**Green Pace Developer: Security Policy Guide**



# 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 | Validate input data means always checking any data the program receives from outside sources such as user input, files, or network connections. If we don’t validate this input properly, it can lead to serious security issues like SQL injection, buffer overflows, or system crashes. Basically, all external data should be treated as untrusted until it's been verified as safe. |
| 1. Heed Compiler Warnings | Compiler warnings are meant to help developers catch potential issues early. It’s important to use the strictest warning settings and fix all the issues the compiler flags. On top of that, using static and dynamic analysis tools can uncover vulnerabilities before the program even runs, making your code more secure and reliable. |
| 1. Architect and Design for Security Policies | Security should be considered from the beginning of the software design process. Build the system with different permission levels in mind, and break it into smaller parts that only have access to what they need. That way, if one part is attacked, the rest of the system is still protected. |
| 1. Keep It Simple | Security should be a core part of the software design process, not something added later. Design systems with layered access and permissions, and break components into smaller parts that only access what they need. That way, if one part is compromised, the rest of the system stays protected. |
| 1. Default Deny | Set the system to block access unless it’s specifically allowed. Users or processes shouldn’t be able to do anything unless given them permission. This way, unauthorized actions are automatically denied. |
| 1. Adhere to the Principle of Least Privilege | Only give users and programs the minimum access they need to do their job. If higher privileges are needed, they should only be used for a short time. This limits the damage if something goes wrong or an account gets hacked. |
| 1. Sanitize Data Sent to Other Systems | Clean up and check data before sending it to things like database or other applications. Even if it was already validates, we need to make sure that outgoing data doesn’t contain anything harmful, this helps prevent attacks like SQL injection and command injection. |
| 1. Practice Defense in Depth | This principle is about using layers of security instead of relying on just one thing. If one layers fails, others can still protect the system. For example, using both a firewall and a strong input validation gives extra protection. |
| 1. Use Effective Quality Assurance Techniques | Use solid testing methods to find and fix security issues. Techniques like fuzz testing, penetration testing, and code reviews are really useful. It also helps to have someone outside the team to review the code. |
| 1. Adopt a Secure Coding Standard | Having a set of secure coding guidelines helps everyone on the team write safer code. It also makes the codebase more consistent and easier to maintain. Standards help us avoid common mistakes and keep security a top priority. |

### 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 Coding Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Using the correct integer types prevents issues such as overflows and ensures compatibility across different platforms. |

| **Noncompliant Code** |
| --- |
| Using a signed integer for array size can lead to unexpected behavior if dataSize is negative. |
| int dataSize = getDataSize();  char buffer[dataSize]; |

| **Compliant Code** |
| --- |
| Using size\_t ensures that the size is non-negative and appropriate for array indexing. |
| size\_t dataSize = getDataSize();  std::vector<char> buffer(dataSize); |

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

| **Principle(s):** Architect and Design for Security Policies   * Choosing the correct data type supports secure architecture and system resilience, avoiding overflow vulnerabilities and platform inconsistencies. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.9 | portability | Detects data type misuse across platforms |
| Clang-Tidy | 14.0 | cppcoreguidelines-pro-type-vararg | Flags improper type usage |
| Coverity | 2023.9 | TAINTED\_SCALAR | Identifies improper or unsafe use of integer types |
| SonarQube | 10.3 | C:S128 | Enforces strong typing and portability |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Data Value Coding Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Validating input values prevents unexpected behavior and potential vulnerabilities from malicious input. |

| **Noncompliant Code** |
| --- |
| No validation on index can lead to out-of-bounds access |
| int index = getUserInput();  array[index] = value; |

| **Compliant Code** |
| --- |
| Validating index ensures it is within the bounds of the array. |
| int index = getUserInput();  if (index >= 0 && index < arraySize) {  array[index] = value;  } |

