |
| int arr[10];  arr[15] = 5; // Accessing out-of-bounds index |

| **Compliant Code** |
| --- |
| This code checks that the index is within valid bounds before assigning a value, preventing buffer overflow and ensuring data integrity. |
| int arr[10];  if (index >= 0 && index < 10) {  arr[index] = 5;  } |

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

| **Principles(s):** Validate Input Data - Making sure values are within allowed ranges so the program doesn’t crash or act out of normal. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High – Out-of-range values can cause crashes, buffer overflows, or unexpected behavior. | Somewhat Often – Happens often if input isn’t checked or values come from users. | Medium – You’ll need to add checks and maybe adjust logic, but it’s not terrible. | High – Needs to be handled early to avoid bugs and security issues. | 3 – More common than data type issues, and just as risky. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3 | cpp:S3518 | Detects cases where an index could be out of bounds, helping prevent buffer overflows and value-related bugs. |
| clang-tidy | 17 | clang-analyzer-core.UndefinedBinaryOperatorResult | Flags undefined results from math with unvalidated inputs. Helps catch situations where values aren't checked before use. |

#### 

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Handling strings correctly is crucial to prevent buffer overflows and memory corruption. Improper string manipulation can lead to serious security vulnerabilities, such as buffer overflow attacks. Ensuring string operations respect buffer limits reduces the risk of data corruption. |

| **Noncompliant Code** |
| --- |
| This code copies a string that is larger than the allocated buffer, causing a buffer overflow and possible memory corruption. |
| char str[10];  strcpy(str, "This is too long"); // Buffer overflow risk |

| **Compliant Code** |
| --- |
| This code uses strncpy() to copy a string safely and ensures the string is null-terminated, preventing buffer overflow and ensuring safe string handling. |
| char str[10];  strncpy(str, "Safe", sizeof(str) - 1);  str[sizeof(str) - 1] = '\0'; |

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

| **Principles(s):** Validate Input Data - Making sure strings don’t go past the buffer and mess up memory or crash the program. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High – Bad string handling can cause buffer overflows and even let attackers run code. | Somewhat Often – Happens a lot if string lengths aren’t checked right. | Medium – You need to fix how strings are handled, maybe use safer functions. | High – Needs to be fixed right away to avoid serious bugs or attacks. | 3 – Pretty common and dangerous if not caught. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3 | cpp:S3518 | Detects buffer overflows by checking if array or string access is out of bounds. |
| Coverity Static Analysis | 2023.6 | STRING\_OVERFLOW | Finds places where string copies or operations may write past buffer limits, helping prevent overflows and crashes. |

#### 

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | SQL injection occurs when user inputs are inserted directly into SQL statements without proper validation. This can allow attackers to manipulate the query and gain unauthorized access to the database. Using parameterized queries or prepared statements prevents SQL injection by ensuring that user inputs are treated as data, not executable code. |

| **Noncompliant Code** |
| --- |
| This code inserts user input directly into the query, allowing malicious users to modify the SQL statement and compromise the database. |
| std::string query = "SELECT \* FROM users WHERE name = '" + user\_input + "'"; |

| **Compliant Code** |
| --- |
| This code uses parameterized queries, ensuring that user input is treated as data, preventing SQL injection, and protecting the database from malicious manipulation. |
| std::string query = "SELECT \* FROM users WHERE name = ?";  stmt->setString(1, user\_input); |

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

| **Principles(s):** Validate Input Data - Making sure user input doesn’t mess with the database or change the query on accident. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High – Can let attackers steal data, change records, or break the whole database. | Somewhat Often – Happens if user input isn’t handled the right way. | Low – Just need to use parameterized queries or prepared statements. | High – Should always be fixed right away to protect the database. | 4 – One of the most serious and well-known types of attacks. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3 | cpp:S2077 | Detects potential SQL injection vulnerabilities by flagging dynamic queries that include unchecked user input. |
| Checkmarx CxSAST | 9.6 | SQL\_Injection | Scans for unsafe SQL string construction and suggests fixes like using parameterized queries. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Proper memory management is essential to prevent memory leaks, buffer overflows, and dangling pointers. Failing to free allocated memory or accessing freed memory can lead to crashes or exploitable vulnerabilities. Secure coding practices help ensure that memory is properly allocated, used, and released. |

| **Noncompliant Code** |
| --- |
| This code allocates memory dynamically but does not release it, leading to a memory leak that could eventually exhaust system resources. |
| int\* ptr = new int[10]; |

| **Compliant Code** |
| --- |
| This code correctly frees the dynamically allocated memory, preventing memory leaks and ensuring that system resources are properly managed. |
| int\* ptr = new int[10];  delete[] ptr; |

