**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 | Ensure that all input data is within the expected range and format to prevent security issues. By validating input, you protect against injection attacks, buffer overflows, and other vulnerabilities. This practice ensures that only valid, expected data is processed by the application, thereby maintaining data integrity and application stability. |
| 1. Heed Compiler Warnings | Compiler warnings often indicate potential issues that could lead to security vulnerabilities. Addressing these warnings is crucial as it helps identify and fix problems early in the development process. By resolving compiler warnings, you make the code more robust and secure, reducing the risk of exploits and enhancing overall code quality. |
| 1. Architect and Design for Security Policies | Integrating security principles during the design phase ensures that the application's architecture supports secure coding practices. This approach makes the application more resilient against attacks and easier to maintain. Designing with security in mind from the beginning helps identify potential threats and vulnerabilities early, allowing for the implementation of effective countermeasures. |
| 1. Keep It Simple | Complex code increases the likelihood of bugs and security flaws. Simplifying code reduces the risk of introducing vulnerabilities and makes it easier to audit, maintain, and secure. By keeping code straightforward and clear, developers can more easily identify and address potential security issues, ensuring a more secure application. |
| 1. Default Deny | Implementing a default deny policy means that access is not granted unless explicitly allowed. This minimizes the attack surface by preventing unauthorized access by default. By adhering to the default deny principle, you ensure that only trusted and verified users or processes can access the system, reducing the risk of unauthorized access and potential security breaches. |
| 1. Adhere to the Principle of Least Privilege | Grant only the minimum level of access necessary for users and processes. This reduces the potential damage from compromised accounts or exploited vulnerabilities. By limiting permissions, you minimize the impact of any security incidents, ensuring that users and processes can only access the information and resources essential for their tasks. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data before sending it to other systems prevents the propagation of harmful inputs and ensures data integrity across systems. This practice protects against injection attacks and other vulnerabilities that could arise from untrusted data. By cleaning and validating data, you maintain the security and reliability of interconnected systems. |
| 1. Practice Defense in Depth | Implementing multiple layers of security controls increases the complexity and cost for attackers, providing a more robust defense against breaches. This strategy ensures that if one security measure fails, additional layers of protection are still in place to thwart attacks. By practicing defense in depth, you create a more resilient security posture that can withstand various types of threats. |
| 1. Use Effective Quality Assurance Techniques | Thorough testing and code reviews help identify and mitigate security vulnerabilities early in the development process. Effective quality assurance techniques ensure higher code quality and security by catching potential issues before they become significant problems. Regular testing, code reviews, and security audits are essential practices for maintaining a secure development lifecycle. |
| 1. Adopt a Secure Coding Standard | Following a secure coding standard ensures consistency in applying security best practices across the development team. This reduces the likelihood of introducing vulnerabilities and promotes a culture of security awareness. By adhering to established coding standards, developers can write more secure code, making the application more robust against attacks and easier to maintain. |

### 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 Safety** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Ensuring correct data type usage prevents type-related vulnerabilities such as type confusion and buffer overflows. |

| **Noncompliant Code** |
| --- |
| Incorrect use of data types can lead to unexpected behavior or vulnerabilities. |
| int array[10];  float \*ptr = (float \*)array; |

| **Compliant Code** |
| --- |
| Proper use of data types ensures memory safety and predictable behavior. |
| int array[10]; int \*ptr = array; |

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

| **Principles(s):** **Validate Input Data:** Ensuring correct data type usage prevents vulnerabilities like buffer overflows or type confusion, which could arise from incorrect data handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 11.0 | Type Safety Checker | Detects improper type conversions and ensures data type safety. |
| SonarQube | 8.9 | Data Validation Rules | Catches type-related issues during the code review process. |
| CodeQL | 2.7 | Type Analysis | Performs semantic code analysis to detect type safety issues. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Data Value Constraints** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Constraining data values ensures that only valid data is processed, preventing logic errors and vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Lack of value constraints can lead to undefined behavior or security issues. |
| int age = getUserInput(); if (age < 0 || age > 150) { // Invalid age } |

| **Compliant Code** |
| --- |
| Proper value constraints prevent invalid data from causing issues. |
| int age = getUserInput(); if (age >= 0 && age <= 120) { // Valid age } |

