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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_heading=h.g33krgleh4ho)

[Purpose 2](#_heading=h.h3ff1e7vo0du)

[Scope 2](#_heading=h.hnbt7ja1zqm)

[Module Three Milestone 2](#_heading=h.rrrszv8fllz3)

[Ten Core Security Principles 2](#_heading=h.8gva9xj4ln6f)

[C/C++ Ten Coding Standards 3](#_heading=h.n7unhuwjyrwr)

[Coding Standard 1 4](#_heading=h.b6h9tj2kfcb2)

[Coding Standard 2 5](#_heading=h.a7y28dnf9ak7)

[Coding Standard 3 6](#_heading=h.1ursgcy2px04)

[Coding Standard 4 7](#_heading=h.5cb401ugbnz)

[Coding Standard 5 8](#_heading=h.6hqewujyj3ua)

[Coding Standard 6 9](#_heading=h.oi42pr4z2igc)

[Coding Standard 7 10](#_heading=h.v8pzu1c2jl58)

[Coding Standard 8 11](#_heading=h.wgqd9j65cfrd)

[Coding Standard 9 13](#_heading=h.knpzx5a5gjp8)

[Coding Standard 10 14](#_heading=h.x8kpxw18qtg3)

[Defense-in-Depth Illustration 15](#_heading=h.uf8n4s10msii)

[Project One 15](#_heading=h.shvm7xshikq5)

[1.](#_heading=h.z1kxbgxynnt0) Revise the C/C++ Standards 15

[2.](#_heading=h.uxfi81ljwrhd) Risk Assessment 15

[3.](#_heading=h.1tapyasvuvb8) Automated Detection 15

[4.](#_heading=h.dmw4ewsuqj2n) Automation 15

[5.](#_heading=h.xupk01u3d5e) Summary of Risk Assessments 16

[6.](#_heading=h.69rpqybqavps) Create Policies for Encryption and Triple A 16

[7.](#_heading=h.tjjtjhx9ncrq) Map the Principles 17

[Audit Controls and Management 18](#_heading=h.73kudsvyjxrc)

[Enforcement 18](#_heading=h.ao6d755wsm1y)

[Exceptions Process 18](#_heading=h.b0b6zxw0s070)

[Distribution 19](#_heading=h.rqcju65kkbm2)

[Policy Change Control 19](#_heading=h.og0s4lke6xf2)

[Policy Version History 19](#_heading=h.fjwzk9b8g0mc)

[Appendix A Lookups 19](#_heading=h.bxm1fa8d8dpt)

[Approved C/C++ Language Acronyms 19](#_heading=h.cngwg4myr4n0)

## 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 | Input from users or external sources must be validated to ensure it meets expected format and range. |
| 1. Heed Compiler Warnings | Compiler warnings often indicate issues or vulnerabilities. They should be treated as errors and resolved before deployment. |
| 1. Architect and Design for Security Policies | Security must be considered during all phases of design and development. A secure architecture helps prevent vulnerabilities later on in the project. |
| 1. Keep It Simple | Simplicity in code design improves maintainability. Complex programming makes errors more difficult to debug. |
| 1. Default Deny | Access should be denied by default unless explicitly granted in order to reduce access by unauthorized actions or users. |
| 1. Adhere to the Principle of Least Privilege | Every action or user should operate with minimum privileges. This limits the damage of a compromised process. |
| 1. Sanitize Data Sent to Other Systems | Any data sent to external systems should be sanitized to prevent injection or other vulnerabilities. |
| 1. Practice Defense in Depth | Multiple layers of security ensure that if one mechanism fails, others are in place to protect the system and control potential damage. |
| 1. Use Effective Quality Assurance Techniques | Rigorous testing, static code analysis, and peer reviews help to identify vulnerabilities early in the development cycle. |
| 1. Adopt a Secure Coding Standard | Secure coding standards ensure code is written consistently by every member of the project and thus creates a uniformity to the project that increases its maintainability. |

### 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 | Explicit Type Declaration |

| **Noncompliant Code** |
| --- |
| Implicit type conversion can cause truncation. |
| int a = 300;  char c = a; |

| **Compliant Code** |
| --- |
| Use explicit casting or avoid narrowing conversions. |
| int a = 300;  char c = static\_cast<char>(a); |

| **Principles(s):** 1, 4  Principle one deals with validating input. While this doesn’t directly deal with external input, it is still a good idea to ensure the system is utilizing the data type that is expected. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 14.0 | cert-dcl30-cpp | enforces explicit casts |
| SonarQube | 9.9 | cert-dcl30-cpp | flags implicit narrowing conversions |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Numeric Overflow |

| **Noncompliant Code** |
| --- |
| Math operations can cause numeric overflow if operations produce results outside of the data type’s range. |
| char a;  a = 200 + 200; |

