**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 | All input must be validated to ensure it meets expected formats and constraints before it is processed. Input validation prevents malicious data from being used to exploit software vulnerabilities such as buffer overflows, SQL injection, and command injection. Validating data early and consistently reduces attack surfaces significantly. |
| 1. Heed Compiler Warnings | Compilers often emit warnings that can indicate potentially dangerous or non-portable code. Developers must treat these warnings seriously, using them as a means to improve code safety and reliability. Enabling all warnings and treating them as errors is a best practice that encourages secure coding. |
| 1. Architect and Design for Security Policies | Security must be considered from the initial stages of system design. This involves choosing appropriate security controls, defining trust boundaries, and designing with least privilege and secure defaults in mind. Good architecture anticipates threats and integrates defense mechanisms proactively. |
| 1. Keep It Simple | Complex code is more prone to bugs and vulnerabilities. Following the KISS (Keep It Simple, Stupid) principle helps ensure that systems are maintainable, testable, and secure. Simpler code is easier to audit and less likely to include hidden flaws. |
| 1. Default Deny | Systems and applications should be configured with a “deny by default” posture, granting access or privileges only when explicitly allowed. This reduces the risk of unintended access or privilege escalation and enforces a principle of least surprise. |
| 1. Adhere to the Principle of Least Privilege | Each process, user, or component should operate using the minimum set of privileges necessary to perform its function. This minimizes the potential impact of a compromise, containing security incidents within the smallest possible scope. |
| 1. Sanitize Data Sent to Other Systems | Before transmitting data to other systems (such as databases, browsers, or third-party APIs), it must be sanitized to remove any potentially malicious or unintended commands. Sanitization ensures that the data cannot be used to manipulate the target system in unintended ways. |
| 1. Practice Defense in Depth | No single security control is sufficient on its own. Defense in depth involves layering multiple security measures (e.g., input validation, firewalls, access control) to protect systems even if one layer fails. Redundancy in security provides resilience against evolving threats. |
| 1. Use Effective Quality Assurance Techniques | Robust QA practices such as static analysis, dynamic testing, fuzzing, and code reviews are essential for detecting security flaws early. Security is not just a development concern but must be continuously validated through testing and analysis. |
| 1. Adopt a Secure Coding Standard | Following a well-defined secure coding standard, such as the SEI CERT C/C++ coding rules, helps enforce consistent practices that reduce the chance of introducing security vulnerabilities. Standards provide a framework for writing reliable, maintainable, and safe code. |

### 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** | Enforce Explicit Data Type Usage |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Using explicit, well-defined data types prevents unintended behavior caused by implicit conversions, platform-specific size assumptions, and undefined behavior. In C/C++, failing to use the correct data types can lead to memory corruption, integer overflows, and security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This code uses implicit data type conversion, which can lead to truncation or overflow. |
| int getUserAge() {  short age;  cin >> age; // Potential data truncation if input exceeds 'short' limit  return age;  } |

| **Compliant Code** |
| --- |
| This code uses explicit data types and range checking to prevent truncation or overflow issues |
| int getUserAge() {  int age; cin >> age;  if (age < 0 || age > 120) {  throw std::out\_of\_range("Invalid age input.");  }  return age;  } |

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

| **Principles(s):**   * **Validate Input Data:** Ensures that user-supplied data is appropriate and secure. * **Adopt a Secure Coding Standard:** Follows best practices from the CERT C++ guidelines. * **Keep It Simple:** Reduces potential issues with type handling and improves clarity. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 16.0 | clang-analyzer-core.CallAndMessage | Detects type conversion issues and bad usage patterns including implicit casting, especially with dangerous types. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Validate All External Input Values** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Validating values prevents malicious inputs from altering system behavior. |

| **Noncompliant Code** |
| --- |
| Doesn’t validate user-supplied index. |
| int arr[10];  cin >> index;  cout << arr[index]; // Potential out-of-bounds access |

| **Compliant Code** |
| --- |
| Validates array access. |
| int arr[10];  cin >> index;  if (index >= 0 && index < 10)  cout << arr[index];  else  throw std::out\_of\_range("Invalid index."); |