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

| **Principles(s):** **Architect and Design for Security Policies:** Applying data value constraints during the design phase ensures that security is integrated into the application's architecture, making it more resilient against potential attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.12 | Value Range Checker | |  | | --- | | Analyzes value constraints and  detects potential errors in  data validation. |  |  | | --- | |  | |
| Cppcheck | 2.6 | Range Analysis | Provides range analysis to ensure data values stay within predefined bounds. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **String Correctness** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Ensuring string correctness prevents buffer overflows and format string vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Unsafe string operations can lead to buffer overflows. |
| char buffer[10]; strcpy(buffer, userInput); |

| **Compliant Code** |
| --- |
| Safe string operations prevent buffer overflows. |
| char buffer[10]; strncpy(buffer, userInput, sizeof(buffer) - 1); buffer[sizeof(buffer) - 1] = '\0'; |

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

| **Principles(s):** **Practice Defense in Depth:** String correctness is a key part of a multi-layered security approach, which helps protect against vulnerabilities like buffer overflows and format string attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify Static Code annalyzer | 20.1.0 | String management | Detects unsafe string operations and ensures proper memory handling. |
| Coverity | 2020.09 | Buffer Overflow Prevention | Analyzes code for potential buffer overflow vulnerabilities due to unsafe string operations. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **SQL Injection Prevention** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Preventing SQL injection ensures database security and integrity by preventing malicious SQL code execution. |

| **Noncompliant Code** |
| --- |
| Unsafe SQL queries can be exploited by SQL injection. |
| string query = "SELECT \* FROM users WHERE name = '" + userInput + "'"; |

| **Compliant Code** |
| --- |
| Parameterized queries prevent SQL injection. |
| string query = "SELECT \* FROM users WHERE name = ?"; preparedStatement.setString(1, userInput); |

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

| **Principles(s):** **Sanitize Data Sent to Other Systems:** By ensuring that data is cleaned and validated before use in SQL queries, this principle directly prevents SQL injection attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OWASP ZAP | 2.10.0 | SQL injection Scanner | |  | | --- | | Actively scans for SQL injection  vulnerabilities. |  |  | | --- | |  | |
| SonarQube | 8.9 | SQL injection detection | Statically analyzes the code to detect SQL injection vulnerabilities. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Memory Protection** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Proper memory management prevents buffer overflows, memory leaks, and other memory-related vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Unsafe memory allocation and access can lead to vulnerabilities. |
| char \*buffer = (char \*)malloc(10); strcpy(buffer, userInput); |

| **Compliant Code** |
| --- |
| Safe memory allocation and access prevent vulnerabilities. |
| char \*buffer = (char \*)malloc(10); strncpy(buffer, userInput, 9); buffer[9] = '\0'; |

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

| **Principles(s):** **Adopt a Secure Coding Standard:** Following secure coding practices, including proper memory management, reduces the likelihood of introducing memory-related vulnerabilities |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17.0 | Memory Leak Detection | Detects memory leaks and improper memory handling in C++ applications. |
| AddressSanitizer | GCC 9.3 | Memory Safety | Automatically detects out-of-bounds access and use-after-free issues. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Assertions** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions help catch logical errors and invalid states during development, preventing potential security issues. |

| **Noncompliant Code** |
| --- |
| Lack of assertions can allow invalid states to persist. |
| void process(int value) { // process value } |

| **Compliant Code** |
| --- |
| Assertions catch invalid states early in the development process. |
| void process(int value) { assert(value >= 0); // process value } |