| **Compliant Code** |
| --- |
| Ensure that the operation’s result will be within the data type’s range. |
| char a;  char b = 200;  char c = 200;  if((std::numeric\_limits<char>::max() - b) > c){  // handle error  }  else {  a = b + c;  } |

| **Principles(s):** 1, 9  Validating input data prevents possible overflow by ensuring values won’t produce overflow ahead of time.  Quality assurance can provide test cases that ensure that the operations being performed provide expected output. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12 | warning: icpp19 | warns if operations may exceed type ranges |
| Coverity | 2024.03 | Integer overflow | Identifies possible overflows |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Ensure string has sufficient space |

| **Noncompliant Code** |
| --- |
| The loop copies data from src to dest. However, because the loop does not account for the null-termination character, it may be incorrectly written 1 byte past the end of dest. |
| void copy(size\_t n, char src[n], char dest[n]) {  size\_t i;    for (i = 0; src[i] && (i < n); ++i) {  dest[i] = src[i];  }  dest[i] = '\0';  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the loop termination condition is modified to account for the null-termination character that is appended to dest: |
| void copy(size\_t n, char src[n], char dest[n]) {  size\_t i;    for (i = 0; src[i] && (i < n - 1); ++i) {  dest[i] = src[i];  }  dest[i] = '\0';  } |

| **Principles(s):** 1, 7  Validating input data ensures that the string will be valid sized before using it.  Sanitizing data can ensure that strings being passed are sufficient before being used in other systems. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | cert-err34-cpp | Detects error in buffer copies |
| Clang-Static Analyzer | 21.0.0 | core.BufferOverflow | Detects buffer risks |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Utilize parameterized queries when interacting with user input |

| **Noncompliant Code** |
| --- |
| Directly using user input in SQL queries is unsafe. |
| std::string query = "SELECT \* FROM users WHERE name = '" + username + "'";  executeSQL(query); |

| **Compliant Code** |
| --- |
| Use parameterized queries. |
| std::string query = "SELECT \* FROM users WHERE name = ?";  prepareAndExecute(query, username); |

| **Principles(s):** 1, 7, 8  Validating input data ensures all SQL inputs to be checked.  Sanitizing data sent to other systems limits the effect that injected data has on the entire system.  Defense-in-depth adds additional layers to protect the system from such attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OWASP ZAP SAST | 2.11.1 | Injection | Static analysis |
| SonarQube | 9.9 | cpp:S3649 | Identifies non-parameterized SQL |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Do not access freed memory |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, s is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. Typically, dynamic memory allocations and deallocations are far removed, making it difficult to recognize and diagnose such problems |
| struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the dynamically allocated memory is not deallocated until it is no longer required. |
| struct S {  void f();  };  void g() noexcept(false) {  S \*s = new S;  // ...  s->f();  delete s;  } |

| **Principles(s):** 3, 6  Architect and design for security ensures rules are in the design to protect the system.  Defense-in-depth utilizes compile-time and run-time measures for protecting the system |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2024.03 | Use after free | Find instances of deleted memory being accessed |
| Clang-Tidy | 14.0 | cert-env33-cpp | Warns user if memory is accessed after being freed |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Do not use assertions in release code |

| **Noncompliant Code** |
| --- |
| Using assert for input validation in production. |
| assert(userAge > 0); |

| **Compliant Code** |
| --- |
| Use runtime validation that is always active. |
| if (userAge <= 0) {  throw std::invalid\_argument("Age must be positive");  } |

| **Principles(s):** 2, 9  Listening to compiler warnings ensures that assertions are not ignored during release.  Effective quality assurance techniques promote the use of asserts in test cases instead of in deliverable code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | cpp:S2643 | Flag asserts |
| Cppcheck | 2.12 | warning:assert | Warns if assert is used |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Handle all exceptions |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  try {  f();  } catch (...) {  // Handle error  }  } |

| **Principles(s):** 3,9  Architect and design for security allows for a precise exception-handling model.  Effective quality assurance ensures that all exceptions will be handled during testing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | cpp:S5970 | Flags uncaught exceptions |
| Cppcheck | 2.12 | warning:exception | Warns on uncaught exceptions |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Data Values | STD-008-CPP | Initialize All Variables |

| **Noncompliant Code** |
| --- |
| Uses a variable before initializing it. |
| int total;  std::cout << total << std::endl; |

| **Compliant Code** |
| --- |
| Always initialize variables |
| int total = 0;  std::cout << total << std::endl; |

| **Principles(s):** 1, 9  Validating input data ensures that the values being used will correctly initialize a variable.  Effective quality assurance can utilize debugging to ensure variables are never uninitialized. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | High | Low | Low | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 14.0 | cert-dcl21-cpp | Finds uninitialized variables |
| Cppcheck | 2.12 | warning:uninitvar | Warns on uninitialized variables |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory Management | STD-009-CPP | Use smart pointers when applicable |

| **Noncompliant Code** |
| --- |
| Uses a regular pointer without ownership control. |
| MyClass\* obj = new MyClass(); |

| **Compliant Code** |
| --- |
| Use std::unique\_ptr for ownership. |
| std::unique\_ptr<MyClass> obj = std::make\_unique<MyClass>(); |

