**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_wmmeikdibh0b)

[Purpose 2](#_lbz44jisjvtg)

[Scope 2](#_it9q78b05596)

[Module Three Milestone 2](#_yy2cjex4dzpc)

[Ten Core Security Principles 2](#_3em9vkdzi4be)

[C/C++ Ten Coding Standards 3](#_s4ftp2aekg8y)

[Coding Standard 1 4](#_i513o5tvictk)

[Coding Standard 2 5](#_xahal5tjllut)

[Coding Standard 3 6](#_nc8d46jlwd)

[Coding Standard 4 7](#_vq76w0dam804)

[Coding Standard 5 8](#_82zkffn3hwhs)

[Coding Standard 6 9](#_dxtrd94n2bwn)

[Coding Standard 7 10](#_frbpcr129xl3)

[Coding Standard 8 11](#_v8esahnaa8zs)

[Coding Standard 9 13](#_1r1ht4p5s2g)

[Coding Standard 10 14](#_wk6c7ht9qc5m)

[Defense-in-Depth Illustration 15](#_50juhrmcgob2)

[Project One 15](#_cgucfqqi4b8d)

[1.](#_docdyxrczcfc) Revise the C/C++ Standards 15

[2.](#_psjrsraabqeq) Risk Assessment 15

[3.](#_c6nq0ijl7sd3) Automated Detection 15

[4.](#_dmg5l8gwnv33) Automation 15

[5.](#_acwhjaaactmv) Summary of Risk Assessments 16

[6.](#_s464owi0fl22) Create Policies for Encryption and Triple A 16

[7.](#_4d1bmn41clgr) Map the Principles 17

[Audit Controls and Management 18](#_wgxvjfvjb9q8)

[Enforcement 18](#_4xvsdbkub9qf)

[Exceptions Process 18](#_txattj7np74a)

[Distribution 19](#_f1tpg0tymwd1)

[Policy Change Control 19](#_hhq4m7xddfnu)

[Policy Version History 19](#_zdo3ynf3f40j)

[Appendix A Lookups 19](#_vzcu19wwz6wm)

[Approved C/C++ Language Acronyms 19](#_58m7mypkupb1)

## 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 user input must be checked before processing. Never trust data from external sources without verification. Implement strict validation rules for format, length, and type. This prevents injection attacks and malformed data issues. Proper validation stops malicious input from compromising systems. |
| 1. Heed Compiler Warnings | Compiler warnings often reveal potential security flaws. Address all warnings during development rather than ignoring them. Many vulnerabilities start as minor warnings left unresolved. Treat warnings as serious indicators of possible problems. Fixing warnings early improves code safety. |
| 1. Architect and Design for Security Policies | Security must be built into systems from the start. Design architectures with protection mechanisms in mind. Follow established security frameworks and standards. Good design prevents vulnerabilities rather than patching them later. Security policies guide proper implementation. |
| 1. Keep It Simple | Complex code creates more opportunities for mistakes. Simple designs are easier to secure and maintain. Avoid unnecessary features that increase attack surfaces. Clear, straightforward implementations reduce security risks. Simplicity makes vulnerabilities easier to spot. |
| 1. Default Deny | Systems should block access by default. Only allow explicitly permitted actions and users. This minimizes exposure to potential threats. Default deny provides a strong security baseline. It ensures nothing gets through unless properly authorized. |
| 1. Adhere to the Principle of Least Privilege | Users and systems should have minimum necessary access. Grant only the permissions required for specific tasks. Restrict elevated privileges to essential cases. This limits damage from compromised accounts. Least privilege reduces overall system vulnerability. |
| 1. Sanitize Data Sent to Other Systems | Clean all data before passing it between components. Remove potentially harmful characters and constructs. Ensure proper encoding for different contexts. Sanitization prevents injection attacks across system boundaries. It maintains data integrity throughout processing. |
| 1. Practice Defense in Depth | Implement multiple layers of security controls. No single protection should be relied upon alone. Diverse safeguards create redundancy against failures. Layered defenses make systems more resilient to attacks. Depth ensures backup protections exist. |
| 1. Use Effective Quality Assurance Techniques | Rigorous testing finds vulnerabilities before deployment. Combine automated scans with manual code reviews. Test for both functionality and security requirements. Quality assurance catches flaws that might otherwise be missed. Thorough testing is essential for secure systems. |
| 1. Adopt a Secure Coding Standard | Standardized coding practices ensure consistent implementation of security controls. Teams should select and enforce appropriate guidelines for their technology stack. Standards help developers avoid common vulnerabilities during implementation. Regular training keeps coding practices aligned with current threats. |