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

| **Principles(s):** Validate Input Data   * Ensures values like indices are within expected bounds to prevent memory issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3 | cpp:S3512 | Ensures array indices are validated before access to avoid out-of-bounds. |
| PVS-Studio | 7.26 | V557 | Detects expressions that may lead to invalid array access due to unchecked values. |
| Infer | 1.1.0 | ARRAY\_OUT\_OF\_BOUNDS\_L1 | Flags potential array index errors from unvalidated input. |
| CodeSonar | 8.1 | RangeError | Checks if input values are within acceptable range before use. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **String Correctness Coding Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Avoid Dangerous String Functions. Functions like strcpy do not perform bounds checking, leading to potential buffer overflows. |

| **Noncompliant Code** |
| --- |
| strcpy can overflow dest if source is too large |
| char dest[10];  strcpy(dest, source); |

| **Compliant Code** |
| --- |
| strncpy limits the number of characters copied, preventing overflow, and ensures null-termination |
| char dest[10];  strncpy(dest, source, sizeof(dest) – 1);  dest[sizeof(dest) -1] = ‘\0’; |

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

| **Principles(s):** Keep It Simple   * Avoiding dangerous functions like strcpy keeps memory management simpler and safer. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2024.1 | STRING\_NULL | Detects null dereferences and unsafe string operations. |
| PVS-Studio | 7.26 | V579 | Flags unsafe string functions like strcpy without bounds checking. |
| Fortify | 23.1 | Buffer Overflow | Highlights risks of buffer overflow in string copying and manipulation. |
| Clang-Tidy | 15.0 | security.insecureAPI.strcpy | Warns about use of unsafe string copy functions. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **SQL Injection Coding Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Parameterized queries prevent SQL injection by separating code from data. |

| **Noncompliant Code** |
| --- |
| Linking user input directly into SQL queries can lead to SQL injection. |
| std::string query = “SELECT \* FROM users WHERE name = ‘” + userInput + “’”;  executeQuery(query); |

| **Compliant Code** |
| --- |
| Using prepared statements with placeholders separates user input from the SQL command |
| PreparedStatement stmt = connection.prepareStatement("SELECT \* FROM users WHERE name = ?");  stmt.setString(1, userInput);  stmt.executeQuery(); |

**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 injection by treating input and code separately. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarCube | 10.3 | c:S2077 | Detects unsafe SQL string concatenation |
| Fortify SCA | 2023.1 | SQL Injection | Flags improper query building methods |
| CppCheck | 2.9 |  | Limited, but can support with comments and extensions |

#### 

#### Coding Standard 5

| **Coding Standard** | **Label** | **Memory Protection Coding Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Buffer overflows are dangerous because they can allow attackers to overwrite return addresses or function pointers, potentially leading to arbitrary code execution or system compromise. |

| **Noncompliant Code** |
| --- |
| Writing beyond the bounds of buffer causes undefined behavior. |
| char buffer[10];  for (int i = 0; i <= 10; ++i) {  buffer[i] = 'a';  } |

| **Compliant Code** |
| --- |
| Loop condition ensures writing within the bounds of buffer |
| char buffer[10];  for (int i = 0; i < 10; ++i) {  buffer[i] = 'a';  } |

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

| **Principles(s):** Practice Defense in Depth   * Proper indexing is a layer of memory protection to avoid crashes or arbitrary code execution. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.26 | V512 | Detects buffer overflow risks from improper indexing. |
| Coverity | 2024.1 | BUFFER\_OVERRUN | Detects writes beyond buffer bounds. |
| SonarQube | 10.3 | cpp:S2293 | Flags out-of-bound array access. |
| Clang-Tidy | 15.0 | bugprone-sizeof-expression | Warns about incorrect use of sizes causing overflow issues. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Assertions Coding Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions may be disabled in production, so they should not be used for essential runtime checks. |

| **Noncompliant Code** |
| --- |
| If assertions are disabled, pointer may be null, leading to a crash. |
| assert(pointer != nullptr);  \*pointer = value; |

| **Compliant Code** |
| --- |
| Explicitly checking for null ensures safe dereferencing. |
| if (pointer != nullptr) {  \*pointer = value;  } else {  // Handle error  } |

