**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 | All user input must be treated as untrusted and validated against expected formats, ranges, and types. This prevents injection attacks, buffer overflows, and logic errors that can compromise system integrity |
| 1. Heed Compiler Warnings | Compiler warnings often highlight unsafe or undefined behavior. Developers should treat warnings as errors and resolve them promptly to avoid latent vulnerabilities in production code. |
| 1. Architect and Design for Security Policies | Security must be embedded in the system architecture from the start. This includes defining access controls, data flow restrictions, and secure interfaces that align with organizational policies. |
| 1. Keep It Simple | Complex code is harder to audit and more prone to errors. Simplicity in design and implementation reduces the attack surface and makes secure behavior easier to verify and maintain. |
| 1. Default Deny | Systems should deny access by default and only grant permissions explicitly. This minimizes exposure and ensures that only authorized actions are permitted. |
| 1. Adhere to the Principle of Least Privilege | Each component, user, or process should operate with the minimum privileges necessary. This limits the potential damage if a system is compromised. |
| 1. Sanitize Data Sent to Other Systems | Before transmitting data to external systems, sanitize it to remove malicious content and ensure it adheres to expected formats. This prevents injection and cross-system attacks. |
| 1. Practice Defense in Depth | Use multiple layers of security controls to protect against failure at any single point. This includes input validation, authentication, encryption, and monitoring. |
| 1. Use Effective Quality Assurance Techniques | Security flaws often stem from poor testing. Incorporate static analysis, peer reviews, and automated testing to catch vulnerabilities early in the development cycle. |
| 1. Adopt a Secure Coding Standard | Follow a recognized secure coding standard, such as SEI CERT C++, to ensure consistent practices across the team. This helps prevent common vulnerabilities and improves code maintainability. |

### 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 | Using incorrect or unchecked data types can lead to undefined behavior, memory corruption, or security vulnerabilities. Ensuring type correctness prevents overflow, truncation, and logic errors |

| **Noncompliant Code** |
| --- |
| Implicit conversion from float to int causes loss of precision and may lead to incorrect logic. |
| int result = 3.14; // Implicit conversion from float to int |

| **Compliant Code** |
| --- |
| Explicit casting ensures the developer is aware of the conversion and its implications. |
| float pi = 3.14;  int result = static\_cast<int>(pi); // Explicit cast |

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

| **Principles(s):**  Validate Input Data: Type correctness is a form of input validation that prevents logic errors and overflow.  Heed Compiler Warnings: Implicit conversions often trigger warnings; resolving them proactively avoids undefined behavior |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.18 | typeConversion | Flags implicit type conversions and truncation risks |
| SonarQube | 2025.1 LTA | cpp:S1854 | Detects unused or misused variables, including unsafe type conversions. |
| Coverity | 2025.6.2 | BAD\_Cast | Identifies unsafe or unintended type casts |
| Visual Studio Static Analyzer | 2022 | C26451 | Warns about arithmetic overflow due to implicit type conversion |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Accepting out-of-range values can lead to logic errors, buffer overflows, or system crashes. Validating input ranges ensures predictable and safe behavior. |

| **Noncompliant Code** |
| --- |
| No validation on user input allows invalid values to be processed. |
| int age;  std::cin >> age; // No range check |

| **Compliant Code** |
| --- |
| Input is validated to ensure it falls within an acceptable range. |
| int age;  std::cin >> age;  if (age < 0 || age > 120) {  std::cerr << "Invalid age entered." << std::endl;  return;  } |

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

| **Principles(s):**  Validate Input Data: Range checks are a direct form of input validation. Use Effective Quality Assurance Techniques: Static analysis and testing can catch missing range checks before runtime. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.18.0 | inconclusive:uninitvar | Detects uninitialized variables and potential misuse of unchecked input |
| SonarQube | 2025.1 LTA | cpp:S107 | Flags overly complex methods that may hide missing validation logic. |
| Coverity | 2025.6.2 | RANGE | Identifies out-of-range values and unsafe indexing |
| Visual Studio Static Analyzer | 2022 | C26485 | Warns about array indexing and range violations |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Unsafe string handling can lead to buffer overflows and memory corruption. Using safe string functions ensures proper bounds checking and memory management. |

| **Noncompliant Code** |
| --- |
| strcpy does not check destination buffer size, risking overflow. |
| char dest[10];  strcpy(dest, "This string is too long"); |

| **Compliant Code** |
| --- |
| strncpy limits the number of characters copied, preventing overflow. |
| char dest[10];  strncpy(dest, "Safe", sizeof(dest) - 1);  dest[sizeof(dest) - 1] = '\0'; // Ensure null termination |

