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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Ensuring that all inputs, especially from untrusted sources, are validated is crucial to preventing security vulnerabilities. By validating inputs, we reduce the risk of attacks from external data sources, which can otherwise exploit software weaknesses. |
| 1. Heed Compiler Warnings | Compiler warnings should never be ignored. They highlight potential issues in the code that could lead to security vulnerabilities. By addressing these warnings promptly, you reduce the risk of underlying flaws persisting in the system. |
| 1. Architect and Design for Security Policies | When designing software architecture, it is essential to incorporate security policies from the beginning. This ensures that potential security risks are mitigated early on and that the system follows established guidelines. |
| 1. Keep It Simple | A simple design minimizes the risk of security vulnerabilities. Complexity in code increases the likelihood of errors, making it harder to identify and fix issues. Keeping your code straightforward reduces these risks and improves maintainability. |
| 1. Default Deny | Adopting a "deny by default" approach ensures that only authorized actions are allowed. Standardizing how things are done ensures consistency, and deviations from the norm are quickly identifiable, enhancing security. |
| 1. Adhere to the Principle of Least Privilege | Grant only the necessary access required to perform a task, and ensure this access is available for only as long as needed. This limits the potential impact of security breaches and reduces the risk of exploitation. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data before it is transmitted to external systems is essential to preventing injection attacks. This process ensures that malicious inputs are filtered out, protecting the receiving system from exploitation. |
| 1. Practice Defense in Depth | Always implement multiple layers of security, so if one layer is compromised, additional layers will provide continued protection. This layered approach, often compared to an onion, helps to mitigate risks effectively. |
| 1. Use Effective Quality Assurance Techniques | Ensure that security measures are functioning correctly by incorporating processes for continuous testing, reviewing, and evaluation. This includes both internal reviews and external input to identify potential vulnerabilities. |
| 1. Adopt a Secure Coding Standard | Follow a secure coding standard tailored to the programming language being used. This helps ensure consistent implementation of security practices throughout the development process. |

### 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** | **Data Type Handling** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Ensuring correct data types prevents data truncation and overflow issues, reducing the risk of undefined behavior. |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| short x = 70000; // Noncompliant: Overflow occurs because 'short' can't hold this value |

| **Compliant Code** |
| --- |
| [Compliant description] |
| int x = 70000; // Compliant: Using 'int' to store large values within valid range |

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

| **Principles(s):** Unspecified values can lead to buffer overflow, allowing an attacker to execute unintended code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Medium | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 6.9.0 | CertC++ - INT50 |  |
| Helix QAC | 2023.1 |  |  |
| Parasoft C/C++ test | 4.4 | CERT\_CPP – INT50 -a | An expression with an enum's underlying type must only hold values matching the enumerators of that enumeration. |
| PRQA QA- C++ | 4.4 | 3013 |  |

Coding Standard 2

| **Coding Standard** | **Label** | **Data Value Integrity** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Ensuring the validity of input values reduces the risk of errors or attacks that exploit unexpected or out-of-range values. |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| int age = -10; // Noncompliant: Negative age value, no validation |

| **Compliant Code** |
| --- |
| [Compliant description] |
| int age = getUserInput();  if (age < 0 || age > 120) {  throw std::out\_of\_range("Invalid age"); // Compliant: Input validation ensures correct age range  } |

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

| **Principles(s):** Using incorrect pointers or references can lead to undefined behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Overflow\_upon\_dereference |  |
| PVS-Studio | 7.07 | V783 |  |
| Helix QAC | 2023.1 |  |  |
| Parasoft C/C++ | 2024.1 | CERT\_CPP-CTR5!-a | Avoid modifying a container while iterating over it |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Avoid Buffer Overflows in String Handling** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Buffer overflows can lead to data corruption or security vulnerabilities. Using safe string handling functions prevents these issues. |

| **Noncompliant Code** |
| --- |
| strcpy does not check buffer size, causing potential buffer overflow when the string exceeds the buffer size. |
| char buffer[8];  strcpy(buffer, "Overflow!"); // Noncompliant: No boundary check, causes overflow |

| **Compliant Code** |
| --- |
| snprintf safely copies the string and ensures it doesn’t exceed the buffer size. |
| char buffer[8];  snprintf(buffer, sizeof(buffer), "Safe"); // Compliant: snprintf ensures the buffer size is respected |