### 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 | following the one-definition rule |

| **Noncompliant Code** |
| --- |
| The code fails to validate integer input ranges. |
| int userValue;  std::cin >> userValue; |

| **Compliant Code** |
| --- |
| The code properly validates integer input ranges. |
| int userValue;  if (!(std::cin >> userValue) || userValue < MIN\_RANGE || userValue > MAX\_RANGE) {  std::cout << "Invalid input\n";  return;  } |

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

| **Principles(s):** Principle 3 (Input Validation) - Ensures only expected data types are processed, preventing type confusion vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | S3685 | Detects unvalidated type conversions |
| Cppcheck | 2.10 | invalidPointerCast | Flags risky type casts |
| Coverity | 2023.03 | TYPE\_MISMATCH | Static analysis for type safety |
| Clang-Tidy | 17 | clang-analyzer-core.Undefined | Checks type operations |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD–002-CPP] | Do not read unprompted memory |

| **Noncompliant Code** |
| --- |
| The code reads uninitialized memory which may contain sensitive data. |
| char secretData[100];  printf("%s", secretData); |

| **Compliant Code** |
| --- |
| The code properly initializes memory before reading it. |
| char secretData[100] = {0};  if(needsProcessing) {  getApprovedData(secretData, sizeof(secretData));  printf("%s", secretData);  } |

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

| **Principles(s):** Principle 5 (Memory Safety) - Prevents buffer overflows and string handling errors. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Very Likely | Medium | Critical | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.8 | bufferAccessOutOfBounds | Detects string boundary violations |
| PVS-Studio | 7.23 | V512 | Finds unsafe string operations |
| GCC | 12.2 | -Wformat-overflow | Warns about format string issues |
| Flawfinder | 2.0.19 | String handling checks | Identifies risky string functions |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-002-CPP] | Do not access an object outside of its lifetime |

| **Noncompliant Code** |
| --- |
| The code accesses a string after its memory has been freed. |
| char\* str = new char[100];  delete[] str;  strcpy(str, "unsafe"); |

| **Compliant Code** |
| --- |
| The code ensures the string exists during access. |
| char\* str = new char[100];  strcpy(str, "safe");  delete[] str; |

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

| **Principles(s):** Principle 7 (Parameterized Queries) - Mandates prepared statements to block injection. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SQLMap | 1.7 | Injection detection | Tests for SQLi vulnerabilities |
| Checkmarx | 2023.3 | SQL\_Injection | Static code analysis |
| SonarQube | 9.9 | S3649 | SQL query validation |
| Veracode | 22.12 | CWE-89 | Scans for injection patterns |

#### 

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Prevent SQL Injection |

| **Noncompliant Code** |
| --- |
| The code directly concatenates user input into SQL queries. |
| std::string query = "SELECT \* FROM users WHERE name = '" + userInput + "'";  sqlite3\_exec(db, query.c\_str(), callback, 0, &err); |

| **Compliant Code** |
| --- |
| The code uses parameterized queries to prevent injection. |
| std::string query = "SELECT \* FROM users WHERE name = ?";  sqlite3\_stmt\* stmt;  sqlite3\_prepare\_v2(db, query.c\_str(), -1, &stmt, 0);  sqlite3\_bind\_text(stmt, 1, userInput.c\_str(), -1, SQLITE\_STATIC); |