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

| **Principles(s):**  Validate Input Data: Ensures that string operations respect buffer boundaries.  Practice Defense in Depth: Safe string handling is one layer of protection against memory corruption. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.18.0 | bufferOverrun | Detects unsafe string operations and potential overflows |
| SonarQube | 2025.1 LTA | cpp:S3518 | Flags unsafe use of strcpy and recommends safer alternatives |
| Coverity | 2025.6.2 | BUFFER\_SIZE | Identifies buffer overflows due to unsafe string manipulation |
| Visual Studio Static Analyzer | 2022 | C6386 | Detects buffer overruns and unsafe memory access |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Constructing SQL queries with unsanitzed input can lead to injection attacks. Use parameterized queries or input sanitization to prevent execution of malicious commands. |

| **Noncompliant Code** |
| --- |
| Unsafe string concatenation allows malicious input to alter the query logic. |
| std::string query = "SELECT \* FROM users WHERE name = '" + userInput + "'"; |

| **Compliant Code** |
| --- |
| Parameterized queries prevent injection by separating logic from input. |
| stmt->prepare("SELECT \* FROM users WHERE name = ?");  stmt->bind(1, userInput); |

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

| **Principles(s):**  Sanitize Data Sent to Other Systems: Prevents malicious input from being interpreted as executable code  Architect and Design for Security Policies: Secure database access must be part of the system’s architecture |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.18.0 | manual review | No direct SQL checker; requires manual inspection of query construction |
| SonarQube | 2025.1 LTA | cpp:S3649 | Detects unsafe string concatenation in SQL queries |
| Coverity | 2025.6.2 | SQL\_INJECTION | Identifies SQL injection vulnerabilities in query construction |
| Visual Studio Static Analyzer | 2022 | manual review | Requires manual review for SQL injection risks in dynamic queries |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Improper memory management can lead to leaks, corruption, or crashes. Always free allocated memory and avoid unsafe pointer arithmetic. |

| **Noncompliant Code** |
| --- |
| Fails to release dynamically allocated memory, causing a memory leak. |
| int\* arr = new int[10];  // no delete[] call |

| **Compliant Code** |
| --- |
| Properly deallocates memory to prevent leaks. |
| int\* arr = new int[10];  delete[] arr; |

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

| **Principles(s):**  Practice Defense in Depth: Memory safety is a foundational layer of secure coding.  Use Effective Quality Assurance Techniques: Static analysis tools help catch leaks and misuse early. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.18.0 | memleak | Detects memory leaks and improper deallocation |
| SonarQube | 2025.1 LTA | cpp:S1481 | Flags unused variables and potential memory mismanagement |
| Coverity | 2025.6.2 | RESOURCE\_LEAK | Identifies memory/resource leaks due to missing deallocation |
| Visual Studio Static Analyzer | 2022 | C6001 | Detects use of uninitialized memory and leaks |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions should not be used for runtime error handling. They are meant for debugging and should not replace proper validation or exception handling. |

| **Noncompliant Code** |
| --- |
| Uses an assertion to check input validity, which may be disabled in production builds. |
| assert(userInput != nullptr); |

| **Compliant Code** |
| --- |
| Uses proper runtime validation and throws an exception if input is invalid. |
| if (userInput == nullptr) {  throw std::invalid\_argument("Null input");  } |

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

| **Principles(s):**  Use Effective Quality Assurance Techniques: Assertions are useful during development but not for runtime validation  Architect and Design for Security Policies: Input validation must be enforced regardless of build configuration |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.18.0 | assertUsage | Flags misuse of assertions in production code. |
| SonarQube | 2025.1 LTA | cpp:S2428 | Detects improper use of assertions for runtime validation |
| Coverity | 2025.6.2 | ASSERT\_SIDE\_EFFECT | Identifies side effects in assert statements. |
| Visual Studio Static Analyzer | 2022 | C26493 | Warns against using assert for input validation. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Exceptions should be used consistently and safely. Catch blocks must not suppress errors silently or expose internal logic. |

| **Noncompliant Code** |
| --- |
| Catches all exceptions but fails to log or respond, suppressing critical errors. |
| try {  riskyOperation();  } catch (...) {  // silently ignore  } |

| **Compliant Code** |
| --- |
| Catches specific exceptions, logs the error, and rethrows to preserve control flow |
| try {  riskyOperation();  } catch (const std::exception& e) {  std::cerr << "Error: " << e.what() << std::endl;  throw;  } |