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

| **Principles(s):** Dereferencing a null pointer results in undefined behavior. When this occurs, it can lead to unexpected program termination at random times. Attackers may exploit this vulnerability to execute arbitrary code |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 20.10 | Assert\_failure |  |
| Astree | 2023.1 |  |  |
| Parasoft/C++ test | 2024.2 | CERT\_CPP-STR51-a | Avoid null pointer Dereferencing |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevent SQL Injection with Parameterized Queries** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | SQL injections can compromise databases and expose sensitive data. Using parameterized queries ensures that user input is handled securely. |

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

| **Compliant Code** |
| --- |
| Using a prepared statement with placeholders prevents SQL injection, ensuring input is treated as data. |
| std::string query = "SELECT \* FROM users WHERE username = ?";  preparedStatement.setString(1, username); // Compliant: Using parameterized query |

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

| **Principles(s):** When a deallocation function is called on a pointer that was not obtained through an allocation function, it results in undefined behavior, potentially leading to a security breach. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++ | 2024.2 | CERT\_CPP-MEM56-a | Avoid storing an already-owned pointer value in an unrelated smart pointer.  4o |
| HELIX QAC | 2023.1 |  |  |
| Astree | 20.10 | Dangling\_pointer\_use |  |
| PVS – Studio | 7.20 | V1006 |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Properly Deallocate Memory** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Memory leaks can occur if dynamically allocated memory is not freed. Proper memory management prevents resource exhaustion and program crashes. |

| **Noncompliant Code** |
| --- |
| The memory allocated is never deallocated, leading to a memory leak. |
| int\* ptr = new int[10];  // Noncompliant: No delete, memory leak occurs |

| **Compliant Code** |
| --- |
| The allocated memory is properly freed, preventing a memory leak. |
| int\* ptr = new int[10];  delete[] ptr; // Compliant: Properly deallocating memory to avoid memory leak |

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

| **Principles(s):** If a deallocation function receives a pointer that was not allocated by an allocation function, it will result in undefined behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.20 | V515, V554, V611, V701, V748, V773 |  |
| Astree | 20.10 | Invalid\_dynamic\_memory\_allocation\_dangling\_pointer\_use |  |
| Helix QAC | 2023.1 |  |  |
| Parasoft c/C++ | 2024.2 | CERT\_CPP-MEM51-a  CERT\_CPP-MEM51-b  CERT\_CPP-MEM51-c  CERT\_CPP-MEM51-d |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use Assertions for Debugging, Not Input Validation** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions should be used during development to catch programming errors, not for input validation. In production, they can be disabled, leading to unchecked conditions. |

| **Noncompliant Code** |
| --- |
| Assertions are used to check user input, but they may be disabled in production, leaving the condition unchecked. |
| assert(input > 0); // Noncompliant: Using assertions to check user input |

| **Compliant Code** |
| --- |
| Using standard error handling mechanisms ensures the program can handle invalid input appropriately |
| if (input <= 0) {  throw std::invalid\_argument("Input must be greater than zero"); // Compliant: Proper input validation with error handling  } |

