**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 | Validating input data is crucial to prevent injection attacks and ensure that the data is in the expected format. This principle helps mitigate the risk of malicious data entering the system, which could lead to severe vulnerabilities. |
| 1. Heed Compiler Warnings | Ignoring compiler warnings can lead to subtle bugs and security vulnerabilities. By addressing all compiler warnings, developers can catch potential issues early in the development process, improving the overall security and stability of the software. |
| 1. Architect and Design for Security Policies | Designing software with security in mind from the beginning ensures that security measures are integrated into the system architecture. This proactive approach helps prevent security flaws and reduces the need for costly fixes later. |
| 1. Keep It Simple | Complex systems are harder to secure and more prone to errors. By keeping designs simple, developers can more easily identify and mitigate potential security issues, resulting in more robust and secure software. |
| 1. Default Deny | The default deny principle ensures that access is restricted by default, only allowing access when explicitly granted. This minimizes the attack surface and helps prevent unauthorized access to system resources. |
| 1. Adhere to the Principle of Least Privilege | Granting the least amount of privilege necessary for a task limits the potential damage that can be done by a compromised account or process. This principle helps contain security breaches and reduces the risk of privilege escalation attacks. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data before sending it to other systems prevents the propagation of malicious data and ensures that the receiving system processes safe and valid input. This is critical for maintaining the integrity of data exchanges. |
| 1. Practice Defense in Depth | Implementing multiple layers of security controls provides redundancy in case one layer fails. This comprehensive approach ensures that even if one security measure is bypassed, additional measures are in place to protect the system. |
| 1. Use Effective Quality Assurance Techniques | Regularly testing and reviewing code through quality assurance techniques such as static analysis, code reviews, and automated testing helps identify and fix security vulnerabilities early in the development process. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard provides guidelines for writing secure code, helping developers avoid common security pitfalls and ensuring that security best practices are consistently applied. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Ensuring the correct use of data types prevents errors such as buffer overflows and type mismatches. |

| **Noncompliant Code** |
| --- |
| This code can cause a buffer overflow as the string exceeds the buffer size. |
| char buffer[10];  strcpy(buffer, "This string is too long for the buffer"); |

| **Compliant Code** |
| --- |
| This code safely copies the string to the buffer, preventing overflow. |
| char buffer[10]; strncpy(buffer, "Safe", 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):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 13.0.0 | Core foundation function checker | Analyzes code for correct data type usage, identifying potential type mismatches and buffer overflows. |
| Cppcheck | 2.8 | Mismatches | Detects incorrect usage of data types, such as type mismatches that could lead to runtime errors. |
| Clang-Tidy | 13.0.0 | Modernize-Use-Using | Ensures modern C++ practices, including correct data type usage, helping to prevent type-related vulnerabilities. |
| Bug Finder | R2023a | Data type misuse | Analyzes code for potential data type misuse and helps to ensure type safety. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Validating data values prevents invalid or malicious data from being processed. |

| **Noncompliant Code** |
| --- |
| This code assigns an invalid age value. |
| int age = -5; |

| **Compliant Code** |
| --- |
| This code checks the age value to ensure it is within a valid range. |
| int age = 5;  if (age < 0 || age > 120) {  throw std::invalid\_argument("Invalid age value");  } |

**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 | Likely | Low | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.2 | Invalid value checker | Detects invalid or out-of-range values that could lead to unexpected behavior or vulnerabilities. |
| Polyspace Code Prover | R2023a | Range checking | Performs static analysis to ensure that data values fall within expected ranges. |
| CodeSonar | 6.3p0 | Value analysis | Analyzes value assignments and checks for potential value-related errors. |
| Frama-C | 22.0 (Vanadium) | Value analysis | Checks data values for validity and range compliance during static analysis. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Ensuring string correctness prevents buffer overflows and data corruption. |

| **Noncompliant Code** |
| --- |
| This code can cause a buffer overflow. |
| char name[10];  strcpy(name, "A very long name"); |

| **Compliant Code** |
| --- |
| This code safely copies the string to the buffer. |
| char name[10];  strncpy(name, "Short", sizeof(name) - 1);  name[sizeof(name) - 1] = '\0'; |

**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** |
| --- | --- | --- | --- | --- |
| High | Possible | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2024.1 | String operation checker | Ensures safe string operations, preventing buffer overflows and data corruption. |
| Splint | 3.1.2 | String handling | Analyzes C code to ensure safe string operations, detecting potential buffer overflows. |
| AppScan Source | 10.0.3 | String manipulation safety | Scans for unsafe string operations that could lead to buffer overflows or data corruption. |
| Klocwork | 2024.1 | Buffer overflow detection | Identifies potential buffer overflow issues related to string handling. |