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

| **Principles(s):** Architect and Design for Security Policies   * Critical logic must not rely on optional debugging tools like assert. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | assertWithoutEffect | Detects assertions that do not protect critical code paths. |
| Coverity | 2024.1 | MISRA-C2012-14.3 | Flags use of asserts for critical checks instead of runtime validation. |
| SonarQube | 10.3 | cpp:S2583 | Detects dead code caused by disabled assertions. |
| Visual Studio Static Analyzer | 17.9 | C26010 | Warns when assert is used without fallback error handling. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Exceptions Coding Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Catching exceptions by value can lead to object slicing and unnecessary copies. |

| **Noncompliant Code** |
| --- |
| Catching by value may slice the exception object. |
| try {  // code  } catch (std::exception e) {  // handle  } |

| **Compliant Code** |
| --- |
| Catching by const reference preserves the exception object's integrity. |
| try {  // code  } catch (const std::exception& e) {  // handle  } |

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

| **Principles(s):** Adopt a Secure Coding Standard   * Correctly handling exceptions avoids data loss, resource leaks, and corrupted objects. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 15.0 | bugprone-exception-escape | Detects exception slicing when catching by value. |
| PVS-Studio | 7.26 | V666 | Warns about catching exceptions by value causing object slicing. |
| Coverity | 2024.1 | C++ exception handling | Flags catching exceptions by value as noncompliant. |
| SonarQube | 10.3 | cpp:S2227 | Flags improper exception catching patterns. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Input Validation Coding Standard** |
| --- | --- | --- |
| Input Validation | [STD-008-CPP] | Validating user input prevents unexpected behavior and security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| No validation on userInput can cause exceptions or incorrect values |
| int age = std::stoi(userInput); |

| **Compliant Code** |
| --- |
| Validating and handling exceptions ensures robust input processing |
| try {  int age = std::stoi(userInput);  if (age >= 0 && age <= 120) {  // proceed  } else {  // handle invalid age  }  } catch (const std::exception& e) {  // handle conversion error  } |

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

| **Principles(s):** Validate Input Data   * Ensures input is safe and within valid ranges to prevent exceptions and logic errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify SCA | 23.1 | Input Validation | Detects missing input validation and improper handling. |
| CodeSonar | 8.1 | Input Validation | Flags missing or incomplete user input validation. |
| Cppcheck | 2.10 | untrustedInput | Detects unsanitized input use. |
| Clang-Tidy | 15.0 | cert-msc32-c | Ensures input is validated before conversion and usage. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Resource Management Coding Standard** |
| --- | --- | --- |
| Resource Management | [STD-009-CPP] | Proper management of resources prevents resource leaks and ensures stability. |

| **Noncompliant Code** |
| --- |
| Manual resource management is error-prone |
| FILE\* file = fopen("data.txt", "r");  // use file  fclose(file); |

| **Compliant Code** |
| --- |
| RAII ensures file is closed automatically when it goes out of scope. |
| std::ifstream file("data.txt");  if (file.is\_open()) {  // use file  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques   * Using RAII prevents manual leaks and supports stable, maintainable code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2024.1 | RESOURCE\_LEAK | Detects resource leaks such as missing fclose or close calls. |
| Valgrind | 3.21.0 | Memcheck | Runtime detection of file descriptor and memory leaks. |
| PVS-Studio | 7.26 | V595 | Flags missing resource release calls. |
| Clang Static Analyzer | 15.0 | unix.ResourceLeak | Detects leaking resources like open files or sockets. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Avoid goto Statements** |
| --- | --- | --- |
| Avoid using goto | [STD-010-CPP] | Using goto can make code harder to read and maintain, leading to potential errors. |

| **Noncompliant Code** |
| --- |
| Goto leads to spaghetti code and complicates control flow. |
| if (error) {  goto cleanup;  }  // ...  cleanup:  // cleanup code |

| **Compliant Code** |
| --- |
| Cleaner and easier to use |
| if (error) {  // handle error  } else {  // normal execution |