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

| **Principles(s):** Static assertions help identify errors and defects in the code, enabling us to resolve them quickly. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | P1 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Codesonar | 6.0p0 |  | Users can develop a custom check utilizing the assert() function. |
| ÉCLAIR | 1.2 |  |  |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Catch Specific Exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Catching specific exceptions provides clearer error handling and ensures that unexpected issues are addressed appropriately. |

| **Noncompliant Code** |
| --- |
| Catching all exceptions with a generic handler makes it hard to understand and resolve specific issues. |
| try {  // Code that might throw exceptions  } catch (...) {  std::cout << "Error occurred"; // Noncompliant: Catching all exceptions without specific handling  } |

| **Compliant Code** |
| --- |
| Handling specific exceptions allows better error diagnosis and appropriate corrective actions. |
| try {  // Code that might throw exceptions  } catch (const std::runtime\_error& e) {  std::cerr << "Runtime error: " << e.what(); // Compliant: Specific exception handling for better diagnostics  } |

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

| **Principles(s):** When an exception is thrown and not caught, it can cause the program to terminate unexpectedly, creating opportunities for attacks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2023.1 |  |  |
| Astree | 20.10 | Potentially-throwing-static-initialization | Partially Checked |
| Parasoft C/C++ | 2024.2 | CERT\_CP-ERR58-a | Exceptions only show when start up |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-008-CPP] | Input validation prevents malicious or malformed data from entering the system, reducing the risk of attacks. |

| **Noncompliant Code** |
| --- |
| Only the length of the input is checked, leaving other potential issues (e.g., SQL injection or XSS) unaddressed. |
| std::string input = getUserInput();  if (input.length() > 100) {  std::cout << "Input too long"; // Noncompliant: Only length validation is performed  } |

| **Compliant Code** |
| --- |
| Input is thoroughly validated using a proper validation function, preventing potential security issues. |
| std::string input = getUserInput();  if (!isValidInput(input)) {  throw std::invalid\_argument("Invalid input"); // Compliant: Comprehensive input validation  } |

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

| **Principles(s):** Alternating between input and output operations on a stream without an intervening flush or repositioning call results in undefined behavior. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Medium | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++ test | 2024.2 | CERT\_CPP\_FIO50-a | Avoid alternating between input and output operations on a stream without an intervening flush or repositioning call. |
| Helix QAC | 2021.1 |  |  |
| Polyspace Bug Finder | R2020a | ECRT C++: FIO50-CPP | Verifies alternating input and output operations on a stream without a flush or positioning call(rule fully covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Use Mutexes for Thread Synchronization** |
| --- | --- | --- |
| Thread Safety | [STD-009-CPP] | Proper synchronization mechanisms, such as mutexes, should be used to avoid race conditions and ensure thread safety. |

| **Noncompliant Code** |
| --- |
| Without synchronization, both threads may access and modify the variable counter simultaneously, leading to a race condition. |
| int counter = 0;  std::thread t1([&] { counter++; });  std::thread t2([&] { counter++; });  t1.join();  t2.join(); // Noncompliant: No synchronization, potential race condition |

| **Compliant Code** |
| --- |
| Using a mutex ensures that access to shared data is synchronized, preventing race conditions. |
| int counter = 0;  std::mutex mtx;  std::thread t1([&] {  std::lock\_guard<std::mutex> lock(mtx);  counter++;  });  std::thread t2([&] {  std::lock\_guard<std::mutex> lock(mtx);  counter++;  });  t1.join();  t2.join(); // Compliant: Mutex used to ensure synchronized access to shared data |

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

| **Principles(s):** Emphasizes ensuring thread safety by using synchronization mechanisms like mutexes to prevent race conditions when multiple threads access shared resources |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace | 2024.1 | Thread Safety | Analyzes thread safety by detecting race conditions and synchronization issues. |
| Coverity | 2023.3 | Race Condition Checker | Identifies potential race conditions and thread safety issues in code. |
| PVS-Studio | 7.20 | Thread Analyazer | Scans for race conditions |
| Parasoft C/C++ Test | 2024.2 |  | Ensures Proper Mutex Usage |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Ensure Proper Resource Release** |
| --- | --- | --- |
| Resource Management | [STD-010-CPP] | Failing to release resources, such as files or sockets, can lead to resource exhaustion. Proper resource management ensures the program runs efficiently. |