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

| **Principles(s):** 4, 6  Keeping it simple helps explain how smart pointers are preferable over manual creation and deletion of memory utilizing code space.  The principle of least privilege restricts direct memory manipulation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | cpp:S4436 | Flags “new” with pointers |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Miscellaneous | STD-010-CPP | Ensure value-returning functions return a value from every exit path |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the programmer forgot to return the input value for positive input, so not all code paths return a value. |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, all code paths now return a value. |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  return a;  } |

| **Principles(s):** 3, 9  Architect and design for security ensures that all functions will be designed prior to implementation and thus returns will be identified prior.  Effective quality assurance can utilize peer reviews to ensure the developer implements this correctly. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2024.03 | Missing return statement | Detects functions without returns |
| SonarQube | 9.9 | cpp:S2589 | Flags missing returns |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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.

During the build phase of the DevOps process, developers will utilize tools such as Clang-Tidy and Cppcheck within the development environment in order to catch issues early to reduce technical debt. During the verify and test stage, automated testing will ensure that the quality assurance standards set in this document are upheld and can provide instant feedback for developers on the issues outlined here. During the Production phase, specifically the maintain and stabilize stage, the team will utilize a continuous integration platform such as Jenkins and software such as SonarQube to ensure that any updates to the software fall in line with the rules and policies stated within this document.

### 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 | Medium | Medium | Low | Medium | L3 |
| STD-002-CPP | High | Medium | Medium | High | L2 |
| STD-003-CPP | High | High | Low | Medium | L2 |
| STD-004-CPP | High | Medium | Medium | High | L1 |
| STD-005-CPP | High | Medium | Medium | Medium | L2 |
| STD-006-CPP | Low | Medium | Low | Low | L3 |
| STD-007-CPP | Medium | Medium | Low | Medium | L2 |
| STD-008-CPP | Low | High | Low | Low | L3 |
| STD-009-CPP | High | High | Medium | High | L1 |
| STD-010-CPP | Medium | Low | Low | Low | L3 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

### 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 | Encrypt any data that is stored on a device or server. It is usually performed through encrypting all of the data on the medium. Disk encryption can be performed with BitLocker or LUKS. Database encryption can be performed with Transparent Data Encryption. This level of encryption is used to prevent sensitive data from being accessed even if the medium it is stored on is accessed. It should be used on any hardware/server medium that stores sensitive data. |
| Encryption in flight | Encrypt any data that is traveling across a network i.e. from a client to a server or amongst services. TLS, SSH, and VPNs can be used for safely transferring data. This is used to protect data that may be transferred from one place to another and protects from man-in-the-middle attacks. It should be used for any API endpoint connection or client-server interactions. |
| Encryption in use | Encrypt any data that is currently being used by a device or application. This differs from encryption at rest due to the medium that it is stored on. Encryption at rest refers to hard disks/drives that store backup data. Encryption in use refers to data that is currently being used by a device/application’s memory. This is important in case whatever device is utilizing the data or the network the device is on is breached/compromised. This is often implemented through hardware on the specific device that can help to maintain this level of security, but cloud VMs also have in-use encryption capabilities. Certain applications can be designed as well to encrypt any data being utilized. This type of encryption is usually utilized for high-sensitivity material such as financial data or health records. It is used when security of the material is a top priority and no risk can be taken. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Verifies who the user is. This is often implemented by using credentials in a login system. A more secure and common implementation is Multi-factor authentication. By utilizing this, it helps to ensure that only legitimate users can access data. |
| Authorization | This determines what each user or service is allowed to access. A common implementation of this is Role-Based Access Control. This ensures that only specific users or services are allowed to access specific files or functions of the system. This method represents the idea of least privilege. Only specific data/functions should be accessible to those with access. This is implemented if the system is designed to interact with other systems or if the system has a user hierarchy. |
| Accounting | This is used to track who does what within the system. Logging is an important part of security because it allows solutions/fixes to be developed faster by tracking exactly where/what/who caused it. There are many different logging softwares that can be implemented to enact this functionality. It is commonly implemented when there are many users for a system i.e. enterprise businesses. This ensures that users can be held responsible/accountable for their actions such as making specific changes to a database, adding a new user, or accessing a file. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 05/25/2025 | Completed 10 core security principles and 10 security standards | Brady Weber | [Insert text.] |
| 3.0 | 06/12/2025 | Completed the remaining document | Brady Weber | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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