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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | All data entering the system needs validation. This principle keeps bad data from sneaking in and causing issues. |
| 1. Heed Compiler Warnings | Warnings from the compiler often hint at hidden problems. Paying attention to them is essential for preventing small mistakes from becoming big security risks. |
| 1. Architect and Design for Security Policies | The architecture and design should focus on security. It's about building secure foundations instead of trying to fix vulnerabilities later. |
| 1. Keep It Simple | Simplicity is underrated in security. Overcomplicated code can hide vulnerabilities, making simple and clear designs safer. |
| 1. Default Deny | Access should be denied unless explicitly allowed. This approach minimizes unauthorized access by keeping everything closed off by default. |
| 1. Adhere to the Principle of Least Privilege | Each part of the system should have only the permissions it needs. This prevents unnecessary access that could cause security risks. |
| 1. Sanitize Data Sent to Other Systems | Data moving between systems should be sanitized to avoid security issues like injection attacks. |
| 1. Practice Defense in Depth | This is all about layers. Multiple defenses keep the system protected, even if one fails. |
| 1. Use Effective Quality Assurance Techniques | Testing is key. Catching errors and bugs early means fewer security risks in production. |
| 1. Adopt a Secure Coding Standard | Having a consistent coding standard improves code quality and prevents common security pitfalls. |

### 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 | This standard ensures that variables and data types match the expected format, preventing errors and ensuring the application behaves as expected. |

| **Noncompliant Code** |
| --- |
| Uses atoi, which doesn’t validate if userInput contains valid numeric data. |
| int age = atoi(userInput); // Converts input to integer without validation |

| **Compliant Code** |
| --- |
| Safely converts userInput to an integer if it’s valid. |
| int age;  if (std::istringstream(userInput) >> age) {  // Processes valid input only  } |

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

| **Principles(s):** Validate Input Data (1) ensures type safety by preventing invalid data formats, and Secure Coding Standards (10) mitigate vulnerabilities from unsafe type conversions. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | cpp:S1067 | Detects unsafe type conversions. |
| Clang-Tidy | 12.0 | cppcoreguidelines-narrowing-conversions | Highlights unsafe narrowing conversions. |
| CodeQL | 2.5 | cpp/type\_safety | Identifies risky type conversions. |
| Coverity | 2020.12 | UNINIT | Finds type safety issues related to uninitialized data. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Ensuring that values are within acceptable ranges prevents invalid data from causing crashes or errors in the application. |

| **Noncompliant Code** |
| --- |
| Index isn’t validated, risking out-of-bounds access. |
| int index = userInput;  array[index] = value; // No boundary check |

| **Compliant Code** |
| --- |
| Ensures index is within valid bounds, preventing overflow. |
| if (index >= 0 && index < arraySize) {  array[index] = value; // Checked bounds  } |

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

| **Principles(s):** Validate Input Data (1) enforces value ranges, while Defense in Depth (8) ensures multiple layers protect against crashes. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Medium | Medium | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2020.12 | ARRAY\_BOUNDS | Detects array bound issues. |
| Clang Static Analyzer | 12.0 | core.DivideZero | Identifies potential divide-by-zero errors. |
| CodeQL | 2.5 | cpp/array\_bounds | Detects array out-of-bounds issues. |
| PVS-Studio | 7.10 | V102 | Checks for boundary violations in data access. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Proper handling of strings ensures that buffer overflows are avoided and data integrity is maintained. |

| **Noncompliant Code** |
| --- |
| Fails to limit copied input size, risking overflow. |
| char buffer[10];  strcpy(buffer, userInput); // No check on input length |

| **Compliant Code** |
| --- |
| Copies input within buffer size, preventing overflow. |
| strncpy(buffer, userInput, sizeof(buffer) - 1);  buffer[sizeof(buffer) - 1] = '\0'; // Ensures no overflow |

