**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 a critical security principle that involves checking and sanitizing all input from users to ensure it meets expected formats and values. This helps prevent malicious input that could lead to vulnerabilities such as SQL injection, cross-site scripting, and buffer overflow attacks. |
| 1. Heed Compiler Warnings | Compiler warnings often indicate potential flaws in code that could lead to security vulnerabilities or unstable behavior. Developers should pay attention to these warnings and address them promptly, as they may highlight issues such as uninitialized variables, type mismatches, or other risky coding practices. |
| 1. Architect and Design for Security Policies | Security should be integrated into the architecture and design phase of software development. By incorporating security policies and considerations early on, developers can create systems that are inherently more secure. This approach involves defining security requirements, threat modeling, and choosing appropriate security controls that align with the organization's security goals. |
| 1. Keep It Simple | Complexity is the enemy of security. Simple code is easier to understand, audit, and maintain, which reduces the likelihood of introducing security flaws. Adhering to the principle of simplicity means avoiding unnecessary complexity in code and design. |
| 1. Default Deny | The default deny principle ensures that access to resources is denied unless explicitly allowed. This approach minimizes the attack potential by restricting access to only what is necessary and explicitly permitted. |
| 1. Adhere to the Principle of Least Privilege | This principle involves giving users and systems the minimum level of access required to perform their tasks. By restricting permissions, even if an account or process is compromised, the damage that can be done is limited. |
| 1. Sanitize Data Sent to Other Systems | Data sent to other systems, especially those that execute commands or queries, must be sanitized to prevent injection attacks. Sanitizing data involves removing or escaping potentially harmful characters to ensure that input is treated as data, not executable code. |
| 1. Practice Defense in Depth | Defense in depth is a layered security strategy that involves implementing multiple security measures at various levels. This approach ensures that if one security control fails, others are in place to provide protection. It can be put into place by combining different techniques, such as firewalls, intrusion detection systems, encryption, and access controls. |
| 1. Use Effective Quality Assurance Techniques | Effective quality assurance techniques, such as code reviews, static analysis, and security testing, are essential for identifying and fixing security vulnerabilities before software is deployed. QA processes help ensure that security is built into the software development lifecycle. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard provides guidelines and best practices for writing secure code. These standards help developers avoid common pitfalls and vulnerabilities by promoting consistency, security awareness, and the use of well-established coding techniques. |

### 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** |
| --- | --- | --- |
| **Validate Input Data** | STD-001-DAT | Validating input data is essential to prevent unauthorized access and data corruption. Input validation ensures that only properly formatted and expected data is processed, reducing the risk of injection attacks and buffer overflows. |

| **Noncompliant Code** |
| --- |
| This code takes user input and directly uses it to form an SQL query. If the user input contains malicious SQL commands, it can lead to SQL injection. |
| std::string user\_input;  std::cin >> user\_input;  std::string query = "SELECT \* FROM users WHERE username = '" + user\_input + "'";  execute\_query(query); |

| **Compliant Code** |
| --- |
| This compliant example uses a parameterized query to handle user input safely and includes validation to check for valid input characters, preventing SQL injection. |
| std::string user\_input;  std::cin >> user\_input;    if (validate\_input(user\_input)) {  std::string query = "SELECT \* FROM users WHERE username = ?";  execute\_query(query, user\_input);  } else {  std::cout << "Invalid input." << std::endl;  } |

**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** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | S3649: Input Validation | SonarQube detects input validation issues, focusing on user input vulnerabilities such as SQL injection and XSS attacks. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Heed Compiler Warnings** | STD-002-DAV | Compiler warnings often indicate potential flaws or vulnerabilities. Ignoring these warnings can lead to security issues, such as undefined behavior or exploitable bugs. |

| **Noncompliant Code** |
| --- |
| The noncompliant code has a narrowing conversion, which may cause data loss and lead to unexpected behavior or security vulnerabilities. |
| int num = 1000;  char small\_num = num; |

| **Compliant Code** |
| --- |
| The compliant code checks the value before casting, avoiding loss of data and ensuring that the conversion is safe. |
| int num = 1000;  if (num >= std::numeric\_limits<char>::min() && num <= std::numeric\_limits<char>::max()) {  char small\_num = static\_cast<char>(num);  } else {  std::cout << "Number out of range for char type." << std::endl;  } |

**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 | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| GCC | 12 | -Wall | GCC with the -Wall flag enables all warnings, helping developers catch potential issues before code is deployed. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-SCR | Proper handling of strings is crucial in preventing buffer overflows, data corruption, and injection attacks. |

| **Noncompliant Code** |
| --- |
| This code attempts to copy a long string into a smaller buffer, which can cause a buffer overflow and potentially allow attackers to execute arbitrary code. |
| char destination[10];  strcpy(destination, "This is a very long string"); |

| **Compliant Code** |
| --- |
| Using strncpy limits the number of characters copied and prevents buffer overflow by ensuring the buffer size is respected. |
| char destination[10];  strncpy(destination, "This is a very long string", sizeof(destination) - 1); |

