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



# Green Pace Secure Development Policy

(From Templet Provided SNHU - CS-405 - Alan Spencer, Adjunct Instructor (M.S)

# - Duane Wegner – August 6, 2025)

## 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 6](#_Toc52464060)

[Coding Standard 3 8](#_Toc52464061)

[Coding Standard 4 9](#_Toc52464062)

[Coding Standard 5 11](#_Toc52464063)

[Coding Standard 6 12](#_Toc52464064)

[Coding Standard 7 14](#_Toc52464065)

[Coding Standard 8 16](#_Toc52464066)

[Coding Standard 9 18](#_Toc52464067)

[Coding Standard 10 20](#_Toc52464068)

[Defense-in-Depth Illustration 22](#_Toc52464069)

[Project One 22](#_Toc52464070)

[1. Revise the C/C++ Standards 22](#_Toc52464071)

[2. Risk Assessment 22](#_Toc52464072)

[3. Automated Detection 22](#_Toc52464073)

[4. Automation 22](#_Toc52464074)

[5. Summary of Risk Assessments 23](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 23](#_Toc52464076)

[7. Map the Principles 25](#_Toc52464077)

[Audit Controls and Management 26](#_Toc52464078)

[Enforcement 26](#_Toc52464079)

[Exceptions Process 26](#_Toc52464080)

[Distribution 27](#_Toc52464081)

[Policy Change Control 27](#_Toc52464082)

[Policy Version History 27](#_Toc52464083)

[Appendix A Lookups 27](#_Toc52464084)

[Approved C/C++ Language Acronyms 27](#_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 | Ensuring all input data is properly validated before use prevents malicious or malformed input from causing unexpected behavior or vulnerabilities. This principle guards against attacks such as buffer overflows, injection flaws, and data corruption by enforcing strict input checks and sanitization. |
| 1. Heed Compiler Warnings | Compiler warnings often reveal potential security and correctness issues. Developers should carefully address all warnings rather than ignore them, as doing so improves code quality and reduces hidden bugs or vulnerabilities. |
| 1. Architect and Design for Security Policies | Security must be considered early in the design and architecture phases. Building security policies into system architecture ensures consistent enforcement of access controls, data protection, and secure communication throughout the application lifecycle. |
| 1. Keep It Simple | Simple, clear, and understandable code reduces the chance of errors and security flaws. Overly complex code is harder to analyze and maintain, increasing the risk of vulnerabilities and defects. |
| 1. Default Deny | By default, all access and actions should be denied unless explicitly allowed. This principle minimizes exposure and restricts system behavior to known safe operations only. |
| 1. Adhere to the Principle of Least Privilege | Users and processes should be granted the minimum privileges necessary to perform their tasks. Limiting privileges reduces the impact of accidental or malicious misuse. |
| 1. Sanitize Data Sent to Other Systems | Any data exchanged with external systems should be sanitized and validated to prevent injection attacks or data corruption. Proper sanitation ensures system boundaries remain secure. |
| 1. Practice Defense in Depth | Multiple layers of security controls and mitigations should be employed so that if one control fails, others still protect the system. Defense in depth provides redundancy and resilience. |
| 1. Secure and Privacy-Conscious Logging | Avoid logging sensitive information such as passwords or personal data. Mask or exclude sensitive details in logs to prevent unauthorized disclosure and comply with privacy regulations. |
| 1. Consistent and Correct Error Code Handling | Always check and handle error codes properly. Ignoring errors can lead to silent failures, inconsistent states, and security vulnerabilities, while diligent error handling ensures predictable, reliable software behavior. |

### 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** | **Data Type Correctness and Safety** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Ensuring correct use of data types prevents unintended behavior, data corruption, and security vulnerabilities such as integer overflows and type confusion. This standard helps maintain type safety, enforce proper conversions, and avoid errors related to implicit casts or mismatched types. |

| **Noncompliant Code** |
| --- |
| This code uses implicit conversions between signed and unsigned integers, which can cause unexpected results or overflows. |
| int index = -1;  unsigned int uindex = index; // Implicit conversion from negative signed to unsigned  if (uindex > 10) {  // ...  } |

| **Compliant Code** |
| --- |
| This code uses explicit checks and consistent data types to avoid implicit conversions and potential errors. |
| int index = -1;  if (index < 0) {  // Handle error  } else {  unsigned int uindex = static\_cast<unsigned int>(index);  if (uindex > 10) {  // ...  }  } |

