**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 | Ensures that all input data is verified against expected formats and constraints. This prevents injection attacks, buffer overflows, and unexpected behaviors by rejecting malformed or malicious input. |
| 1. Heed Compiler Warnings | Compiler warnings often indicate potential vulnerabilities or logical errors in code. Addressing these warnings improves code quality and reduces the likelihood of security flaws. |
| 1. Architect and Design for Security Policies | Incorporating security into the architecture and design phase helps identify and mitigate risks early, reducing vulnerabilities and ensuring compliance. This proactive approach establishes a secure foundation for the system, minimizing attack surfaces and future breaches. |
| 1. Keep It Simple | Simple code is easier to audit, debug, and maintain. Complexity often leads to unnoticed vulnerabilities and errors that attackers can exploit. |
| 1. Default Deny | By default, deny access to resources unless explicitly allowed. This principle minimizes attack surfaces and ensures access control adheres to security requirements. |
| 1. Adhere to the Principle of Least Privilege | Users and processes should have only the permissions necessary to complete their tasks. This limits the impact of accidental or malicious actions. |
| 1. Sanitize Data Sent to Other Systems | Data passed between systems should be validated and sanitized to ensure integrity and prevent injection attacks, particularly in SQL and command-line contexts. |
| 1. Practice Defense in Depth | Employs multiple layers of security controls to protect against vulnerabilities at different stages. If one layer fails, additional layers provide fallback protection. |
| 1. Use Effective Quality Assurance Techniques | Employ robust testing methods, such as static analysis, dynamic analysis, and fuzz testing, to identify and resolve security vulnerabilities during development. |
| 1. Adopt a Secure Coding Standard | Following established coding standards like SEI CERT C++ Coding Standard ensures consistency and helps avoid known vulnerabilities and coding pitfalls. |

### 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 insufficient data types can result in overflow errors, data loss, or undefined behaviors. Choosing appropriate data types ensures compatibility, accuracy, and security. |

| **Noncompliant Code** |
| --- |
| This example shows an integer overflow vulnerability |
| int value = 2147483647; // Maximum value for 32-bit int  value += 1; // Causes overflow |

| **Compliant Code** |
| --- |
| This example avoids overflow by using a larger data type. |
| long value = 2147483647LL; // Use larger data type  value += 1; // Handles larger range safely |

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

| **Principles(s):** 1: Validate Input Data  Ensures that the correct data type is used helps prevent overflow errors, data loss, and undefined behaviors, which could lead to security vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.16 (Latest) | Type conversion | Provides unique code analysis to detect bugs and focuses on detecting undefined behavior and dangerous coding constructs. |
| Clang-Tidy | 21.0.0 (Latest) | cert-str34-c | Detects improper data type usage, ensuring secure handling of values. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Restricting input values to valid ranges prevents invalid operations and errors. Data values should be validated during input and use. |

| **Noncompliant Code** |
| --- |
| This example demonstrates unvalidated input causing an invalid operation. |
| int divisor = 0;  int result = 100 / divisor; // Division by zero |

| **Compliant Code** |
| --- |
| This example ensures input validation before performing the operation. |
| int divisor = 0;  if (divisor != 0) {  int result = 100 / divisor;  } else {  std::cerr << "Error: Division by zero!" << std::endl;  } |

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

| **Principles(s):** 1: Validate Input Data  Ensures that the correct data type is used helps prevent overflow errors, data loss, and undefined behaviors, which could lead to security vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | Latest | S2083 | Identifies missing boundary checks and unsafe input handling. |
| CppCheck | 2.16 (Latest) | Out of bounds | Provides unique code analysis to detect bugs and focuses on detecting undefined behavior and dangerous coding constructs. |
| Clang-Tidy | 21.0.0 (Latest) | cert-str34-c | Detects improper data type usage, ensuring secure handling of values. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Strings should be properly managed to avoid buffer overflows, memory leaks, and undefined behavior. |

| **Noncompliant Code** |
| --- |
| This example shows a buffer overflow vulnerability with unsafe string copying. |
| char buffer[10];  strcpy(buffer, "This is a very long string"); // Exceeds buffer size |

| **Compliant Code** |
| --- |
| This example uses safer string functions to prevent buffer overflows. |
| char buffer[10];  strncpy(buffer, "Short", sizeof(buffer) - 1);  buffer[sizeof(buffer) - 1] = '\0'; // Ensure null termination |