**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 | High | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.6 | Strcpy misuse | Cppcheck detects unsafe string handling functions like strcpy that can cause buffer overflows. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-SQL | SQL injection attacks can lead to unauthorized data access and manipulation. Using parameterized queries or prepared statements ensures that input is treated as data, not executable code. |

| **Noncompliant Code** |
| --- |
| This code is vulnerable to SQL injection because it directly incorporates user input into the SQL query. |
| std::string username;  std::cin >> username;  std::string query = "SELECT \* FROM users WHERE username = '" + username + "'";  execute\_query(query); |

| **Compliant Code** |
| --- |
| This compliant code uses parameterized queries to separate the query structure from the input data, preventing SQL injection. |
| std::string username;  std::cin >> username;  std::string query = "SELECT \* FROM users WHERE username = ?";  execute\_query(query, 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** |
| --- | --- | --- | --- | --- |
| Critical | Likely | Low | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | S2077: SQL Injection | Detects unsanitized input being directly passed to SQL queries, mitigating SQL injection risks. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-MPR | Protecting memory allocations and accesses is critical to prevent buffer overflows, memory leaks, and access violations, which can lead to system crashes or exploitable vulnerabilities. |

| **Noncompliant Code** |
| --- |
| If the memory allocation fails, using array can lead to undefined behavior, including crashes or data corruption. |
| int\* array = new int[100];  array[0] = 1; |

| **Compliant Code** |
| --- |
| By checking if array is nullptr, this code avoids dereferencing a null pointer and ensures that memory is allocated successfully before use. |
| int\* array = new (std::nothrow) int[100];  if (array != nullptr) {  array[0] = 1;  } else {  std::cerr << "Memory allocation failed." << std::endl;  } |

**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** |
| --- | --- | --- | --- | --- |
| Critical | Possible | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.23.0 | Memcheck | Valgrind’s memcheck tool is used to detect memory leaks and incorrect memory handling that could lead to vulnerabilities. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-ASR | Assertions are used to check for conditions that should never occur. They help catch logical errors and ensure that the code behaves as expected. |

| **Noncompliant Code** |
| --- |
| Without input validation, this code may result in a division by zero, causing the program to crash. |
| void process(int value) {  int result = 100 / value;  } |

| **Compliant Code** |
| --- |
| The assertion ensures that value is not zero before proceeding, preventing division by zero and maintaining program stability. |
| void process(int value) {  assert(value != 0);  int result = 100 / 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 | Unlikely | Low | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.6 | assertions | Cppcheck can identify improper or dangerous use of assertions that may cause undefined behavior in release builds. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-EXC | Exceptions provide a mechanism for handling runtime errors and unexpected conditions. Proper use of exceptions ensures that errors are caught and handled appropriately. |

| **Noncompliant Code** |
| --- |
| Without exception handling, if the file operation fails, the program will terminate unexpectedly. |
| void readFile(const std::string& filename) {  std::ifstream file(filename);  file.exceptions(std::ifstream::failbit | std::ifstream::badbit);  } |

| **Compliant Code** |
| --- |
| The use of try-catch blocks ensures that exceptions are caught and handled, allowing the program to provide meaningful error messages and continue execution. |
| void readFile(const std::string& filename) {  std::ifstream file;  try {  file.open(filename);  file.exceptions(std::ifstream::failbit | std::ifstream::badbit);  } catch (const std::ios\_base::failure& e) {  std::cerr << "File operation failed: " << e.what() << std::endl;  }  } |

**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 | Medium | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.6 | Exception safety | Cppcheck evaluates the correct use of exceptions, ensuring they are used properly to handle errors. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Proper Use of Mutexes | STD-008-MUT | In multithreaded applications, proper use of mutexes is essential to prevent race conditions and ensure data consistency. Mutexes allow threads to access shared resources safely. |

| **Noncompliant Code** |
| --- |
| This code does not use mutexes, which can lead to race conditions. |
| int counter = 0;  void increment() {  counter++;  } |

| **Compliant Code** |
| --- |
| The use of std::lock\_guard with a mutex ensures that only one thread can access counter at a time, preventing race conditions and maintaining data integrity. |
| int counter = 0;  std::mutex counter\_mutex;  void increment() {  std::lock\_guard<std::mutex> lock(counter\_mutex);  counter++;  } |

**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** |
| --- | --- | --- | --- | --- |
| Critical | Possible | High | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ThreadSanitizer | 10.2 | mutex | ThreadSanitizer helps detect race conditions, ensuring mutexes are properly used in multi-threaded environments. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secure Data Encryption | STD-009-ENC | Encrypting sensitive data, both at rest and in transit, is crucial for protecting data confidentiality and integrity and using strong encryption algorithms and proper key management practices minimizes the risk of data breaches. |

| **Noncompliant Code** |
| --- |
| Storing passwords in plaintext exposes them to unauthorized access if the file is compromised. |
| std::string password = "password1234";  std::ofstream file("passwords.txt");  file << password; |

| **Compliant Code** |
| --- |
| This code encrypts the password before storing it, ensuring that sensitive information is protected even if the storage medium is compromised. |
| std::string encrypt(const std::string& plaintext) {  }  std::string password = " password1234";  std::string encrypted\_password = encrypt(password);  std::ofstream file("passwords.txt");  file << encrypted\_password; |