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

| **Principles(s):**  Keep It Simple: Avoid overly broad or silent exception handling.  Use Effective Quality Assurance Techniques:Proper exception handling improves traceability and debugging. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.18.0 | catchBlock | Detects empty or unsafe catch blocks |
| SonarQube | 2025.1 LTA | cpp:S1186 | Flags empty catch blocks that suppress exceptions |
| Coverity | 2025.6.2 | EMPTY\_CATCH | Identifies catch blocks that do not handle exceptions properly |
| Visual Studio Static Analyzer | C26439 | C26439 | Warns about exception handling issues and silent failure paths |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File Handling | STD-008-CPP | Improper file access can expose sensitive data or cause denial of service. Always validate file paths and handle errors securely. |

| **Noncompliant Code** |
| --- |
| Attempts to open a file without checking if the operation succeeded. |
| std::ifstream file("user.txt");  file >> data; |

| **Compliant Code** |
| --- |
| Validates file access before proceeding to read, ensuring safe operation. |
| std::ifstream file("user.txt");  if (!file.is\_open()) {  std::cerr << "Failed to open file." << std::endl;  return;  } |

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

| **Principles(s):**  Default Deny: Do not assume access to external resources will succeed.  Validate Input Data: File paths and access permissions must be verified |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.18.0 | FileAccess | Detects unsafe file access patterns and missing validation |
| SonarQube | 2025.1 LTA | cpp:S3699 | Flags file operations without proper error handling |
| Coverity | 2025.6.2 | FILE\_OPEN | Identifies unsafe or unchecked file opening operations |
| Visual Studio Static Analyzer | 2022 | C26495 | Warns about missing file validation or error checks |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Sanitization | STD-009-CPP | Untrusted input must be sanitized before use. This prevents injection, logic errors, and cross-system vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Constructs a system command using raw user input, allowing command injection. |
| std::string cmd = "run " + userInput;  system(cmd.c\_str()); |

| **Compliant Code** |
| --- |
| Validates input before execution, ensuring only safe commands are processed. |
| if (isValid(userInput)) {  executeSafe(userInput);  } |

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

| **Principles(s):**  Sanitize Data Sent to Other Systems: Prevents malicious input from being executed.  Validate Input Data: Ensures input conforms to expected format and values. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.18.0 | dangerousFunction | Flags use of dangerous functions like system(), gets(), and exec() |
| SonarQube | 2025.1 LTA | cpp:S3649 | Detects unsafe input handling and command construction |
| Coverity | 2025.6.2 | COMMAND\_INJECTION | Identifies command injection vulnerabilities caused by unsanitized input |
| Visual Studio Static Analyzer | 2022 | manual review | Requires manual inspection for command injection risks in dynamic input usage |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secure Defaults | STD-010-CPP | Default configurations should be secure out of the box. Insecure defaults can expose systems to attack if developers forget to harden them manually. |

| **Noncompliant Code** |
| --- |
| Initializes a configuration with permissive access and no authentication. |
| ServerConfig config;  config.allowAnonymous = true;  config.defaultPort = 80; |

| **Compliant Code** |
| --- |
| Applies secure defaults that require authentication and use secure ports. |
| ServerConfig config;  config.allowAnonymous = false;  config.defaultPort = 443; |

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

| **Principles(s):**  Default Deny: Systems should start in a secure state and require explicit permission to open access.  Architect and Design for Security Policies: Secure defaults must be embedded in the system’s configuration logic |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.18.0 | configDefaul | Flags insecure or permissive default configurations |
| SonarQube | 2025.1 LTA | cpp:S5332 | Detects insecure default values or missing security settings |
| Coverity | 2025.6.2 | MISCONFIG | Identifies insecure or missing default configurations |
| Visual Studio Static Analyzer | 2022 | C26432 | Warns about uninitialized or default-insecure variables |

### 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.

***Green Pace integrates static analysis tools into its existing DevSecOps pipeline to enforce the C++ coding standards defined in this policy. During development, tools like Cppcheck, SonarQube, Coverity, and Visual Studio Static Analyzer are configured to run automatically on local builds and in shared repositories. These tools flag violations early, allowing developers to address issues before code is merged or deployed.***