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

| **Principles(s):** Validate Input Data (1), Simplicity (4), and Secure Coding Standards (10) prevent buffer overflows by ensuring safe string handling. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Low | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0 | cstring.OutOfBounds | Detects potential buffer overflows. |
| Fortify Static Code Analyzer | 21.1 | CWE-120 | Identifies buffer overflow issues. |
| CodeQL | 2.5 | cpp/strcpy\_overflow | Highlights unsafe string copy operations. |
| PVS-Studio | 7.10 | V105 | Identifies unsafe string handling practices. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | This standard helps protect applications from SQL injection attacks by ensuring user input is properly sanitized and queries are safely executed. |

| **Noncompliant Code** |
| --- |
| Direct concatenation of user input into SQL query risks injection. |
| std::string query = "SELECT \* FROM users WHERE name = '" + userInput + "'";  executeQuery(query); // Vulnerable to SQL injection |

| **Compliant Code** |
| --- |
| Parameterized queries protect against SQL injection. |
| std::string query = "SELECT \* FROM users WHERE name = ?";  executeQuery(query, userInput); // Uses parameterized query |

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

| **Principles(s):** Sanitize Data Sent to Other Systems (7) and Defense in Depth (8) protect against SQL injection vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | High | Critical | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Checkmarx | 8.8 | SQLInjection | Identifies SQL injection vulnerabilities. |
| Fortify Static Code Analyzer | 21.1 | CWE-89 | Identifies unsafe SQL query construction. |
| SonarQube | 8.9 | cpp:S3649 | Scans for SQL injection risks in C++. |
| CodeQL | 2.5 | cpp/sql\_injection | Detects injection vulnerabilities in SQL queries. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Protecting memory ensures that memory leaks and dangling pointers do not lead to crashes or vulnerabilities in the application. |

| **Noncompliant Code** |
| --- |
| Memory is allocated but never freed, causing a leak. |
| int\* ptr = new int[10];  // Memory leak: ptr not deleted |

| **Compliant Code** |
| --- |
| Deletes allocated memory, preventing leaks. |
| int\* ptr = new int[10];  delete[] ptr; // Memory properly released |

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

| **Principles(s):** Defense in Depth (8) and Quality Assurance (9) ensure memory safety by preventing leaks and crashes. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- | --- |
| Medium | | Medium | Low | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind (Memcheck) | 3.16 | Memcheck | Identifies memory leaks and misuses. |
| AddressSanitizer | GCC 9.3 | leak | Detects memory leaks and invalid accesses. |
| Clang Static Analyzer | 12.0 | core.DynamicMemory | Checks for dynamic memory misuse. |
| Coverity | 2020.12 | MISSING\_FREE | Finds instances of memory not freed. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Uses assertions to verify expected code conditions, reducing bugs in production. |

| **Noncompliant Code** |
| --- |
| Lacks a check for the value parameter, assuming it’s always positive. |
| void process(int value) {  // Assume value is always positive  } |

| **Compliant Code** |
| --- |
| Adds an assertion to enforce precondition that value is positive. |
| void process(int value) {  assert(value > 0); // Ensures value is positive  } |

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

| **Principles(s):** Defense in Depth (8) and Secure Coding Standards (10) enforce runtime checks to prevent errors in production. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Low | Low | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.10 | V100 | Checks for missing assertions and validates conditions. |
| Clang Static Analyzer | 12.0 | core.NullDereference | Detects conditions where assertions could prevent null dereference. |
| CodeQL | 2.5 | cpp/asserts | Analyzes conditions for missing assertions. |
| SonarQube | 8.9 | cpp:S905 | Flags potential areas where assertions are required. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Catches and manages exceptions to prevent program crashes. |

| **Noncompliant Code** |
| --- |
| Fails to check for division by zero, which would throw an exception. |
| int process(int value) {  return 100 / value; // Risk of division by zero  } |

| **Compliant Code** |
| --- |
| Checks for zero and throws an exception if encountered. |
| int process(int value) {  if (value != 0) {  return 100 / value;  }  throw std::invalid\_argument("Division by zero");  } |

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

| **Principles(s):** Simplicity (4) and Secure Coding Standards (10) ensure safe exception handling, avoiding unexpected crashes. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Low | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | cpp:ExceptionHandling | Detects incorrect exception handling. |
| CodeQL | 2.5 | cpp/exception\_safety | Identifies exception safety issues. |
| PVS-Studio | 7.10 | V601 | Checks for unhandled exceptions. |
| Clang Static Analyzer | 12.0 | core.CallAndMessage | Highlights exception risks in function calls. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Pointer Safety | STD-008-CPP | Proper handling of pointers is essential to prevent issues like null pointer dereferencing and dangling pointers, which can lead to crashes or undefined behavior. |

| **Noncompliant Code** |
| --- |
| This code attempts to dereference a null pointer, which leads to undefined behavior and potential crashes. |
| int\* ptr = nullptr;  \*ptr = 10; // Dereferencing a null pointer |

| **Compliant Code** |
| --- |
| The code ensures that ptr is assigned a valid address before dereferencing, preventing null pointer issues. |
| int value = 10;  int\* ptr = &value;  \*ptr = 20; // Safe: ptr points to a valid object |