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

| **Principles(s):** **Use Effective Quality Assurance Techniques:** Assertions help catch logical errors and invalid states during development, enhancing the overall security and quality of the application |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.6 | Assertion Validator | Verifies that assertions are correctly used to catch logical errors. |
| Clang static Analyzer | 11.0 | Assertion Enforcement | Checks for the presence and correct use of assertions in the codebase. |
| PVS-Studio | 7.12 | Logical Checks | Detects missing or misplaced assertions, ensuring they are used effectively. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Exception Handling** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Proper exception handling ensures that errors are managed gracefully, preventing unexpected crashes and vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Lack of exception handling can lead to unhandled errors and crashes. |
| void readFile(const char\* filename) { FILE \*file = fopen(filename, "r"); // read file fclose(file); } |

| **Compliant Code** |
| --- |
| Proper exception handling ensures errors are managed gracefully. |
| void readFile(const char\* filename) { FILE \*file = fopen(filename, "r"); if (file == NULL) { // handle error } else { // read file fclose(file); } } |

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

| **Principles(s):** **Practice Defense in Depth:** Proper exception handling is a defensive measure that ensures errors are managed gracefully, preventing unexpected crashes and vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static analyzer | 11.0 | Exception Safety | Checks for proper use of exception handling practices. |
| Coverity | 2020.09 | Exception Management | Detects unhandled exceptions and improper handling patterns. |
| SonarQube | 8.9 | Exception Handling | Ensures that exceptions are handled correctly to prevent crashes. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Boundary Checking** |
| --- | --- | --- |
| Boundary Checking | [STD-008-CPP] | Proper boundary checking prevents buffer overflows and array out-of-bounds errors, ensuring memory safety. |

| **Noncompliant Code** |
| --- |
| Lack of boundary checking can lead to buffer overflows. |
| int array[10]; for (int i = 0; i <= 10; ++i) { array[i] = i; } |

| **Compliant Code** |
| --- |
| Proper boundary checking prevents out-of-bounds errors. |
| int array[10]; for (int i = 0; i < 10; ++i) { array[i] = i; } |

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

| **Principles(s):** **Validate Input Data:** Boundary checking ensures that input data stays within expected ranges, preventing buffer overflows and out-of-bounds errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.6 | Array Bounds Checker | Detects out-of-bounds errors to ensure memory safety |
| Valgrind | 3.17.0 | Boundary Violation Detector | Identifies boundary violations in array usage. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Resource Management** |
| --- | --- | --- |
| Resource Management | [STD-009-CPP] | Effective resource management prevents resource leaks and ensures that resources are properly allocated and deallocated. |

| **Noncompliant Code** |
| --- |
| Poor resource management can lead to memory leaks and resource exhaustion. |
| FILE \*file = fopen("data.txt", "r"); // process file // forgot to close the file |

| **Compliant Code** |
| --- |
| Proper resource management ensures resources are released appropriately. |
| FILE \*file = fopen("data.txt", "r"); if (file) { // process file fclose(file); } |

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

| **Principles(s):** **Adhere to the Principle of Least Privilege:** Proper resource management ensures that resources are allocated and deallocated securely, minimizing the risk of resource-related vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17.0 | Resource Leak Detector | Detects resource leaks and improper resource handling in C++ applications. |
| PVS-Studio | 7.12 | Resource Management Checker | Analyzes code for proper resource allocation and deallocation. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Error Reporting** |
| --- | --- | --- |
| Error Reporting | [STD-010-CPP] | Proper error reporting ensures that errors are communicated effectively without revealing sensitive information that could be exploited by attackers. |

| **Noncompliant Code** |
| --- |
| Directly exposing error messages can provide attackers with valuable information. |
| void connectToDatabase() { if (!db.connect()) { std::cerr << "Database connection failed: " << db.getError() << std::endl; } } |

| **Compliant Code** |
| --- |
| Proper error handling that logs errors without exposing sensitive information. |
| void connectToDatabase() { if (!db.connect()) { logError("Database connection failed."); // Handle error appropriately } } |

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

| **Principles(s):** **Heed Compiler Warnings:** Proper error reporting often begins with addressing compiler warnings, ensuring that potential issues are caught and resolved early in the development process. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Low | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 11.0 | Error Handling Checker | Detects improper error reporting that may reveal sensitive information. |
| PVS-Studio | 7.12 | Error Management | Ensures that errors are reported in a secure manner, avoiding exposure of sensitive details­­­­­­­­­ |

### 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.

[Insert your written explanations here.]