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

| **Principles(s):**   * **Validate Input Data**: Directly aligns with the need to check user inputs to prevent security issues. * **Default Deny**: Denies potentially harmful inputs unless explicitly verified. * **Use Effective Quality Assurance Techniques**: Value validation is a core test case for QA and static analysis tools. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity Static Analysis | 2023.12 | BAD\_INDEX | Identifies improper array access, including out-of-bounds indexing caused by unvalidated input |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Avoid Unsafe String Handling** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Improper string handling can lead to buffer overflows and memory corruption. |

| **Noncompliant Code** |
| --- |
| Uses unsafe strcpy without bounds checking. |
| char dest[10];  strcpy(dest, input); // No bounds check |

| **Compliant Code** |
| --- |
| Uses strncpy with limits. |
| char dest[10];  strncpy(dest, input, sizeof(dest) - 1);  dest[9] = '\0'; |

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

| **Principles(s):**   * **Sanitize Data Sent to Other Systems**: Prevents malformed or dangerous string data from propagating. * **Practice Defense in Depth**: Combines safer string functions with buffer sizing and null-termination. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify Static Code Analyzer | 23.1 | Buffer Overflow: Overrun-Static | Flags use of unsafe functions like strcpy, recommending safer alternatives such as strncpy. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevent SQL Injection** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Unsensitized input in SQL queries can lead to severe data breaches. |

| **Noncompliant Code** |
| --- |
| Directly uses user input in SQL query. |
| string query = "SELECT \* FROM users WHERE name = '" + userInput + "'"; |

| **Compliant Code** |
| --- |
| Uses prepared statements. |
| PreparedStatement stmt = conn->prepareStatement("SELECT \* FROM users WHERE name = ?");  stmt->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**: Core to preventing malicious SQL input. * **Architect and Design for Security Policies**: Using prepared statements reflects good system design for secure DB access. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 LTS | cpp:S3649 | Warns against building SQL queries with user input directly, suggests prepared statements or parameterized queries. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Prevent Use-After-Free and Double-Free** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Improper memory management can lead to crashes or arbitrary code execution. |

| **Noncompliant Code** |
| --- |
| Double frees memory. |
| char\* p = new char[10];  delete[] p;  delete[] p; // Error |

| **Compliant Code** |
| --- |
| Sets pointer to null after freeing. |
| char\* p = new char[10];  delete[] p;  p = nullptr; |

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

| **Principles(s):**   * **Practice Defense in Depth**: Proper memory management is part of layered defenses. * **Use Effective Quality Assurance Techniques**: Tools can detect memory misuse early in the pipeline. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind / Memcheck | 3.21.0 | Invalid read/write | Runtime memory error detector that flags improper memory use such as use-after-free and double-delete. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Do Not Rely on Assertions for Input Validation** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions can be disabled in production and should not replace runtime checks. |

| **Noncompliant Code** |
| --- |
| Uses assert() to check input. |
| assert(input > 0); // May not run in production |

| **Compliant Code** |
| --- |
| Uses explicit runtime checks. |
| if (input <= 0) {  throw std::invalid\_argument("Input must be positive.");  } |

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

| **Principles(s):**   * **Validate Input Data**: Assertions aren’t sufficient for real validation—runtime checks are required. * **Use Effective Quality Assurance Techniques**: Encourages testable, production-ready input validation logic. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | P1 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.12 | assertWithSideEffect | Detects incorrect use of assertions for input validation, and flags code that might behave incorrectly in production. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle Exceptions Safely and Specifically** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Catching all exceptions generically can hide bugs and cause undefined behavior. |

| **Noncompliant Code** |
| --- |
| Catches all exceptions silently. |
| try {  riskyFunction();  } catch (...) {  // Do nothing  } |

| **Compliant Code** |
| --- |
| Catches specific exception types. |
| try {  riskyFunction();  } catch (const std::exception& ex) {  logError(ex.what());  } |

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

| **Principles(s):**   * **Keep It Simple**: Specific exception handling leads to clearer, more reliable code. * **Use Effective Quality Assurance Techniques**: Enables better error tracking and code coverage in testing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 16.0 | misc-throw-by-value-catch-by-reference | Identifies poor exception handling patterns, including catching all exceptions or catching by value. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Prevent Buffer Overflow in Manual Memory Operations** |
| --- | --- | --- |
| [Student Choice] | [STD-008-CPP] | Buffer overflows remain one of the most critical vulnerabilities in C/C++ programs. |

| **Noncompliant Code** |
| --- |
| Unsafe manual memory copy. |
| char buffer[8];  memcpy(buffer, input, strlen(input)); // Unsafe |

| **Compliant Code** |
| --- |
| Uses size-limited copy. |
| char buffer[8];  strncpy(buffer, input, sizeof(buffer) - 1);  buffer[7] = '\0'; |