**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** |
| --- | --- | --- | --- | --- |
| Critical | Likely | High | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | S2078: Data Encryption | Detects the proper implementation of data encryption functions and safe storage of sensitive data. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Use of Safe Libraries | STD-010-LIB | Using safe and trusted libraries minimizes vulnerabilities introduced by insecure or outdated third-party code. Trusted libraries are maintained, tested, and patched for security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This code relies on an outdated cryptographic library, which may have known vulnerabilities that attackers can exploit. |
| void encryptData() {  old\_crypto\_encrypt("data");  } |

| **Compliant Code** |
| --- |
| The compliant code uses the OpenSSL library, a well-maintained and secure cryptographic library, to perform encryption operations to ensure data security. |
| void encryptData() {  EVP\_EncryptInit(...);  EVP\_EncryptUpdate(...);  EVP\_EncryptFinal(...);  } |

**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** |
| --- | --- | --- | --- |
| OWASP Dependency-Check | 10 | library vulnerabilities | Scans libraries for known vulnerabilities and ensures only safe and updated versions 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 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.

Incorporating security checks into the DevOps process requires integrating these tools within the CI/CD pipeline. For Green Pace, tools like SonarQube and Cppcheck will be integrated into the build stage, ensuring that code meets security standards before it can be merged. Automation will be part of the build process, with checks run on every commit. This ensures that violations or vulnerabilities are detected early, before deployment.

Additionally, Valgrind and ThreadSanitizer will be used in the testing stage to catch memory-related issues and race conditions in a controlled test environment. Dependency-Check will be run at the beginning of each project to ensure libraries used in the build are up to date and free of known vulnerabilities.

### 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-DAT | High | Likely | Medium | High | 5 |
| STD-002-DAV | Medium | Possible | Medium | Medium | 3 |
| STD-003-SCR | High | Unlikely | High | High | 4 |
| STD-004-SQL | Critical | Likely | Low | High | 5 |
| STD-005-MPR | Critical | Possible | Medium | High | 5 |
| STD-006-ASR | Medium | Unlikely | Low | Medium | 2 |
| STD-007-EXC | Medium | Likely | Medium | Medium | 3 |
| STD-008-MUT | Critical | Possible | High | High | 5 |
| STD-009-ENC | Critical | Likely | High | High | 5 |
| STD-010-LIB | High | Possible | Medium | High | 4 |

### 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 encrypting data when it is stored, whether in databases, file systems, or backups. Sensitive data must be encrypted using algorithms such as AES-256 to prevent unauthorized access, ensuring that even if the storage medium is compromised, the data remains secure. |
| Encryption in flight | Encryption in flight applies to data transmitted across networks. Secure protocols like HTTPS or TLS must be used to encrypt data between clients and servers. This policy is critical to ensure that data cannot be intercepted or modified during transmission. |
| Encryption in use | Encryption in use refers to protecting data when it is being actively processed in memory. Techniques such as homomorphic encryption may be used for highly sensitive operations, though this is an evolving field. This policy applies to highly sensitive applications where data integrity is paramount during processing. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identity of users accessing the system. The policy mandates the use of multi-factor authentication (MFA) for all user logins to prevent unauthorized access, ensuring that credentials alone cannot be used for authentication. |
| Authorization | Authorization determines what resources users can access once authenticated. This policy ensures that users are only granted access to the resources they need based on the principle of least privilege, limiting the potential damage in the event of a security breach. |
| Accounting | Accounting, or auditing, involves tracking user actions and changes in the system. The policy mandates logging all access to sensitive resources and changes made by users. These logs must be reviewed regularly to ensure compliance and to detect any suspicious activity early. |

**\***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
* **STD-001-DAT (Validate Input Data)** is supported by Principle 1: Validate Input Data and Principle 7: Sanitize Data Sent to Other Systems. Both principles emphasize input validation and sanitization to protect the system from malicious input.
* **STD-002-DAV (Heed Compiler Warnings)** is supported by Principle 2: Heed Compiler Warnings. This ensures that developers pay attention to warnings that may indicate vulnerabilities.
* **STD-003-SCR (String Correctness)** is supported by Principle 4: Keep It Simple. Ensuring strings are handled correctly reduces the complexity and risks of buffer overflows.
* **STD-004-SQL (SQL Injection)** is supported by Principle 1: Validate Input Data and Principle 7: Sanitize Data Sent to Other Systems to protect against SQL injection attacks.
* **STD-005-MPR (Memory Protection)** is supported by Principle 8: Practice Defense in Depth, as it prevents critical memory-related vulnerabilities that can lead to deeper attacks.
* **STD-006-ASR (Assertions)** is supported by Principle 3: Architect and Design for Security Policies, ensuring that potential issues are caught early in the development lifecycle.

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 09/18/2024 | Coding Standards | Mark Hall |  |
| 1.2 | 10/10/2024 | Final Report | Mark Hall |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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