**Principles(s)**

| Validates input data to prevent errors (1); keeps code simple and clear (4); applies default deny (fail-safe) to avoid unsafe conversions (5); enforces secure and privacy-conscious logging by avoiding sensitive leaks (9). |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CPPCheck | 2.13 | Implicit conversions | Flags implicit and unsafe casts |
| Clang Static Analyzer | 17.0.0 | clang-analyzer-core,  clang-analyzer-core.UndefinedBinaryOperatorResult | Detects unsafe type conversions and misuse |
| SonarQube | 10.7 | cpp:S3242 | Detects potential integer overflow and type mismatches |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Data Value Validation and Range Checking** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Validating data values and enforcing valid ranges prevents logic errors, runtime faults, and security vulnerabilities such as buffer overflows or integer wraparounds. This standard requires developers to explicitly check values before use to ensure they fall within acceptable, expected ranges. |

| **Noncompliant Code** |
| --- |
| The following code does not check if the user input is within an acceptable range, which may lead to undefined behavior or security risks. |
| int array[10];  int index = getUserInput(); // No validation of input value  array[index] = 5; // Potential out-of-bounds access |

| **Compliant Code** |
| --- |
| This example validates the user input before using it to access the array, preventing out-of-bounds errors. |
| int array[10];  int index = getUserInput();  if (index >= 0 && index < 10) {  array[index] = 5; // Safe access after validation  } else {  // Handle invalid input  } |

**Principles(s)**

| Validates input data and enforces default deny to reject invalid input (1, 5); employs defense in depth by layering validation checks (8). |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.21.0 | Memcheck | Detects buffer overflow and invalid memory access |
| Clang-Tidy | 17.0.0 | clang-analyzer-core.ArrayBound | |  | | --- | | Finds potential buffer overflow vulnerabilities |  |  | | --- | |  | |
| SonarQube | 10.7 | Cpp:S2671 | Detects unchecked array or buffer access |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Proper Handling and Validation of Strings** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Incorrect handling of strings can lead to buffer overflows, memory corruption, and security vulnerabilities such as injection attacks. This standard enforces safe string manipulation practices, including proper length checks, use of safe functions, and avoiding dangerous functions like strcpy without bounds checks. |

| **Noncompliant Code** |
| --- |
| The code uses strcpy without verifying the destination buffer size, risking buffer overflow. |
| char buffer[10];  strcpy(buffer, "This string is too long for buffer"); // No bounds checking |

| **Compliant Code** |
| --- |
| The code uses strncpy with explicit length limits to prevent buffer overflow. |
| char buffer[10];  strncpy(buffer, "This string fits", sizeof(buffer) - 1);  buffer[sizeof(buffer) - 1] = '\0'; // Ensure null termination |

**Principles(s)**

| Ensures safe string handling by validating input (1); keeps code simple and understandable (4); consistent and correct error code handling to avoid buffer overflows and faults (10). |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify | 22.1 | |  | | --- | | Dangerous function use detection |  |  | | --- | |  | | Identifies calls to unsafe functions like strcpy, gets, sprintf |
| CPPCheck | 2.13 | Misc suspicious function calls | Flags usage of functions known for vulnerabilities |
| SonarQube | 10.7 | cpp:S4996 | Detects deprecated or unsafe function calls |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Prevent SQL Injection Vulnerabilities** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | SQL injection occurs when untrusted input is improperly included in SQL statements, allowing attackers to execute malicious commands. This standard enforces the use of parameterized queries and input sanitization to prevent injection attacks and protect database integrity. |

| **Noncompliant Code** |
| --- |
| This code concatenates user input directly into an SQL query string, exposing the application to SQL injection. |
| std::string userInput = getUserInput();  std::string query = "SELECT \* FROM users WHERE name = '" + userInput + "';";  executeSQL(query); |

| **Compliant Code** |
| --- |
| This example uses parameterized queries to safely incorporate user input into SQL statements, preventing injection. |
| std::string userInput = getUserInput();  SQLStatement stmt = prepareSQL("SELECT \* FROM users WHERE name = ?");  stmt.bind(1, userInput);  stmt.execute(); |