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

| **Principles(s):** Principle 3 (Input Validation) and Principle 7 (Defense in Depth) apply here. Input validation ensures all SQL queries are sanitized before processing. Defense in depth means using multiple layers like parameterized queries alongside input checks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | Critical | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | SQLInjectionCheck | Finds concatenated SQL queries |
| Checkmarx | 2023.3 | SQL\_Injection | Scans for unsafe query patterns |
| Cppcheck | 2.10 | sqlInjection | Basic SQLi detection in C++ |
| Visual Studio | 2022 | C26800 | Warns about raw string queries |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Detect and manage memory allocation errors |

| **Noncompliant Code** |
| --- |
| The code does not check if memory allocation succeeded. |
| int\* data = new int[1000];  data[0] = 42; |

| **Compliant Code** |
| --- |
| The code verifies memory allocation before use. |
| int\* data = new(std::nothrow) int[1000];  if (data == nullptr) {  std::cerr << "Memory allocation failed";  return; |

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

| **Principles(s):** Principle 4 (Memory Safety) and Principle 8 (Least Privilege). Memory safety prevents buffer overflows. Least privilege restricts memory access to only what’s needed. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Very Likely | Medium | Critical | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.20 | Memcheck | Detects memory leaks |
| Clang-Tidy | 17 | clang-analyzer-core | Finds unsafe memory ops |
| Cppcheck | 2.10 | arrayIndexOutOfBounds | Checks buffer overflows |
| Coverity | 2023.6 | MEMORY\_LEAK | Advanced heap analysis |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Use static assertion to test a constant expression |

| **Noncompliant Code** |
| --- |
| The code uses runtime check for constant expression validation. |
| const int buffer\_size = 1024;  if (buffer\_size < 100) {  printf("Buffer too small");  } |

| **Compliant Code** |
| --- |
| The code uses static assertion for compile time checking. |
| const int buffer\_size = 1024;  static\_assert(buffer\_size >= 100, "Buffer too small"); |

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

| **Principles(s):** Principle 4 (Defense in Depth) and Principle 7 (Fail Securely) map to this standard. Assertions act as runtime checks to validate assumptions, providing an additional layer of defense. If an assertion fails, the program should terminate safely rather than continue with invalid state. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | assertWithSideEffect | Detects dangerous assertions with side effects |
| SonarQube | 9.9 | S3925 | Flags assert() misuse in C++ |
| Clang-Tidy | 17 | bugprone-assert-side-effect | Catches asserts that may alter program state |
| Visual Studio | 2022 | C6001 | Warns about unused assertion results |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Do not abruptly terminate the program |

| **Noncompliant Code** |
| --- |
| The code calls exit when encountering an error condition. |
| if (file == nullptr) {  exit(1);  } |

| **Compliant Code** |
| --- |
| The code throws an exception for error conditions. |
| if (file == nullptr) {  throw std::runtime\_error("File open failed");  } |

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

| **Principles(s):** Principle 5 (Least Privilege) and Principle 9 (Error Handling). Exceptions must not expose sensitive data (least privilege) and should provide meaningful error context without crashing (error handling). |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | Critical | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | exceptThrowInNoexcept | Detects throws in noexcept functions |
| PVS-Studio | 7.23 | V509 | Finds empty catch blocks |
| Coverity | 2023.03 | MISSING\_EXCEPT | Flags uncaught exceptions |
| Clang-Tidy | 17 | misc-throw-by-value-catch-by-reference | Ensures proper exception handling |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Resource Management | STD-008-CPP | Release acquired resources |

| **Noncompliant Code** |
| --- |
| The code fails to release allocated file resources. |
| FILE\* file = fopen("data.txt", "r");  if (file) {  process\_file(file);  } |

| **Compliant Code** |
| --- |
| The code ensures proper resource cleanup. |
| FILE\* file = fopen("data.txt", "r");  if (file) {  process\_file(file);  fclose(file);  } |