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

| **Principles(s):** Least Privilege (6) and Defense in Depth (8) mitigate pointer-related vulnerabilities like null dereferences. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 12.0 | core.NullDereference | Detects attempts to dereference null pointers. |
| CodeQL | 2.5 | cpp/pointer\_safety | Highlights issues related to unsafe pointer handling. |
| PVS-Studio | 7.10 | V601 | Checks for null pointer dereferencing. |
| AddressSanitizer | GCC 9.3 | use-after-free | Detects pointers used after memory has been released. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File I/O Safety | STD-009-CPP | Safe file I/O operations prevent unauthorized access and ensure correct handling of files, avoiding resource leaks or unintentional overwrites. |

| **Noncompliant Code** |
| --- |
| The code doesn’t check if the file opened successfully, which could lead to undefined behavior if the file doesn’t exist or is inaccessible. |
| std::ifstream file("data.txt"); |

| **Compliant Code** |
| --- |
| This example checks whether the file was successfully opened before proceeding, reducing the risk of undefined behavior. |
| std::ifstream file("data.txt");  if (!file.is\_open()) {  throw std::runtime\_error("File could not be opened");  } |

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

| **Principles(s):** Default Deny (5), Sanitize Data Sent to Other Systems (7), and Defense in Depth (8) secure file operations against unauthorized access. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Medium | Low | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeQL | 2.5 | cpp/file\_access | Analyzes file access to ensure safety and proper error handling. |
| Coverity | 2020.12 | RESOURCE\_LEAK | Detects unclosed file handles and potential leaks. |
| SonarQube | 8.9 | cpp:S995 | Identifies resources that may be improperly managed in file handling. |
| Fortify Static Code Analyzer | 21.1 | CWE-404 | Highlights potential issues in managing file resources. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Concurrency Control | STD-010-CPP | Concurrency control prevents race conditions and other issues in multithreaded environments, ensuring thread-safe access to shared resources. |

| **Noncompliant Code** |
| --- |
| Without synchronization, concurrent access to sharedCounter could cause race conditions. |
| void update() {  sharedCounter++;  } |

| **Compliant Code** |
| --- |
| Uses a mutex to synchronize access to sharedCounter, ensuring thread-safe operations. |
| void update() {  std::lock\_guard<std::mutex> lock(mutex);  sharedCounter++;  } |

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

| **Principles(s):** Least Privilege (6), Defense in Depth (8), and Secure Coding Standards (10) prevent race conditions and ensure thread safety. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helgrind (Valgrind Tool) | 3.16 | RaceDetector | Detects race conditions and data races in multithreaded code |
| ThreadSanitizer | GCC 9.3 | race | Identifies race conditions in concurrent code execution. |
| Coverity | 2020.12 | ATOMICITY | Ensures atomicity in operations to prevent race conditions. |
| Clang Thread Safety Analysis | 12.0 | Thread Safety | Analyzes thread safety issues and potential race conditions. |

### 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.

**Automation Explanation**

Automation will play a crucial role in enforcing and ensuring compliance with the standards defined in this policy. Green Pace already has a robust DevOps infrastructure, which can be enhanced by incorporating DevSecOps practices to automate security at every stage of the software development lifecycle.

**Integration into the DevSecOps Pipeline**

To enforce compliance, the following modifications will be made to the DevOps process:

* **Pre-Commit Hooks**

Implement static analysis tools like SonarQube, Clang-Tidy, and PVS-Studio to automatically check for coding standard violations and vulnerabilities before code is committed to the repository.

* **Continuous Integration (CI)**

Integrate security testing tools into the CI pipeline. Tools such as Coverity and CodeQL will automatically analyze code quality and detect potential risks during each build cycle.