**Principles(s)**

| Validates input to prevent SQL injection (1); uses default deny to reject unsafe inputs (5); sanitizes data before sending to database to avoid injection (7); practices defense in depth with multiple sanitization layers (8). |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Veracode | 3.21.0 | Input validation rules | Detects missing or improper validation |
| Clang-Tidy | 17.0.0 | clang-analyzer-security | Flags unsafe input handling |
| SonarQube | 10.7 | cpp:S2099 | Detects insufficient input validation |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Safe Memory Management and Protection** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Proper memory management prevents vulnerabilities such as buffer overflows, memory leaks, dangling pointers, and use-after-free errors. This standard enforces careful allocation, deallocation, and access to memory to maintain application stability and security. |

| **Noncompliant Code** |
| --- |
| The code allocates memory but does not free it, causing a memory leak and potential resource exhaustion. |
| char\* buffer = new char[100];  // ... use buffer  // missing delete[] buffer; |

| **Compliant Code** |
| --- |
| The code correctly deallocates the allocated memory to avoid leaks. |
| char\* buffer = new char[100];  // ... use buffer  delete[] buffer; |

**Principles(s)**

| Validates memory-related inputs to prevent overflow (1); keeps memory management simple (4); applies default deny (fail-safe) to handle errors safely (5); employs defense in depth with multiple memory safety layers (8); uses secure logging and QA tools to monitor memory issues (9). |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind Memcheck | 3.21.0 | Memory leak detection | Detects memory leaks and invalid accesses |
| CPPCheck | 2.13 | Memory leak and pointer misuse detection | Flags improper memory management |
| SonarQube | 10.7 | cpp:S5332 | Detects unsafe executions of system calls |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use Assertions to Detect Programming Errors** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions help catch programming errors during development by verifying assumptions about program state. This standard encourages the use of assertions to identify invalid states early, improving code reliability and security. |

| **Noncompliant Code** |
| --- |
| The code performs no checks and proceeds despite potentially invalid input, leading to undefined behavior. |
| int divide(int a, int b) {  return a / b; // No check for division by zero  } |

| **Compliant Code** |
| --- |
| The code uses an assertion to verify that the divisor is not zero before proceeding. |
| #include <cassert>  int divide(int a, int b) {  assert(b != 0 && "Division by zero!");  return a / b;  } |

**Principles(s)**

| Simplifies error detection with assertions (4); adds defense in depth by catching errors early at runtime (8); improves quality assurance with tools to catch bugs (9). |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17.0.0 | bugprone-use-after-move | Checks improper use of assertions |
| CPPCheck | 2.13 | Assert check | Detects missing or misused assertions |
| SonarQube | 10.7 | cpp:S2589 | Detects absence or misuse of assertion |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Proper Handling of Exceptions and Error Conditions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Proper exception handling ensures that programs respond gracefully to error conditions without crashing or leaking resources. This standard promotes consistent use of try-catch blocks and avoids catching overly broad exceptions that can hide bugs. |

| **Noncompliant Code** |
| --- |
| The code catches all exceptions using a catch-all handler, which can obscure the source of errors and prevent proper error handling. |
| try {  performOperation();  } catch (...) {  // Swallows all exceptions, no error info logged  } |

| **Compliant Code** |
| --- |
| The code catches specific exceptions and handles them appropriately, enabling better debugging and recovery. |
| try {  performOperation();  } catch (const std::runtime\_error& e) {  std::cerr << "Runtime error: " << e.what() << std::endl;  } catch (const std::exception& e) {  std::cerr << "Exception: " << e.what() << std::endl;  } |

**Principles(s)**

| Uses defense in depth for robust exception handling (8); enforces secure logging and quality assurance for exceptions (9); consistent error code handling ensures stability (10). |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify | 22.1 | Exception handling | Detects improper or missing exceptions |
| CPPCheck | 2.13 | Exception specifications | Checks compliance with exception specifications |
| SonarQube | 10.7 | cpp:S1166 | Detects catch blocks that do not handle exceptions properly |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Safe Use of Concurrency and Synchronization Primitives** |
| --- | --- | --- |
| Concurrency safety | [STD-008-CPP] | Improper handling of concurrency can cause race conditions, deadlocks, and inconsistent state. This standard promotes the correct use of mutexes, locks, and atomic operations to ensure thread-safe code. |

| **Noncompliant Code** |
| --- |
| This code accesses a shared variable without synchronization, risking race conditions. |
| int counter = 0;  void increment() {  counter++; // No mutex protection  } |

| **Compliant Code** |
| --- |
| The code uses a mutex to safely synchronize access to the shared variable. |
| #include <mutex>  std::mutex mtx;  int counter = 0;  void increment() {  std::lock\_guard<std::mutex> lock(mtx);  counter++;  } |