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

| **Principles(s):** Keep It Simple   * Removing goto makes code more structured, readable, and easier to maintain securely. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Rare | Low | Low | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | goto usage detection | Detects presence of goto statements and warns on use |
| SonarQube | 10.3 | cpp:S1854 | Flags goto statements that reduce code clarity. |
| Clang-Tidy | 15.0 | misc-avoid-goto | Warns on use of goto for cleaner structured code. |
| Visual Studio Static Analyzer | 17.9 | C26800 | Flags goto usage as potential maintainability issue. |

### Defense-in-Depth Illustration

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



## Project One

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

### Revise the C/C++ Standards

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

### Risk Assessment

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

### Automated Detection

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

### Automation

Provide a written explanation using the image provided.



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

Automation is a key part of maintaining secure coding practices, and Green Pace can take advantage of its strong DevOps infrastructure to support this. Using the DevSecOps diagram as a reference, automation can be integrated at every phase of the software development lifecycle. For example, during the Plan and Assess stages, static analysis tools like SonarQube, Cppcheck, and Clang-Tidy can be automatically triggered on code commits to detect violations of the coding standards. During the Build and Test phases, automated unit tests and security scanning tools (like Coverity or Fortify) can ensure that vulnerable code is caught early. In the Release and Production phases, CI/CD pipelines can include security gates to block deployment if critical issues are found. Even during Detect and Monitor, automation can help by using log monitoring and alerts to detect abnormal behavior. By integrating tools into each step shown in the diagram, we reduce manual error, ensure policy enforcement, and promote continuous compliance with security standards.

### 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 | Most Likely | Low | High | 4 |
| STD-002-CPP | High | Most Likely | Low | Critical | 5 |
| STD-003-CPP | High | Most Likely | Medium | Critical | 5 |
| STD-004-CPP | High | Most Likely | Medium | Critical | 5 |
| STD-005-CPP | High | Likely | Medium | High | 4 |
| STD-006-CPP | Medium | Possible | Low | Medium | 3 |
| STD-007-CPP | Medium | Possible | Low | Medium | 3 |
| STD-008-CPP | High | Likely | Medium | High | 4 |
| STD-009-CPP | Medium | Possible | Low | Medium | 3 |
| STD-010-CPP | Low | Rare | Low | Low | 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 on physical devices like servers or hard drives. This ensures that even if someone gains unauthorized access to storage, they cannot read the data without decryption keys. Green Pace will enforce this policy for all databases, backups, and log files stored locally or in the cloud. AES-256 is recommended for encrypting stored data. |
| Encryption in flight | Encryption in flight secures data as it travels across networks. This is critical when transmitting sensitive information between users and servers or between internal services. Green Pace must use TLS (Transport Layer Security) for all HTTP communications (HTTPS) to prevent interception or tampering during transit. |
| Encryption in use | Encryption in use protects data while it is actively being processed in memory. Although this is less common, it’s important in high-security environments. Green Pace will adopt secure memory handling and use libraries that support encryption of sensitive data in memory to protect data from being exposed during runtime. |

| 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. All Green Pace systems must enforce strong authentication methods such as multi-factor authentication for user logins. This ensures that only authorized individuals can access the system, minimizing risks of credential theft. |
| Authorization | Authorization determines what an authenticated user is allowed to do. Green Pace will enforce role-based access control to restrict user permissions based on their roles. For example, only administrators can add users or modify sensitive database settings. This reduces the risk of accidental or malicious changes. |
| Accounting | Accounting tracks user actions, such as logins, file accesses, and database changes. Green Pace must maintain audit logs for critical actions like adding new users, accessing secure files, or changing user access levels. This allows the security team to detect unusual activity and support investigations. |

**\***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 |  |
| 1.0 | 06/13/2025 | Provided all security and definitions. | Maridelle Gonzales | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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