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

| **Principles(s):** Principle 4 (Defense in Depth) and Principle 7 (Least Privilege). Memory protection aligns with layered security controls and restricting access to critical memory regions. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | invalidFree | Detects unsafe memory operations |
| Valgrind | 3.20 | Memcheck | Identifies leaks and invalid accesses |
| Clang-Tidy | 17 | clang-analyzer-core | Checks pointer misuse |
| SonarQube | 9.9 | C/C++ Memory Safety Rule | Scans for buffer overflows |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Thread Safety | STD-009-CPP | Protect shared data with mutex locks |

| **Noncompliant Code** |
| --- |
| The code accesses shared data without synchronization. |
| The code accesses shared data without synchronization.  int counter = 0;  void increment() {  counter++;  } |

| **Compliant Code** |
| --- |
| The code uses mutex to protect shared access. |
| std::mutex mtx;  int counter = 0;  void increment() {  std::lock\_guardstd::mutex lock(mtx);  counter++;  } |

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

| **Principles(s):** Principle 3 (Fail Securely) and Principle 9 (Continuous Validation). Assertions enforce invariants and fail fast during development. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 16 | alpha.core.Assert | Flags incorrect assert usage |
| Coverity | 2023.06 | ASSERT\_SIDE\_EFFECTS | Catches assertions with side effects |
| SonarQube | 9.9 | S2699 | Validates assertion logic |
| PVS-Studio | 7.26 | V547 | Detects always-true/false assertions |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Error Handling | STD-010-CPP | Handle all possible error conditions |

| **Noncompliant Code** |
| --- |
| The code ignores potential error conditions. |
| void processFile(const char\* filename) {  FILE\* f = fopen(filename, "r");  readContents(f);  } |

| **Compliant Code** |
| --- |
| The code properly checks and handles errors. |
| void processFile(const char\* filename) {  FILE\* f = fopen(filename, "r");  if (!f) {  throw std::runtime\_error("File open failed");  }  readContents(f);  fclose(f);  } |

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

| **Principles(s):** Principle 2 (Complete Mediation) and Principle 6 (Economy of Mechanism). Exceptions ensure errors are handled explicitly without exposing internals. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | exceptThrowInNoexcept | Finds noexcept violations |
| Clang-Tidy | 17 | bugprone-exception-escape | Detects uncaught exceptions |
| SonarQube | 9.9 | S5034 | Validates try-catch blocks |
| Visual Studio | 2022 | C++ Core Guidelines Checks | Flags unsafe exception patterns |

### 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 | High | Likely | Low | High | 1 |
| STD-003-CPP | Medium | Unlikely | Low | Medium | 2 |
| STD-004-CPP | High | Likely | High | High | 1 |
| STD-005-CPP | Medium | Possible | Medium | Medium | 2 |
| STD-006-CPP | Low | Rare | Low | Low | 3 |
| STD-007-CPP | High | Likely | Medium | High | 1 |
| STD-008-CPP | High | Likely | Low | High | 1 |
| STD-009-CPP | Medium | Possible | Medium | Medium | 2 |
| STD-010-CPP | High | Likely | High | High | 1 |
| AAA-001 | High | Certain | Medium | High | 1 |
| AAA-002 | High | Likely | Medium | High | 1 |
| AAA-003 | Medium | Likely | Low | Medium | 2 |
| SYS-001 | Critical | Certain | High | Critical | 1 |

### 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 | What it is: Encrypts stored data (databases, files).  How to apply: Use AES-256 for files, TDE for databases.  Why: Prevents data theft if storage is compromised. |
| Encryption in flight | What it is: Secures data during network transfer.  How to apply: Enforce TLS 1.3 for all APIs/web traffic.  Why: Stops man-in-the-middle attacks. |
| Encryption in use | What it is: Protects data during processing (e.g., homomorphic encryption).  How to apply: Implement secure enclaves for sensitive operations.  Why: Prevents memory scraping attacks. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | What it is: Verifies user identities.  How to apply: Require MFA + biometrics for all systems.  Why: Blocks unauthorized access. |
| Authorization | What it is: Controls user permissions.  How to apply: Role-based access control (RBAC).  Why: Limits users to least-privilege access. |
| Accounting | What it is: Logs user actions.  How to apply: Centralize logs with SIEM tools.  Why: Enables audit trails for compliance. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 05/20/2025 | Completed Milestone 1 | Evann Hopkins |  |
| 1.2 | 05/28/2025 | Completed Milestone 2 | Evann Hopkins |  |
| Final | 06/11/2025 | Finished filling our template | Evann Hopkins |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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