### 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 | Medium | Possible | Low | Medium | 3 |
| STD-003-CPP | High | Likely | Medium | High | 4 |
| STD-004-CPP | High | Likely | High | High | 5 |
| STD-005-CPP | High | Likely | High | High | 5 |
| STD-006-CPP | Low | Possible | Low | Low | 2 |
| STD-007-CPP | Medium | Likely | Medium | Medium | 3 |
| STD-008-CPP | High | Likely | Medium | High | 4 |
| STD-009-CPP | Medium | Possible | Medium | Medium | 3 |
| STD-010-CPP | Medium | Unlikely | Low | Medium | 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 | **Explanation:** Encryption at rest refers to the process of protecting data that is stored on physical or cloud-based storage systems. This ensures that even if the storage media is accessed by unauthorized individuals, the data remains unreadable without the proper decryption keys.  **How and Why the Policy Applies:** The policy applies to all data stored in databases, file systems, or any other storage mediums within the organization. Encrypting data at rest is essential to protect sensitive information from being compromised in the event of physical theft, unauthorized access, or other security breaches. |
| Encryption in flight | **Explanation:** Encryption in flight refers to the process of securing data as it is transmitted across networks, whether within an organization or over the internet. This protects the data from being intercepted or tampered with during transmission.  **How and Why the Policy Applies:** This policy applies to any data being sent between systems, including internal communications and interactions with external entities. Encrypting data in flight is crucial to maintaining the confidentiality and integrity of information as it moves through potentially vulnerable communication channels. |
| Encryption in use | **Explanation:** Encryption in use refers to the protection of data while it is actively being processed or accessed in memory. This ensures that sensitive data remains secure even while it is being used by applications or systems.  **How and Why the Policy Applies:** This policy is important in scenarios where sensitive data may be exposed during processing, especially in shared or cloud environments. Encrypting data in use helps protect against unauthorized access or data leakage during processing. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | **Explanation:** Authentication is the process of verifying the identity of users, systems, or devices attempting to access resources. It is the first line of defense in ensuring that only authorized entities can access sensitive data and systems.  **How and Why the Policy Applies:** The authentication policy applies to all access points within the organization, including user logins, API access, and system communications. Strong authentication mechanisms are critical to prevent unauthorized access and to ensure that only verified users and systems interact with sensitive information. |
| Authorization | **Explanation:** Authorization is the process of granting or denying specific permissions to authenticated users or systems based on their roles, policies, or attributes. It controls access to resources and actions within the system.  **How and Why the Policy Applies:** This policy applies after authentication and ensures that users or systems are only allowed to perform actions or access data that they are explicitly permitted to. Authorization is crucial for enforcing the principle of least privilege, thereby minimizing the risk of unauthorized access and potential data breaches. |
| Accounting | **Explanation:** Accounting, also known as auditing or logging, involves tracking and recording user and system activities. This provides a trail of actions that can be reviewed for security, compliance, and operational purposes.  **How and Why the Policy Applies:** The accounting policy applies to all significant interactions within the organization’s systems, such as logins, data access, and administrative changes. Maintaining comprehensive logs is essential for monitoring user activities, detecting suspicious behavior, and ensuring compliance with security policies and regulations. |

**\***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 |  |
| 2.0 | 08/12/2024 | Completed Security Policy | Jacob Batrano | [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 |