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

| **Principles(s):** 7: Sanitize Data Sent to Other Systems  Ensures that user input and transmitted data are properly sanitized and validated before being processed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | Latest | S3655 | Detects out-of-bounds string access, which can lead to crashes and security vulnerabilities. |
| CppCheck | 2.16 (Latest) | Out of bounds | Provides unique code analysis to detect bugs and focuses on detecting undefined behavior and dangerous coding constructs. |
| Clang-Tidy | 21.0.0 (Latest) | cert-str34-c | Detects improper data type usage, ensuring secure handling of values. |
| SonarQube | Latest | S5883 | Ensures that user-provided strings are properly sanitized before being used in critical operations. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Preventing SQL injection protects databases from unauthorized access and data breaches. |

| **Noncompliant Code** |
| --- |
| This example demonstrates using unescaped user input in an SQL query. |
| std::string query = "SELECT \* FROM users WHERE username = '" + username + "';";  executeQuery(query); |

| **Compliant Code** |
| --- |
| This example uses parameterized queries to prevent SQL injection. |
| std::string query = "SELECT \* FROM users WHERE username = ?;";  PreparedStatement pstmt = connection.prepareStatement(query);  pstmt.setString(1, username);  pstmt.executeQuery(); |

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

| **Principles(s):** 1: Validate Input Data, 7: Sanitize Data Sent to Other Systems  Validating input and sanitizing database input ensure a bad actor can’t alter SQL query structures. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | Latest | S3649 | Detects unsafe SQL query construction, preventing SQL injection. |
| SonarQube | Latest | S2077 | Identifies unvalidated user input in database queries. |
| CppCheck | 2.16 (Latest) | SQL injection | Flags concatenated SQL statements that introduce vulnerabilities. |
| Clang-Tidy | 21.0.0 (Latest) | cert-str34-c | Ensures safe string manipulation to prevent SQL injection risks. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Proper memory management prevents leaks, dangling pointers, and undefined behavior. |

| **Noncompliant Code** |
| --- |
| This example demonstrates failure to release dynamically allocated memory. |
| int\* ptr = new int(10); |

| **Compliant Code** |
| --- |
| This example uses smart pointers to manage memory safely. |
| std::unique\_ptr<int> ptr = std::make\_unique<int>(10); |

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

| **Principles(s):** 8: Practice Defense in Depth, 10: Adopt a Secure Coding Standard  Both principles apply as secure coding practices and multiple security layers are critical for memory protection. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | Latest | S2259 | Detects null pointer dereferences, preventing memory access violations. |
| SonarQube | Latest | S3519 | Identifies improper memory management practices, such as missing deallocation. |
| CppCheck | 2.16 (Latest) | Memory leak | Flags dynamically allocated memory that is never freed. |
| Clang-Tidy | 21.0.0 (Latest) | cert-mem57-cpp | Ensures proper resource deallocation and memory safety. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions are used to detect and diagnose conditions that should never occur. |

| **Noncompliant Code** |
| --- |
| This example omits assertions, allowing invalid states to go unchecked. |
| int value = -1; // Value should be non-negative  // No assertion to verify this |

| **Compliant Code** |
| --- |
| This example uses assertions to verify assumptions. |
| int value = -1;  assert(value >= 0); // Terminates program if false |

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

| **Principles(s):** 2: Heed Compiler Warnings, 9: Use Effective Quality Assurance Techniques  Assertions improve quality assurance and complement compiler warnings to detect logical errors early. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | Latest | S5683 | Detects missing assertions in critical logic, ensuring proper validation. |
| CppCheck | 2.16 (Latest) | Assertions | Identifies assertions that can be improved or misused. |
| Clang-Tidy | 21.0.0 (Latest) | cert-err33-c | Ensures assertions are correctly implemented without side effects. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Exceptions should be used to handle errors in a structured and recoverable manner. |

| **Noncompliant Code** |
| --- |
| This example does not handle exceptions, leading to program crashes. |
| int result = std::stoi("invalid"); // Throws an exception if conversion fails |

| **Compliant Code** |
| --- |
| This example handles exceptions to ensure program stability. |
| try {  int result = std::stoi("invalid");  } catch (const std::invalid\_argument& e) {  std::cerr << "Invalid input: " << e.what() << std::endl;  } |