**Principles(s)**

| Keeps concurrent code simple and maintainable (4); applies quality assurance to detect concurrency issues (9); consistent error handling prevents concurrency bugs (10). |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17.0.0 | readability-const-return-type | Suggests use of const for return types |
| CPPCheck | 2.13 | Const correctness | Detects missing const qualifiers |
| SonarQube | 10.7 | cpp:S1751 | Detects mutable variables that could be const |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Secure and Privacy-Conscious Logging** |
| --- | --- | --- |
| Data Privacy & Security | [STD-009-CPP] | Logging is crucial for monitoring and troubleshooting, but improper logging can leak sensitive information such as passwords, personal data, or cryptographic keys. This standard requires developers to carefully evaluate what data is logged to avoid exposing confidential information or violating privacy regulations. Logs should avoid including raw sensitive data and instead use masking or anonymization techniques. Also, logs should be protected to prevent unauthorized access. |

| **Noncompliant Code** |
| --- |
| This code logs sensitive user information, specifically the user’s password, in plain text. Logging sensitive data like passwords poses a significant security risk because logs can be accessed by unauthorized users or leaked, potentially exposing confidential credentials and violating privacy regulations. |
| std::string password = getPassword();  log("User password: " + password); // Logs raw sensitive data — a major privacy risk |

| **Compliant Code** |
| --- |
| This code avoids logging sensitive data directly by replacing the password with a placeholder such as [REDACTED]. Masking or omitting sensitive information in logs helps protect user privacy and minimizes the risk of exposing confidential data through log files. |
| std::string password = getPassword();  log("User password: [REDACTED]"); // Masks sensitive data to prevent exposure in logs |

**Principles(s)**

| Heeds compiler warnings to detect sensitive data leaks (2); applies least privilege by limiting sensitive log data exposure (6); sanitizes data before logging to protect privacy (7); uses secure logging practices and tools to protect privacy (9). |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify | 22.1 | Log forging and sensitive data logging | Detects logging of sensitive4 information in plain text |
| SonarQube | 10.7 | cpp:S4721 | Identifies logs containing sensitive data without masking |
| CPPCheck | 2.13 | Security logging sensitive data | Flags any logging statements with potentially sensitive variables |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Consistent and Correct Error Code Handling** |
| --- | --- | --- |
| Robustness and Reliability | [STD-010-CPP] | Ignoring return codes or inconsistent error handling leads to silent failures, unstable application states, and potential security risks such as resource leaks or corrupted data. This standard mandates explicit checking and proper handling of all error codes returned by functions, ensuring that error conditions trigger appropriate corrective actions or safe failures. |

| **Noncompliant Code** |
| --- |
| This code calls a function that returns an error code but completely ignores the result. Ignoring error codes can lead to silent failures, where problems go unnoticed and unhandled, potentially causing unstable system states, resource leaks, or security vulnerabilities. |
| int status = performAction();  // No check on status; errors may go unnoticed causing undefined behavior or security issues |

| **Compliant Code** |
| --- |
| This code explicitly checks the return status of the function and handles errors appropriately by logging the failure and executing a recovery or error-handling routine. Proper error code handling ensures that faults are detected early and the system can respond gracefully, improving stability and security. |
| int status = performAction();  if (status != 0) {  // Log the error and perform recovery or safe exit  log("performAction failed with status: " + std::to\_string(status));  handleError(status);  } |

**Principles(s)**

| Designs error handling into architecture (3); applies default deny to ensure fail-safe error responses (5); consistent and correct error code handling for reliability (10). |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.7 | cpp:S3677 | Flags ignored return values for critical operations |
| Fortify | 22.1 | Unchecked return values | Detects functions where return codes are ignored |
| CPPCheck | 2.13 | Unused returned values | Warns when a return value from a function is not 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.

Automation should be implemented at multiple stages. First, it must be integrated during pre-production verification and testing using static and dynamic analysis tools. Second, automation should run whenever the codebase requires analysis, such as during transitions or health checks. Additional automation can be applied at other stages as needed.