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

| **Principles(s):** Practice Defense in Depth - Having more than one layer of checks makes sure memory is handled safely, even if something slips through. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High – Memory leaks or buffer issues can crash the program or open it up to attacks. | Somewhat Often – Happens when memory isn’t freed, or is used after being deleted. | Medium – You might need to refactor how memory is managed, especially in big codebases. | High – Should be fixed right away to avoid long-term stability or security problems. | 3 – Pretty common and can be dangerous if left unchecked. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.21.0 | Memcheck | Detects memory leaks, invalid reads/writes, and use of freed memory during runtime. |
| clang-tidy | 17 | clang-analyzer-cplusplus.NewDeleteLeaks | Identifies memory that was allocated with new but never released with delete, helping avoid leaks. |

#### 

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions are used to validate assumptions made by the program during execution. They help catch logic errors and incorrect assumptions early in development. However, over-reliance on assertions for runtime checks can introduce vulnerabilities in production if assertions are disabled or ignored. |

| **Noncompliant Code** |
| --- |
| This code uses an assertion to check for division by zero, but if assertions are disabled in production, this check is bypassed, leading to undefined behavior. |
| void divide(int a, int b) {  assert(b != 0); // Assumes b is never zero  int result = a / b;  } |

| **Compliant Code** |
| --- |
| This code explicitly checks for division by zero and throws an exception, ensuring that the error is handled properly even in production environments. |
| void divide(int a, int b) {  if (b == 0) {  throw std::invalid\_argument("Division by zero.");  }  int result = a / b;  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques - Use testing and real error checks instead of relying on assert, especially since assert can be turned off. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium – If assert is used wrong, it can hide bugs or crash the program when checks are skipped. | Somewhat Often – Easy to overuse assert or forget it doesn’t run in production. | Low – Usually just means replacing assert with proper error handling. | Medium – It’s not always critical, but fixing it improves stability and reliability. | Can lead to issues, but usually not the worst-case type. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 17 | readability-assert-usage | Flags overuse of assert() and suggests better error handling methods for production code. |
| Coverity Static Analysis | 2023.6 | ASSERT\_SIDE\_EFFECT | Detects if assert() is used with expressions that have side effects or critical logic, which could be skipped if assertions are disabled. |

#### 

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Proper exception handling ensures that errors are caught and managed effectively, preventing program crashes and potential security vulnerabilities. Incorrect use of exceptions can lead to resource leaks, inconsistent program states, or exposure of sensitive information. |

| **Noncompliant Code** |
| --- |
| This code catches all exceptions without distinguishing between them, making it difficult to identify and handle specific errors properly. |
| try {  // Code that may throw exceptions  riskyOperation();  } catch (...) {  std::cout << "An error occurred." << std::endl;  } |

| **Compliant Code** |
| --- |
| This code catches specific exceptions and handles them appropriately, providing better error management and maintaining application stability. |
| try {  riskyOperation();  } catch (const std::invalid\_argument& e) {  std::cerr << "Invalid argument: " << e.what() << std::endl;  } catch (const std::runtime\_error& e) {  std::cerr << "Runtime error: " << e.what() << std::endl;  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques - Helps make sure errors are caught the right way instead of just hiding them with a catch-all. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium – Bad exception handling can hide real problems or leave the app in a bad state. | Somewhat Often – Easy to use catch-all blocks or forget to handle specific errors. | Low – Fixing usually means updating try-catch blocks and adding proper checks. | Medium – Not always urgent, but important for clean and safe error handling. | 2 – Can cause issues if ignored, but usually won’t lead to major exploits. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3 | cpp:S3626 | Flags catch-all exception handlers (catch(...)) that don’t log or rethrow, which can hide real errors. |
| clang-tidy | 17 | cppcoreguidelines-avoid-catch-all | Warns when catch(...) is used instead of handling specific exception types. |