#### 

#### Coding Standard 4

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

| **Noncompliant Code** |
| --- |
| This code is vulnerable to SQL injection. |
| std::string query = "SELECT \* FROM users WHERE username = '" + username + "'"; |

| **Compliant Code** |
| --- |
| This code uses prepared statements to prevent SQL injection. |
| std::string query = "SELECT \* FROM users WHERE username = ?";  stmt->setString(1, username); |

**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** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OWASP ZAP | 2.10.0 | SQL Injection scanner | Identifies SQL injection vulnerabilities by analyzing query structures and parameters. |
| SQLMap | 1.6.5 | SQL Injection scanner | Automated tool specifically designed to detect SQL injection vulnerabilities. |
| AppScan Enterprise | 10.0.3 | SQL Injection detection | Comprehensive scanning tool for identifying SQL injection risks within web applications. |
| Veracode Static Analysis | 2023.4 | SQL Injection | Analyzes code to detect SQL injection vulnerabilities and provides remediation advice. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Protecting memory prevents vulnerabilities such as buffer overflows and memory leaks. |

| **Noncompliant Code** |
| --- |
| This code causes a memory leak. |
| int\* ptr = new int[10];  // Forgot to delete ptr |

| **Compliant Code** |
| --- |
| This code properly deallocates memory. |
| int\* ptr = new int[10];  delete[] ptr; |

**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** |
| --- | --- | --- | --- | --- |
| High | Possible | High | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17.0 | Memcheck | Detects memory leaks, improper memory access, and buffer overflows, ensuring robust memory management. |
| AddressSanitizer (AScan) | 13.0.0 | Memory corruption | A runtime memory error detector for C/C++ programs, identifies out-of-bounds and use-after-free bugs. |
| Helgrind (part of Valgrind) | 3.17.0 | Memory synchronization | Detects memory synchronization errors in multi-threaded programs. |
| Infer | 1.1.0 | Memory leak detection | Static analysis tool that identifies memory leaks, null pointer dereferences, and resource leaks. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Using assertions helps catch programming errors during development. |

| **Noncompliant Code** |
| --- |
| This code uses an invalid assertion. |
| int index = -1;  assert(index >= 0); |

| **Compliant Code** |
| --- |
| This code uses a valid assertion. |
| int index = 5;  assert(index >= 0); |

**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 | Unlikely | Low | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.14 | V100 | Ensures proper use of assertions, preventing logic errors during runtime. |
| GCC -Wall | 11.2 | Assertion validity | Enables warnings that help detect invalid or unreachable assertions in code. |
| Eclipse CDT | 2024 | Assertion checking | Provides integrated tools to ensure assertions are used correctly in the code. |
| Clang-Tidy | 13.0.0 | AssertLike-Check | Ensures proper use and validation of assert-like functions in C++ code. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Properly handling exceptions ensures the program can recover from errors. |

| **Noncompliant Code** |
| --- |
| This code does not handle division by zero. |
| float divide(float num, float den) {  return num / den;  } |

| **Compliant Code** |
| --- |
| This code checks for division by zero and throws an exception if necessary. |
| float divide(float num, float den) {  if (den == 0) {  throw std::runtime\_error("Division by zero error!");  }  return num / den;  } |

**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** |
| --- | --- | --- | --- | --- |
| High | Possible | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Visual Studio Code Analysis | 2024 | Exception handling checker | Analyzes code for proper exception handling, ensuring that exceptions are correctly managed. |
| Coverity | 2024.1 | Exception safety | Detects improper exception handling and ensures code is exception-safe. |
| Eclipse PTP | 2023 | Exception propagation analysis | Analyzes exception propagation paths to ensure proper handling of exceptions. |
| Cppcheck | 2.8 | Exception handling | Static analysis tool that checks for proper use of exceptions in C++ code. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Validation | STD-008-CPP | Validating input ensures that only expected data is processed. |

| **Noncompliant Code** |
| --- |
| This code does not validate user input. |
| std::cin >> userInput; |

| **Compliant Code** |
| --- |
| This code validates user input before processing. |
| std::cin >> userInput; if (!isValid(userInput)) { throw std::invalid\_argument("Invalid input"); } |

**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** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify Static Code Analyzer | 21.2.1 | Input validation checker | Scans code for input validation issues, preventing malicious data from being processed. |
| Fortify WebInspect | 21.2.1 | Input validation scanner | Dynamic analysis tool that tests for input validation issues in web applications. |
| Parasoft C/C++test | 2024.1 | Input validation | Automates the detection of input validation issues within C/C++ applications. |
| SonarQube | 9.2 | Security Hotspots | Identifies input validation issues that could lead to vulnerabilities like XSS or injection. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Resource Management | STD-009-CPP | Properly managing resources prevents leaks and ensures stability. |

| **Noncompliant Code** |
| --- |
| This code causes a resource leak. |
| FILE\* file = fopen("file.txt", "r");  // Forgot to close the file |

| **Compliant Code** |
| --- |
| This code ensures the file is closed after use. |
| FILE\* file = fopen("file.txt", "r");  if (file) {  fclose(file);  } |