### 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 | 5 |
| STD-002-CPP | Critical | Likely | Medium | High | 5 |
| STD-003-CPP | High | Unlikely | Low | High | 4 |
| STD-004-CPP | Critical | Likely | Medium | High | 5 |
| STD-005-CPP | High | Unlikely | Medium | High | 4 |
| STD-006-CPP | Medium | Likely | Low | Medium | 3 |
| STD-007-CPP | High | Unlikely | Medium | High | 4 |
| STD-008-CPP | Medium | Unlikely | Low | Medium | 3 |
| STD-009-CPP | High | Likely | Low | High | 5 |
| STD-010-CPP | High | Likely | Low | Critical | 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 | Encryption at rest protects stored data on disk or in databases by using cryptographic techniques such as AES-256. This policy applies whenever sensitive data such as user credentials, personal data, or intellectual property is stored, ensuring that data remains secure if physical media or storage is compromised. |
| Encryption in flight | Encryption in flight secures data transmitted between systems or components, typically via protocols like TLS 1.3. This policy applies to all network communications containing sensitive or confidential information to prevent interception, eavesdropping, or tampering. |
| Encryption in use | Encryption in use protects data while it is being processed in memory, often through hardware-based solutions or secure enclaves. This policy applies to sensitive computations or operations where data must remain confidential even during active processing, reducing risk from memory scraping or side-channel attacks. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identity of users or systems attempting access, typically through passwords, multifactor authentication, or certificates. This policy applies to all login attempts and system interactions requiring identity verification to prevent unauthorized access. |
| Authorization | Authorization controls user and process permissions, granting only the minimum necessary privileges to perform tasks. This policy applies to all access controls, user roles, and resource permissions to enforce the principle of least privilege. |
| Accounting | Accounting maintains detailed logs of user actions such as logins, database changes, file access, and user management events. This policy ensures that all activities are auditable for compliance, forensic analysis, and anomaly detection. |

**\***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

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Principles Mapped** | **Justification** |
| STD-001-CPP | 1, 4, 5, 9 | Validates input data to prevent errors (1); keeps code simple and clear (4); applies default deny (fail-safe) to avoid unsafe conversions (5); enforces secure and privacy-conscious logging by avoiding sensitive leaks (9). |
| STD-002-CPP | 1, 5, 8 | Validates input data and enforces default deny to reject invalid input (1, 5); employs defense in depth by layering validation checks (8). |
| STD-003-CPP | 1, 4, 10 | Ensures safe string handling by validating input (1); keeps code simple and understandable (4); consistent and correct error code handling to avoid buffer overflows and faults (10). |
| STD-004-CPP | 1, 5, 7, 8 | Validates input to prevent SQL injection (1); uses default deny to reject unsafe inputs (5); sanitizes data before sending to database to avoid injection (7); practices defense in depth with multiple sanitization layers (8). |
| STD-005-CPP | 1, 4, 5, 8, 9 | Validates memory-related inputs to prevent overflow (1); keeps memory management simple (4); applies default deny (fail-safe) to handle errors safely (5); employs defense in depth with multiple memory safety layers (8); uses secure logging and QA tools to monitor memory issues (9). |
| STD-006-CPP | 4, 8, 9 | Simplifies error detection with assertions (4); adds defense in depth by catching errors early at runtime (8); improves quality assurance with tools to catch bugs (9). |
| STD-007-CPP | 8, 9, 10 | Uses defense in depth for robust exception handling (8); enforces secure logging and quality assurance for exceptions (9); consistent error code handling ensures stability (10). |
| STD-008-CPP | 4, 9, 10 | Keeps concurrent code simple and maintainable (4); applies quality assurance to detect concurrency issues (9); consistent error handling prevents concurrency bugs (10). |
| STD-009-CPP | 2, 6, 7, 9 | Heeds compiler warnings to detect sensitive data leaks (2); applies least privilege by limiting sensitive log data exposure (6); sanitizes data before logging to protect privacy (7); uses secure logging practices and tools to protect privacy (9). |
| STD-010-CPP | 3, 5, 10 | Designs error handling into architecture (3); applies default deny to ensure fail-safe error responses (5); consistent and correct error code handling for reliability (10). |
| \* This charts has been provided for ease of viewing mapped principles | | |

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 |  |
| 1.0.1 | 07/15/2025 | Initial edit of templet to provide 10 coding standards | Duane Wegner | [Insert text.] |
| 1.0.2 | 08/06/2025 | Filled in remaining information for coding standards, corrected policies 9 & 10, filled in templet information for all remaining sections after Defense-in-Depth Illustration | Duane Wegner | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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