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

| **Principles(s):** 3: Architect and Design for Security Policies, 9: Use Effective Quality Assurance Techniques  These principles apply because effective exception handling is essential for robust system architecture and quality assurance. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | Latest | S3824 | Detects exceptions that are caught but not handled properly. |
| SonarQube | Latest | S1948 | Flags exceptions that do not inherit from std::exception, ensuring proper exception hierarchy. |
| CppCheck | 2.16 (Latest) | Exception handling | Identifies uncaught exceptions and improper exception propagation. |
| Clang-Tidy | 21.0.0 (Latest) | cert-err52-cpp | Ensures exceptions are properly caught and handled without exposing security risks. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Sanitization | STD-008-CPP | Sanitizing input ensures malicious data cannot exploit vulnerabilities in the application. |

| **Noncompliant Code** |
| --- |
| This example processes untrusted input without sanitization. |
| std::string input = "<script>alert('XSS');</script>";  display(input); |

| **Compliant Code** |
| --- |
| This example sanitizes input before processing. |
| std::string input = sanitize("<script>alert('XSS');</script>");  display(input); |

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

| **Principles(s):** 8: Practice Defense in Depth, 10. Adopt a Secure Coding Standard  These principles apply because secure coding standards and layered security mechanisms ensure robust randomness. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | Latest | S2245 | Detects insecure PRNG usage, such as std::rand(), which should be replaced with CSPRNGs. |
| CppCheck | 2.16 (Latest) | Insecure randomness | Identifies weak random number generators that lack cryptographic security. |
| Clang-Tidy | 21.0.0 (Latest) | cert-msc51-cpp | Ensures secure random number generation, recommending std::random device or secure libraries. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Error Logging | STD-009-CPP | Logging errors securely helps diagnose issues without exposing sensitive information. |

| **Noncompliant Code** |
| --- |
| This example logs sensitive information, increasing security risk. |
| std::cerr << "Error: " << password << std::endl; // Exposes sensitive data |

| **Compliant Code** |
| --- |
| This example logs generic error messages to prevent leaks. |
| std::cerr << "Error: Invalid credentials" << std::endl; |

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

| **Principles(s):** 3: Architect and Design for Security Policies, 5: Default Deny  These principles apply because secure logging should be an intentional part of system design, with restricted access by default. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | Latest | S4792 | Detects logging of sensitive data, ensuring secure error reporting. |
| SonarQube | Latest | S2076 | Identifies logs that expose database error messages, which can lead to SQL injection risks. |
| CppCheck | 2.16 (Latest) | Logging misuse | Flags logging practices that may disclose sensitive system details. |
| Clang-Tidy | 21.0.0 (Latest) | cert-err33-c | Ensures safe error logging without exposing private information. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secure Random Number Generation | STD-010-CPP | Secure random numbers are essential for cryptography and other security-related tasks. |

| **Noncompliant Code** |
| --- |
| This example uses an insecure random number generator. |
| int randomNumber = rand(); |

| **Compliant Code** |
| --- |
| This example uses a secure random number generator, Mersenne Twister engine which provide better randomness compared to the standard rand() function. |
| std::random\_device rd;  std::mt19937 gen(rd());  std::uniform\_int\_distribution<> dis(1, 100);  int randomNumber = dis(gen); |

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

| **Principles(s):** 8: Practice Defense in Depth, 10: Adopt a Secure Coding Standard  These principles apply because secure coding standards and layered security mechanisms ensure robust randomness. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | Latest | S2245 | Detects insecure PRNG usage, such as std::rand(), which should be replaced with CSPRNGs. |
| CppCheck | 2.16 (Latest) | Insecure randomness | Identifies weak random number generators that lack cryptographic security. |
| Clang-Tidy | 21.0.0 (Latest) | cert-msc51-cpp | Ensures secure random number generation, recommending std::random\_device or secure libraries. |