**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 | Possible | Medium | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 13.0.0 | Resource management checker | Ensures proper allocation and deallocation of resources, preventing leaks and ensuring stability. |
| IBM Rational Purify | 2023 | Resource leak detection | A dynamic analysis tool that identifies memory leaks and improper resource management. |
| Dr. Memory | 2.3.0 | Resource usage | Analyzes memory usage patterns to detect leaks and improper resource management. |
| Polyspace Code Prover | R2023a | Resource management | Statistically verifies resource allocation and deallocation in embedded systems. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Error Logging | STD-010-CPP | Logging errors helps with debugging and monitoring. |

| **Noncompliant Code** |
| --- |
| This code does not log errors. |
| if (error) {  // No logging  } |

| **Compliant Code** |
| --- |
| This code logs errors for further analysis. |
| if (error) {  logError("An error occurred");  } |

**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 | Unlikely | Low | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LogCheck | 1.3.23 | Error logging scanner | Analyzes code for proper error logging practices, ensuring that all errors are logged for further analysis. |
| Log4j | 2.17.1 | Error logging | Analyzes logging practices to ensure that error messages are properly recorded. |
| Splunk | 8.2 | Log analysis | Aggregates and analyzes logs to ensure that error logging is consistent and effective. |
| Syslog-ng | 3.35.1 | Log management | Ensures that system logs, including error logs, are properly managed and stored for future analysis. |

### 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 | Likely | Low | Medium | 3 |
| STD-003-CPP | High | Possible | Medium | High | 4 |
| STD-004-CPP | High | Likely | Medium | High | 5 |
| STD-005-CPP | High | Possible | High | High | 4 |
| STD-006-CPP | Medium | Unlikely | Low | Medium | 2 |
| STD-007-CPP | High | Possible | Medium | High | 4 |
| STD-008-CPP | High | Likely | Medium | High | 5 |
| 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 | Encryption at rest refers to protecting data while it is stored on disk, database, or other storage media. This policy applies to ensure that sensitive data is secured from unauthorized access while it is stored, particularly in environments where the storage device could be lost or stolen. Encryption at rest is crucial in scenarios where data is stored long-term, such as in databases, file systems, and backup storage. The policy ensures that only authorized users with the correct encryption keys can access the stored data, thereby protecting it from breaches. |
| Encryption in flight | Encryption in flight, also known as data in transit encryption, protects data as it moves across networks, between systems, or across the internet. This policy applies to all data that is transferred between users and systems to prevent interception by unauthorized parties. Encryption in flight is essential when transmitting sensitive information over networks, particularly over the internet, where data can be intercepted by malicious actors. The policy ensures that data is encrypted during transmission, protecting its confidentiality and integrity until it reaches its intended destination. |
| Encryption in use | Encryption in use refers to protecting data while it is being processed or actively used in memory. This policy applies to scenarios where sensitive data is decrypted for processing within applications or systems. Encryption in use is vital to ensure that even when data is being utilized, it remains protected from unauthorized access, particularly in environments that handle highly sensitive information, such as financial or medical systems. The policy ensures that data is secured throughout its lifecycle, including while it is being processed, to prevent unauthorized access or breaches. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying the identity of a user, system, or application. This policy applies to ensure that only authorized individuals or systems can gain access to resources or data. Authentication methods can include passwords, biometrics, or multi-factor authentication (MFA). The policy is critical for preventing unauthorized access and ensuring that users are who they claim to be. This is especially important for user logins, access to sensitive systems, and when making changes to databases. |
| Authorization | Authorization refers to the process of determining and enforcing what an authenticated user or system is allowed to do. This policy applies to control access levels and permissions, ensuring that users only have access to the data and resources necessary for their roles. The policy ensures that even if a user is authenticated, they cannot access data or systems beyond their permissions, which is essential for maintaining security and privacy. This applies to managing user levels of access, restricting actions like adding new users, or accessing critical files. |
| Accounting | Accounting, also known as auditing, involves tracking user activities, such as logins, changes made to the database, and access to files. This policy applies to maintain logs that can be reviewed for security, compliance, and forensic purposes. The policy is crucial for detecting and responding to security incidents, as it provides a detailed record of all actions performed by users. It ensures that all user activities are logged and monitored, allowing for accountability and traceability in the event of a breach or misuse of resources. |

**\***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

Data Type (STD-001-CPP): The principles that best support the data type standard are Principle 2: Heed Compiler Warnings and Principle 9: Use Effective Quality Assurance Techniques. Heeding compiler warnings is crucial because improper use of data types often triggers compiler warnings. By addressing these warnings, developers can identify and correct issues early in the development process, thereby supporting the correct use of data types. Additionally, effective quality assurance techniques, such as regular testing and code reviews, help ensure that data types are used correctly, preventing vulnerabilities and ensuring the stability and security of the software.