| **Noncompliant Code** |
| --- |
| The file is opened but not closed, potentially leading to resource exhaustion or file lock issues. |
| FILE\* file = fopen("data.txt", "r");  // Noncompliant: File not closed |

| **Compliant Code** |
| --- |
| Closing the file ensures resources are released, preventing resource leaks |
| FILE\* file = fopen("data.txt", "r");  fclose(file); // Compliant: File properly closed after use |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Low | P3 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.20 | Resource Leak Detector | Find leaks and undisclosed files |
| Parasoft C/C++ Test |  |  | Ensure files are properly closed |

### 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.

To automate the enforcement of our standards, we need to establish a centralized policy repository that integrates into our assessment and planning processes, serving as the hub for organizational policies, including regulatory compliance, release requirements, and operational rules. This repository will be crucial for initiating policy automation and will be used during verification and testing to assess risks, trigger alerts, and send notifications. Automating compliance improves efficiency by reducing repetitive tasks and streamlining transitions and health checks. Automated penetration testing can further minimize false alarms and simplify issue resolution. Additionally, we can automate the production process by setting up a system to generate logs and store them in a database, enabling us to detect vulnerabilities and prevent attacks through signature validation, data integrity checks, and multi-layered security. Implementing automatic save points will act as clean backups, ensuring the system can be quickly restored to a secure state in case of attacks or technical issues.

### 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 | Probable | High | P6 | L2 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L! |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Likely | Low | P9 | L2 |
| STD-008-CPP | Low | Likely | Medium | P6 | L2 |
| STD-009-CPP | Low | Unlikley | Medium | P2 | L3 |
| STD-010-CPP | Medium | Probable | Low | P3 | L2 |

### 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, such as files and databases, by transforming it into an unreadable format using encryption algorithms like AES-256. This policy ensures that sensitive information—such as personal, financial, or business data—remains secure from unauthorized access, even if storage devices are lost or compromised. Compliance with regulations like GDPR and HIPAA makes this policy essential for safeguarding data privacy. |
| Encryption in flight | Encryption in flight secures data during transmission across networks, preventing unauthorized interception or tampering. All communications over the internet or internal networks must use secure protocols such as TLS/SSL or HTTPS. This policy applies to emails, API calls, and remote access, ensuring the protection of data in transit. It is essential for defending against man-in-the-middle attacks and maintaining compliance with security standards. |
| Encryption in use | Encryption in use protects data while it is actively processed in memory or computational environments, such as during analysis or transactions. Techniques like secure enclaves or homomorphic encryption should be utilized to safeguard data during real-time operations. This policy applies when handling sensitive or personal information, ensuring data confidentiality throughout processing and preventing unauthorized access from insiders or memory-based attacks. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication confirms the identity of users seeking access to systems. The policy mandates multi-factor authentication (MFA) for all logins to strengthen security. Additionally, password policies must enforce complexity and expiration requirements. This applies to all users accessing applications, databases, or networks, ensuring that only authorized individuals can enter, thereby minimizing the risk of unauthorized access. |
| Authorization | Authorization regulates the resources and operations users can access based on their roles and permissions. The policy enforces the principle of least privilege, granting users only the minimum access needed to perform their duties. All access changes, including adding new users or modifying permissions, must be logged and reviewed. This policy helps safeguard sensitive information and critical systems from unauthorized access or privilege escalation. |
| Accounting | Accounting monitors and records user activities, such as login attempts, file access, and database changes. The policy requires logging mechanisms to track all system activities, with logs securely stored and periodically reviewed to identify suspicious behavior or policy breaches. This policy ensures regulatory compliance, aids in incident investigations, and promotes accountability by maintaining a detailed audit trail of user actions. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 09/22/2024 | Initial Milestone | Ali Alshara |  |
| 1.2 | 10/13/2024 | Final Revision | Ali Alshara |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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