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

| **Principles(s):**   * **Validate Input Data**: Ensures inputs don’t exceed buffer boundaries. * **Practice Defense in Depth**: Adds layers like boundary checks and safe functions to memory operations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| AddressSanitizer (ASan) | LLVM 16.0 | heap-buffer-overflow | Detects out-of-bounds memory accesses during runtime, especially during manual memory operations. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Secure File Access and Validation** |
| --- | --- | --- |
| [Student Choice] | [STD-009-CPP] | Files must be accessed securely to avoid race conditions, leaks, or improper permissions. |

| **Noncompliant Code** |
| --- |
| Opens user-defined file path without validation. |
| ifstream file(userPath); |

| **Compliant Code** |
| --- |
| Validates path and access mode before use. |
| if (validatePath(userPath)) {  ifstream file(userPath, ios::in);  } |

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

| **Principles(s):**   * **Adhere to the Principle of Least Privilege**: Only opens files that the program needs, and only after validation. * **Architect and Design for Security Policies**: Enforces proper access controls and secure file handling design. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Flawfinder | 2.0.19 | open() misuse and unchecked file paths | Flags dangerous functions and I/O operations, particularly around file access that isn't validated. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Use Secure Random Number Generators** |
| --- | --- | --- |
| [Student Choice] | [STD-010-CPP] | Weak randomness undermines cryptographic operations and session management. |

| **Noncompliant Code** |
| --- |
| Uses rand() for security tokens. |
| int token = rand(); // Not secure |

| **Compliant Code** |
| --- |
| Uses std::random\_device or crypto library. |
| std::random\_device rd;  std::mt19937 gen(rd());  std::uniform\_int\_distribution<> dist(1000, 9999);  int token = dist(gen); |

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

| **Principles(s):**   * **Adopt a Secure Coding Standard**: Uses cryptographically secure random generation, as recommended by CERT and OWASP. * **Architect and Design for Security Policies**: Applies strong randomness in security-critical functions like tokens or keys. * **Use Effective Quality Assurance Techniques**: Can be tested and audited for cryptographic strength. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 LTS | cpp:S2245 | Warns when insecure random number generators like rand() are used. |

### 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 to enforce and ensure compliance with the security standards outlined in this policy. At Green Pace, the existing DevOps process provides a strong foundation for secure software delivery. However, in order to meet the evolving requirements of secure software development, automation must be integrated with a DevSecOps approach—embedding security at every stage of the development lifecycle.

The DevSecOps diagram illustrates this through an infinity-shaped (figure-eight) loop, representing the continuous nature of the software development and deployment cycle. The process begins with assessing and planning, where security requirements are identified and risk assessments are performed. This is followed by designing and building, where secure coding practices and static analysis tools are applied during development.

At the center of the loop lies DevSecOps, emphasizing that security is not an isolated phase, but a constant presence throughout the entire process. As the cycle continues through testing, release, deployment, operation, and monitoring, automated tools enforce policy standards—such as input validation, memory safety, and secure data handling—without relying solely on manual oversight.

To modify the existing DevOps process at Green Pace:

* Security checks (like static code analysis and dependency scanning) should be integrated into CI/CD pipelines.
* Policy compliance should be verified automatically before code is merged or deployed.
* Automated testing should include security unit tests, dynamic scans, and runtime monitoring.
* Auditing and logging mechanisms should be continuously reviewed to ensure traceability and accountability.

By embedding these security controls into each phase, automation ensures that standards are not only consistently applied but also scalable and adaptive. This continuous, automated approach improves compliance, reduces risk, and strengthens the security posture of all Green Pace software systems.