Data Value (STD-002-CPP): This standard is closely linked to Principle 1: Validate Input Data and Principle 7: Sanitize Data Sent to Other Systems. Validating input data ensures that data values are within expected ranges, preventing the processing of invalid or malicious data, which directly supports the data value standard. Furthermore, sanitizing data before it is sent to other systems is critical for ensuring that only valid data is exchanged between systems, which helps protect against potential security breaches and data corruption.

String Correctness (STD-003-CPP): The string correctness standard is supported by Principle 4: Keep It Simple and Principle 7: Sanitize Data Sent to Other Systems. Simplifying string operations reduces the likelihood of errors such as buffer overflows, making it easier to ensure that strings are handled correctly. Additionally, sanitizing data that includes strings before sending it to other systems prevents vulnerabilities like buffer overflows or injection attacks, thereby ensuring data integrity and security.

SQL Injection (STD-004-CPP): The SQL injection standard is supported by Principle 1: Validate Input Data, Principle 7: Sanitize Data Sent to Other Systems, and Principle 8: Practice Defense in Depth. Validating input data is essential for preventing SQL injection, as unvalidated inputs can lead to malicious SQL queries being executed. Sanitizing data helps ensure that only safe inputs are used in database queries, further reducing the risk of SQL injection. Additionally, practicing defense in depth by implementing multiple layers of security controls, such as input validation and prepared statements, ensures that the system remains protected even if one control is bypassed.

Memory Protection (STD-005-CPP): The memory protection standard aligns with Principle 6: Adhere to the Principle of Least Privilege and Principle 9: Use Effective Quality Assurance Techniques. Adhering to the principle of least privilege ensures that memory access is limited to what is strictly necessary, thereby reducing the risk of exploitation. Effective quality assurance techniques, such as regular testing and reviews, help identify and mitigate memory-related vulnerabilities, ensuring robust memory management and protecting against potential security threats.

Assertions (STD-006-CPP): The use of assertions is supported by Principle 3: Architect and Design for Security Policies and Principle 9: Use Effective Quality Assurance Techniques. Assertions play a critical role in enforcing security policies by catching programming errors during development, thereby preventing vulnerabilities from being introduced into the code. Regular use of assertions, coupled with effective quality assurance techniques, ensures that assumptions made in the code are validated at runtime, reducing the risk of errors and improving overall software security.

Exceptions (STD-007-CPP): The exceptions standard is supported by Principle 9: Use Effective Quality Assurance Techniques and Principle 5: Default Deny. Proper handling of exceptions is essential for ensuring that the code can recover gracefully from errors, which is a key aspect of quality assurance. Additionally, following the default deny principle ensures that exceptions are handled in a way that does not expose sensitive information or leave the system in an insecure state, thereby enhancing the overall security posture of the software.

Input Validation (STD-008-CPP): This standard is closely linked to Principle 1: Validate Input Data, Principle 7: Sanitize Data Sent to Other Systems, and Principle 8: Practice Defense in Depth. Validating input data is the first line of defense against many types of attacks, making it a critical component of the input validation standard. Sanitizing data before it is processed or sent to other systems ensures that it is safe and free from malicious content, protecting against vulnerabilities such as cross-site scripting or injection. Practicing defense in depth by implementing multiple validation checks throughout the system ensures that even if one validation step is bypassed, others will still protect the system.

Resource Management (STD-009-CPP): The resource management standard is supported by Principle 6: Adhere to the Principle of Least Privilege and Principle 9: Use Effective Quality Assurance Techniques. Adhering to the principle of least privilege ensures that only necessary resources are allocated and accessed, reducing the attack surface and minimizing the risk of exploitation. Effective quality assurance techniques, including regular analysis and testing of resource management practices, help prevent resource leaks and ensure system stability, supporting secure and efficient resource usage.

Error Logging (STD-010-CPP): The error logging standard is supported by Principle 9: Use Effective Quality Assurance Techniques and Principle 10: Adopt a Secure Coding Standard. Proper error logging is an essential part of quality assurance, as it ensures that errors are captured and analyzed to improve the security and stability of the software. Adopting a secure coding standard helps ensure that errors are logged in a way that does not expose sensitive information, thereby adhering to best practices and enhancing the overall security of the system.

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 | 07/29/2024 | Initial Template | Urika Pye | Urika Pye |
| 2.0 | 08/18/2024 | Completed Template | Urika Pye | Urika Pye |
| [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 |