### 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 can proactively detect vulnerabilities, enforce secure coding standards, and maintain system integrity without slowing down development by integrating security automation at various stages of the software development lifecycle. Security automation should be implemented during the Assess and Plan stage to analyze potential threats and ensure compliance with security policies before development begins. During Design and Build, security checks should be embedded into the continuous integration/continuous deployment (CI/CD) pipeline to automatically enforce secure coding practices and prevent noncompliant code from being integrated into production. The Verify and Test phase should include automated security testing, such as static and dynamic code analysis, vulnerability scanning, and compliance validation, ensuring that security flaws are detected and mitigated before deployment.

In the production phase, automation should be used in Transition and Health Check to verify that security configurations, access controls, and system settings remain compliant with defined policies. Continuous monitoring and detection should be implemented to analyze system logs, identify anomalies, and detect potential security threats in real time. If a security incident occurs, automated response mechanisms should be in place to contain threats, execute rollback procedures, and restore systems to a secure state. Automated compliance enforcement and security baselines should also be maintained in the Maintain and Stabilize phase to prevent unauthorized configuration changes and ensure ongoing adherence to security policies.

### 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 | Medium | Medium | High | 3 |
| STD-002-CPP | High | High | Medium | High | 4 |
| STD-003-CPP | High | High | Medium | High | 4 |
| STD-004-CPP | Critical | High | Medium | Highest | 5 |
| STD-005-CPP | High | Medium | Medium | High | 4 |
| STD-006-CPP | Medium | Medium | Low | Medium | 3 |
| STD-007-CPP | High | Medium | Medium | High | 4 |
| STD-008-CPP | High | Medium | Medium | High | 4 |
| STD-009-CPP | High | Medium | Medium | High | 4 |
| STD-010-CPP | High | Medium | 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 | Encrypting stored data ensures protection against unauthorized access in case of data breaches or physical theft. Data stored in databases, file systems, and backups must use AES-256 encryption, and encryption keys must be securely managed using HSMs or key vaults. This policy applies to all sensitive data storage to comply with GDPR, HIPAA, and PCI-DSS. |
| Encryption in flight | Encrypting data in transit prevents interception and man-in-the-middle (MITM) attacks. All network communications must use TLS 1.3 or higher, ensuring that transmitted data remains confidential and unaltered. This policy applies to web traffic, API communication, email encryption, and VPNs, ensuring secure transmission over untrusted networks. |
| Encryption in use | Encrypting data while it is being processed in memory protects against RAM scraping and insider threats. Technologies like Intel SGX, AMD SEV, and homomorphic encryption must be used for secure computing. This policy applies to cloud-based environments, financial transactions, and AI/ML processing to ensure confidentiality during computations. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Verifying user identity prevents unauthorized access and account compromise. All authentication mechanisms must use multi-factor authentication (MFA), biometrics, or strong passwords, and centralized authentication systems like SSO or OAuth 2.0 should be enforced. This policy applies to all user, service, and device logins to protect sensitive systems from unauthorized access. |
| Authorization | Controlling access based on user roles ensures that individuals only have the minimum privileges necessary to perform their tasks. Role-Based Access Control (RBAC) or Attribute-Based Access Control (ABAC) must be used, with just-in-time access approval for critical systems. This policy applies to databases, APIs, cloud services, and privileged accounts to prevent insider threats and privilege escalation. |
| Accounting | Logging and monitoring all user activities ensure auditability and early detection of security incidents. SIEM tools collect and analyze logs in real-time, and logs should be tamper-proof and stored securely for compliance purposes. This policy applies to all administrative actions, authentication attempts, and system changes, ensuring accountability and forensic capabilities. |

**\***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 | 01/26/2025 | Initial Template | Orion J. Murray |  |
| 2.0 | 02/16/2025 | Section Updates | Orion J. Murray | [Insert text.] |
| [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 |