#### 

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Error Handling | STD-008-CPP | Effective error handling prevents unexpected behavior and ensures that errors are managed in a controlled manner. Poor error handling can lead to system crashes, inconsistent data, or exposure of sensitive information. Proper error handling improves system stability and security. |

| **Noncompliant Code** |
| --- |
| This code exposes system-level error details, which can leak sensitive information and aid attackers in identifying vulnerabilities. |
| FILE\* file = fopen("data.txt", "r");  if (!file) {  std::cout << "Error: " << strerror(errno) << std::endl; // Reveals system details  } |

| **Compliant Code** |
| --- |
| This code provides a generic error message, preventing the exposure of system details while still informing the user about the error. |
| FILE\* file = fopen("data.txt", "r");  if (!file) {  std::cout << "An error occurred while opening the file." << std::endl;  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques - Helps make sure errors are handled the right way and don’t show system info that attackers could use. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High – Exposing system details can help attackers target vulnerabilities. | Somewhat Often – Developers often forget to sanitize error messages or handle them properly. | Low – Fixing usually means just sanitizing error messages and handling exceptions properly. | High – This should be fixed immediately to avoid exposing sensitive information. | 4 – Serious risk of exposing details that can aid attackers. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3 | cpp:S1130 | Flags error messages that are too generic or that might expose sensitive system details. |
| Coverity Static Analysis | 2023.6 | UNCAUGHT\_EXCEPT | Detects exceptions that are thrown but never caught, which could crash the program or leak system state. |

#### 

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Cryptographic Practices | STD-009-CPP | Proper use of cryptographic algorithms protects sensitive data from unauthorized access. Using outdated or weak encryption methods can expose data to attackers. Following established cryptographic standards ensures the confidentiality, integrity, and authenticity of sensitive information. |

| **Noncompliant Code** |
| --- |
| This code uses a simple and weak encryption mechanism that is easy to break, exposing sensitive data to attackers. |
| std::string encrypt(std::string data) {  for (char &c : data) {  c += 1; // Simple and insecure encryption  }  return data;  } |

| **Compliant Code** |
| --- |
| This code uses a standard cryptographic algorithm (AES) from a trusted library, ensuring that sensitive data is securely encrypted. |
| #include <openssl/aes.h>  void encryptData(const unsigned char\* data, unsigned char\* encryptedData, const AES\_KEY\* key) {  AES\_encrypt(data, encryptedData, key);  } |

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

| **Principles(s):** Adopt a Secure Coding Standard - Using trusted encryption libraries instead of trying to make your own or using weak ones. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High – Weak or custom encryption can be broken, exposing sensitive data. | Somewhat Often – Developers sometimes use outdated algorithms or roll their own crypto. | Medium – May require replacing libraries and reworking how data is secured. | High – Needs to be fixed to protect confidentiality and integrity. | 4 – A common and serious issue if encryption isn’t done right. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3 | cpp:S5547 | Flags use of weak or outdated cryptographic algorithms like MD5 or SHA-1. |
| Checkmarx CxSAST | 9.6 | Insecure\_Crypto | Detects insecure or custom encryption methods and recommends stronger, vetted alternatives. |

#### 

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | STD-010-CPP | Proper management of system resources, such as memory, file handles, and network connections, prevents resource leaks, crashes, and performance degradation. Failing to release resources after use can exhaust system resources, making the application vulnerable to denial-of-service (DoS) attacks. |

| **Noncompliant Code** |
| --- |
| This code opens a file but fails to close it, leading to a resource leak that may exhaust available file handles. |
| FILE\* file = fopen("data.txt", "r");  if (file) {  // Process file content  // No fclose() leads to a resource leak  } |

| **Compliant Code** |
| --- |
| This code ensures that the file is properly closed after use, preventing resource leaks and maintaining system stability. |
| FILE\* file = fopen("data.txt", "r");  if (file) {  // Process file content  fclose(file); // Properly closing the file  } |