### 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 | P9 | 2 |
| **STD-002-CPP** | Low | Unlikely | Low | P3 | 3 |
| **STD-003-CPP** | High | Probable | Medium | P18 | 1 |
| **STD-004-CPP** | High | Likely | Medium | P18 | 1 |
| **STD-005-CPP** | High | Likely | Medium | P18 | 1 |
| **STD-006-CPP** | Low | Unlikely | High | P1 | 1 |
| **STD-007-CPP** | Low | Probable | Medium | P4 | 3 |
| **STD-008-CPP** | Low | Likely | Medium | P6 | 2 |
| **STD-009-CPP** | Low | Unlikely | Medium | P2 | 3 |
| **STD-010-CPP** | Medium | Probable | High | P4 | 3 |

### 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 | This protects data while it is stored—whether on a database, mobile device, computer, or in the cloud. Tools like AxCrypt and InnoD8 Tablespace Encryption are effective options. Implementing encryption at rest is critical for safeguarding data from logical theft, physical theft, or unauthorized access. |
| Encryption in flight | Also known as encryption in flight, this involves securing data as it moves across networks. It is typically implemented using secure communication protocols such as TLS (Transport Layer Security) or SFTP (Secure File Transfer Protocol). This practice should be used whenever company data is transmitted to prevent unauthorized access or data leaks. |
| Encryption in use | This refers to keeping data encrypted even while it is actively being processed in memory. It helps protect against threats such as memory dumps from forced system crashes. BitLocker is a strong example of this, as it encrypts data on the system so that it cannot be accessed without the proper decryption key. Encryption should be applied to sensitive company data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies a user's identity using credentials. Common methods include usernames and passwords, single sign-on (SSO), biometrics, and digital certificates. Ensuring proper authentications for all users is essential to prevent theft and unauthorized access to systems. |
| Authorization | Authorization determines what level of access a user has to system resources—such as files, folders, emails, or applications. This helps prevent situations where, for example, an attacker gains access to a lower-level employee's credentials but is still blocked from accessing critical systems or sensitive information. |
| Accounting | Accounting involves maintaining logs and records of user activity. This includes actions such as logging in, accessing or modifying files, creating accounts, and updating databases. These records are essential for detecting internal threats, auditing usage, and investigating incidents by tracing what actions a user has taken within the system. |

### 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

|  |  |  |
| --- | --- | --- |
| **Standard** | **Mapped Principles** | **Justification** |
| **STD-001-CPP** | 1, 4, 10 | **Validate Input Data (1)** ensures user data is reliable before processing. **Keep It Simple (4)** promotes clarity through explicit data types. **Adopt a Secure Coding Standard (10)** ensures consistency and avoids unsafe implicit conversions. |
| **STD-002-CPP** | 1, 5, 9 | **Validate Input Data (1)** is the core of input verification. **Default Deny (5)** supports rejecting unvalidated or out-of-range inputs. **Use Effective QA Techniques (9)** helps catch validation failures early. |
| **STD-003-CPP** | 7, 8 | **Sanitize Data Sent to Other Systems (7)** prevents buffer overflow attacks from spreading. **Defense in Depth (8)** layers bounds checking with safer functions. |
| **STD-004-CPP** | 3, 7 | **Architect and Design for Security Policies (3)** supports using secure database design patterns. **Sanitize Data Sent to Other Systems (7)** helps prevent SQL injection attacks. |
| **STD-005-CPP** | 8, 9 | **Defense in Depth (8)** involves safe memory management as a foundational protection layer. **Use Effective QA Techniques (9)** enables early detection of use-after-free and memory corruption. |
| **STD-006-CPP** | 1, 9 | **Validate Input Data (1)** cannot rely solely on assertions. **Use Effective QA Techniques (9)** ensures robust runtime input checks are part of secure development. |
| **STD-007-CPP** | 4, 9 | **Keep It Simple (4)** is supported by specific and readable exception handling. **Use Effective QA Techniques (9)** allows errors to be tested and traced effectively. |
| **STD-008-CPP** | 1, 8 | **Validate Input Data (1)** helps avoid input-triggered buffer overflows. **Defense in Depth (8)** applies overlapping safeguards through bounds checking and secure functions. |
| **STD-009-CPP** | 3, 6 | **Architect and Design for Security Policies (3)** enforces safe and validated file access. **Principle of Least Privilege (6)** ensures users only access necessary files. |
| **STD-010-CPP** | 3, 10, 9 | **Architect and Design for Security Policies (3)** applies to cryptographic use of randomness. **Adopt a Secure Coding Standard (10)** promotes proper random number generation. **Use Effective QA Techniques (9)** supports testing random number generators for unpredictability. |

## 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 | 06/13/2025 | Updated Template | Jaden Williams |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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