* **Automated Testing**

Use automated test suites to validate coding standards and identify runtime issues. Memory protection tools like AddressSanitizer and Valgrind will be included to test for leaks and pointer safety.

* **Continuous Monitoring**

Implement runtime monitoring tools to ensure compliance during deployment. ThreadSanitizer and Helgrind can detect concurrency issues in multithreaded environments.

* **Incident Response Automation**

Set up alerting mechanisms for any deviations from security policies. Logs from automation tools will be analyzed in real-time to identify and respond to threats quickly.

**DevSecOps Diagram Context**

This approach integrates security checks into every phase of the DevSecOps pipeline:

* **Plan:** Security policies and coding standards are defined.
* **Develop:** Automation tools validate code before and during commits.
* **Build:** Security-focused CI checks are applied to ensure secure builds.
* **Test:** Automated tests validate compliance with security standards.
* **Release:** Secure deployment processes include runtime security checks.
* **Monitor:** Continuous monitoring ensures compliance in production.
* **Respond:** Alerts are triggered for non-compliance or detected threats.

### 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 | Medium | Low | High | 4 |
| STD-002-CPP | Medium | Medium | Medium | Medium | 3 |
| STD-003-CPP | High | High | Low | High | 5 |
| STD-004-CPP | High | High | High | Critical | 5 |
| STD-005-CPP | Medium | Medium | Low | High | 4 |
| STD-006-CPP | Medium | Low | Low | Medium | 3 |
| STD-007-CPP | High | Low | Medium | High | 4 |
| STD-008-CPP | High | Medium | Low | High | 5 |
| STD-009-CPP | Medium | Medium | Low | Medium | 3 |
| STD-010-CPP | High | Medium | Medium | High | 5 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

### 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 | Data stored on physical media, such as disk drives, must be encrypted using AES-256 to ensure confidentiality and protection against unauthorized access. |
| Encryption in flight | All data transmitted between systems or services must use secure protocols such as TLS 1.3 or higher to prevent interception during transmission. |
| Encryption in use | Sensitive data in memory or during processing must leverage secure environments like Intel SGX or equivalent technologies to safeguard runtime integrity. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Users must be authenticated using multi-factor authentication (e.g., password + biometric or hardware token) to verify identity. |
| Authorization | |  | | --- | |  |  |  | | --- | | Implement role-based access control (RBAC) to ensure users can access only the resources required for their role. | |
| Accounting | Log all user actions, such as logins, resource changes, database interactions, and data access, to maintain audit trails. |

**\***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

|  |  |  |
| --- | --- | --- |
| **Standard** | **Principles Mapped** | **Justification** |
| STD-001-CPP | Validate Input Data (1), Secure Coding Standards (10) | Prevents invalid data formats and unsafe type conversions. |
| STD-002-CPP | Validate Input Data (1), Defense in Depth (8) | Ensures values are safe and prevents errors at multiple layers. |
| STD-003-CPP | Validate Input Data (1), Simplicity (4), Secure Coding Standards (10) | Prevents string overflows and ensures safe, clear handling of inputs. |
| STD-004-CPP | Sanitize Data Sent to Other Systems (7), Defense in Depth (8) | Mitigates SQL injection risks by sanitizing inputs and applying layered protections. |
| STD-005-CPP | Defense in Depth (8), Quality Assurance (9) | Ensures memory safety through layered defenses and robust testing. |
| STD-006-CPP | Defense in Depth (8), Secure Coding Standards (10) | Prevents runtime errors by enforcing preconditions. |
| STD-007-CPP | Simplicity (4), Secure Coding Standards (10) | Simplifies exception handling while maintaining secure error management. |
| STD-008-CPP | Least Privilege (6), Defense in Depth (8) | Ensures pointer safety by applying least privilege principles and multi-layered checks. |
| STD-009-CPP | Default Deny (5), Sanitize Data Sent to Other Systems (7), Defense in Depth (8) | Protects file access through controlled, sanitized inputs and layered defenses. |
| STD-010-CPP | Least Privilege (6), Defense in Depth (8), Secure Coding Standards (10) | Ensures thread safety and prevents race conditions through secure coding and restricted access. |

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 | 12/02/2024 | Final Version | Samuel Rincon |  |
|  |  |  |  |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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