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

| **Principles(s):** Practice Defense in Depth - Making sure resources like files and memory are cleaned up so they don’t pile up or break the system. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High – Not releasing resources can crash the app or lock up the system. | Somewhat Often – Easy to forget to close files or free memory, especially in complex code. | Medium – Might need to update logic or add cleanup in more places. | High – Needs to be fixed early to avoid bigger system issues or leaks. | 3 – It happens often and can cause real damage if ignored. |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 17 | clang-analyzer-cplusplus.NewDeleteLeaks | Detects memory leaks from new/delete mismatches and missing deallocation. |
| Valgrind | 3.21.0 | Memcheck | At runtime, finds memory leaks, use-after-free errors, and unclosed file descriptors. |

### 

### Defense-in-Depth Illustration

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



## Project One

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

### Revise the C/C++ Standards

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

### Risk Assessment

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

### Automated Detection

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

### Automation

Provide a written explanation using the image provided.



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

Automation will be used to check code for security issues as it's being written and before it’s deployed. Tools like SonarQube, clang-tidy, and Valgrind will run during development, testing, and integration to catch problems early—like memory leaks, unsafe input, or bad error handling. These tools should be built into the DevSecOps pipeline so that every time code is committed or tested, checks happen automatically. This helps keep code safe, fast, and consistent without slowing down the team.

### 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 | Somewhat Often | Medium | High | 3 |
| STD-003-CPP | High | Somewhat Often | Medium | High | 3 |
| STD-004-CPP | High | Somewhat Often | Low | High | 4 |
| STD-005-CPP | High | Somewhat Often | Medium | High | 3 |
| STD-006-CPP | Medium | Somewhat Often | Low | Medium | 2 |
| STD-007-CPP | Medium | Somewhat Often | Low | Medium | 2 |
| STD-008-CPP | High | Somewhat Often | Low | High | 4 |
| STD-009-CPP | High | Somewhat Often | Medium | High | 4 |
| STD-010-CPP | High | Somewhat Often | 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 | This means encrypting data when it's stored on a disk or database. It protects info if someone gets access to the physical storage or backups. This policy should be used for files, logs, or databases that hold sensitive data. |
| Encryption in flight | This protects data while it's being sent over a network. It stops attackers from reading or changing the data during transfer. This policy should be used when sending info between apps, APIs, or to the cloud. |
| Encryption in use | This protects data while it's being processed in memory. It keeps the info safe from other programs or users on the system. This policy should be used for sensitive data during runtime, especially in shared environments. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | This is about making sure users are who they say they are. It usually involves usernames and passwords, but can also use multi-factor methods. The policy should apply to all systems where users log in or access protected resources. |
| Authorization | This controls what users are allowed to do after they log in. It decides what data or actions each person can access. The policy should apply to all systems to make sure users only get access to what they actually need. |
| Accounting | This tracks what users do in the system. It logs actions like logins, file access, or changes to data. This policy should be used to audit activity, find security issues, and support investigations if something goes wrong. |

**\***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

STD-001-CPP

Principle(s): 1

Explanation: Picking the right data type helps keep input safe and prevents overflow.

STD-002-CPP

Principle(s): 1

Explanation: Keeps values in a safe range so they don’t break the program.

STD-003-CPP

Principle(s): 1

Explanation: Stops strings from going past the buffer and messing up memory.

STD-004-CPP

Principle(s): 1

Explanation: Makes sure user input doesn’t mess with the database or change queries.

STD-005-CPP

Principle(s): 8

Explanation: Layers of checks help catch memory problems before they crash the system.

STD-006-CPP

Principle(s): 9

Explanation: Encourages real testing instead of relying on assert, which can be skipped.

STD-007-CPP

Principle(s): 9

Explanation: Makes sure errors are handled right and not just hidden.

STD-008-CPP

Principle(s): 9

Explanation: Helps make sure error messages don’t show system info and are handled properly.

STD-009-CPP

Principle(s): 10

Explanation: Uses trusted, secure code instead of weak or custom encryption.

STD-010-CPP

Principle(s): 8

Explanation: Having cleanup in place helps protect the system if something else fails.

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 | 4/11/2025 | Completed Security Policy | Christopher Gaunt |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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