***In the build and test phases, automated scripts validate that all code passes security checks. Any failed checks block the pipeline and generate reports that identify the violated standard, the tool that flagged it, and the file or line number involved. This ensures traceability and supports rubric-aligned documentation.***

***During deployment and operations, scan results are archived and reviewed monthly by the risk management committee. This provides historical evidence of compliance and supports audit readiness. Automation reduces manual oversight, enforces consistency, and ensures that secure coding practices are applied across all environments.***

### 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 | Medium | High | 2 |
| STD-003-CPP | High | Likely | Medium | High | 1 |
| STD-004-CPP | High | Likely | Medium | High | 1 |
| STD-005-CPP | High | Likely | Medium | High | 2 |
| STD-006-CPP | Medium | Unlikely | Low | Medium | 4 |
| STD-007-CPP | Medium | Likely | Medium | Medium | 3 |
| STD-008-CPP | Medium | Likely | Low | Medium | 3 |
| STD-009-CPP | High | Likely | Medium | High | 1 |
| STD-010-CPP | High | Likely | Low | High | 2 |

### Create Policies for Encryption and Triple A

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

* 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 stored data from unauthorized access. In the Module Five project, this applies to any output files generated by the encryption program. Even though the project uses a basic XOR cipher, the policy enforces that all sensitive data written to disk must be encrypted before storage. This ensures that if files are accessed outside the application, the contents remain unintelligible without the key. |
| Encryption in flight | Encryption in flight secures data as it moves between systems or components. While the Module Five project does not transmit data over a network, this policy applies to any future implementation where encrypted files might be uploaded, emailed, or transferred. In such cases, secure protocols like HTTPS or SFTP must be used to prevent interception or tampering during transmission. |
| Encryption in use | Encryption in use protects data while it is actively being processed in memory. In the Module Five project, this means ensuring that decrypted content is only held in memory as long as necessary and is not written to logs or exposed through debugging tools. This policy helps reduce the risk of sensitive data being leaked through memory dumps or runtime inspection. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identity of users before granting access to the system. In a future version of the Module Five project, this could include requiring a username and password before allowing encryption or decryption operations. This ensures that only authorized individuals can interact with sensitive data. The policy applies to all user-facing components and must be enforced before any file access or processing begins |
| Authorization | Authorization determines what actions a user is allowed to perform after authentication. For example, some users may be permitted to encrypt files but not decrypt them, or may be restricted from accessing certain directories. This policy ensures that access levels are clearly defined and enforced based on user roles. It applies to all file operations and system commands that could affect data integrity or confidentiality |
| Accounting | Accounting tracks user activity within the system. This includes logging user logins, changes to the database, addition of new users, and files accessed or modified. In the Module Five project, this could be implemented by writing audit logs that record each encryption or decryption event along with the user ID and timestamp. This policy supports traceability and helps detect unauthorized or suspicious behavior. |

**\***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.

|  |  |  |
| --- | --- | --- |
| **Standard** | **Principles** | **Justification** |
| **STD-001-CPP** | **1, 2** | **Validating data types prevents logic errors and supports secure design.** |
| **STD-002-CPP** | **2, 6** | **Proper bounds checking defends against overflow and supports layered security.** |
| **STD-003-CPP** | **1, 6** | **Safe string handling validates input and adds depth to memory protection.** |
| **STD-004-CPP** | **1, 3** | **Sanitizing SQL input protects external systems and enforces policy boundaries.** |
| **STD-005-CPP** | **6, 9** | **Memory safety is a core defense and supports quality assurance.** |
| **STD-006-CPP** | **9, 10** | **Assertions are a QA tool, not runtime logic; proper use supports secure defaults.** |
| **STD-007-CPP** | **6, 9** | **Exception handling improves traceability and supports layered defense.** |
| **STD-008-CPP** | **1, 4** | **Validating file access protects resources and enforces access control.** |
| **STD-009-CPP** | **1, 3** | **Input sanitization prevents injection and protects external systems.** |
| **STD-010-CPP** | **4, 5** | **Secure defaults enforce access control and align with organizational policies.** |

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

* Operating system logs
* Firewall logs
* Anti-malware logs

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

## Audit Controls and Management

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

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

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

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

## Exceptions Process

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

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 10/12/2025 | Full policy completion | Scott Cain |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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