**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 | Always validate input from users or externa sources before processing it. Unvalidated input can introduce security vulnerabilities like buffer overflows, SQL injection, or logic errors. Validating input ensures that it meets expected formats, ranges, and types. |
| 1. Heed Compiler Warnings | Compiler warnings often point to potential issues like uninitialized variables, types mismatches, or memory misuse. Treating these warnings seriously and resolving them proactively helps eliminate many bugs and security flaws early in development. |
| 1. Architect and Design for Security Policies | Security should be integrated at the architecture and design stages, not as an afterthought. This involves choosing secure technologies, designing secure communication channels, and ensuring that user roles, permissions, and access control are well-defined. |
| 1. Keep It Simple | Simple code is easier to review, test, and secure. Complex solutions often hide subtle bugs or vulnerabilities. Following the KISS (Keep It Simple, Stupid) principle reduces the risk of errors and increases code maintainability and security. |
| 1. Default Deny | Systems should be designed to deny access by default, only granting permissions explicitly when needed. This approach limits the attack surface and ensures that no unnecessary access is available to attackers or unauthorized users. |
| 1. Adhere to the Principle of Least Privilege | Each part of a system should operate with the minimum privileges necessary to perform its function. This minimizes damage if a component is compromised and prevents unauthorized access to sensitive data or operations. |
| 1. Sanitize Data Sent to Other Systems | Before sending data to databases, APIs, or other external systems, ensure it is sanitized to remove or escape harmful content. This prevents injection attacks and ensures that communication between components remains safe. |
| 1. Practice Defense in Depth | Uses multiple layers of security controls to protect against threats. If one layer fails, others still offer protection. This approach reduces reliance on any single defense mechanism and improves overall resilience. |
| 1. Use Effective Quality Assurance Techniques | Incorporating code reviews, unit testing, static analysis, and fuzz testing helps identify and fix issues early. These QA practices are crucial for catching vulnerabilities before they reach production. |
| 1. Adopt a Secure Coding Standard | Following established secure coding standards, like the SEI CERT C++ standard, ensures consistency and helps developers avoid common pitfalls. These guidelines cover everything from input handling to memory management and exception safety. |

### 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] | Using inappropriate data types or allowing implicit type conversions can lead to logic errors, buffer overflows, and vulnerabilities – especially when values exceed type limits or wrap unexpectedly. Ensuring data types are properly declared and validated protects against these issues. |

| **Noncompliant Code** |
| --- |
| This code implicitly converts a large unsigned value into a signed int, resulting in unpredictable behavior. |
| unsigned int balance = 4294967295; // max unsigned int  int displayBalance = balance; // implicit conversion causes overflow  std::cout << “Balance: “ << displayBalance << std::endl; |

| **Compliant Code** |
| --- |
| This version uses explicit casting and value checking to ensure the conversion is safe. |
| unsigned int balance = 4294967295;  if (balance <= static\_cast<unsigned int>(std::numeric\_limits<int>::max())) { |
| int displayBalance = static\_cast<int>(balance);  std::cout << “Balance: “ << displayBalance << std::endl; |
| } else { |
| std::cerr << “Error: Balance too large to safely display.” << std::endl; |
| } |

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

| **Principles(s):**  Validate Input Data – Ensuring that data types are appropriate helps verify input correctness and precent logic and memory errors.  Keep It Simple – Using the correct type from the beginning reduces the need for defensive conversions or complex logic to correct issues later.  Adopt a Secure Coding Standard – This standard directly supports consistent, secure handling of data values. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | Latest | bugprone-narrowing-conversions | Detects unsafe narrowing conversions |
| SonarQube | Latest | cxx:S3038 | Flags unsafe implicit conversions |
| CppCheck | Latest | type conversion | Detects type truncation and overflow risks |
| Fortify | Latest | DataTypeMismatch | Identifies unsafe use of incompatible data types |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Allowing unexpected or unchecked data values can result in logic errors, crashes, or security vulnerabilities. Restricting values ensures that the system behaves safely even in edge cases or malicious input scenarios. |

| **Noncompliant Code** |
| --- |
| This code accepts a user-provided age but does not check for valid numeric range or negative input. |
| int age;  std::cin >> age;  std::cout << “You are “ << age << “ years old.” << std::endl; |

| **Compliant Code** |
| --- |
| This version checks if the entered value falls within a valid human age range (0–130). |
| int age;  std::cin >> age;  if (age < 0 || age > 130) { |
| std::cerr << “Error: Invalid age entered.” << std::endl |
| } else { |
| std::cout << “You are “ << age << “ years old.” <<std::endl; |
| } |

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

| **Principles(s):**  Validate Input Data – Validating the range of input data ensures it behaves as expected and avoids dangerous edge cases. Default Deny – Invalid values are explicitly rejected instead of being silently accepted. Sanitize Data Sent to Other Systems – Prevents unsafe or nonsensical data from being used in calculations or output. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | Latest | misc-inefficient-algorithm | Can detect poor logic, including unchecked input |
| SonarQube | Latest | cxx:S2201 | Checks for value misuse or lack of boundary validation |
| Coverity | Latest | CHECKED\_RETURN | Ensures data is validated before use |
| Fortify | Latest | RangeCheck | Detects out-of-range input vulnerabilities |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Improper string handling can lead to buffer overflows, memory corruption, and security vulnerabilities. Using safe and bounded string operations helps avoid these issues and improves both security and stability. |

| **Noncompliant Code** |
| --- |
| This code uses the unsafe strcpy() function, which does not check buffer size and may overflow the destination buffer. |
| char username[10];  strcpy(username, "ThisIsWayTooLong");  std::cout << "Username: " << username << std::endl; |

| **Compliant Code** |
| --- |
| This version uses strncpy() to limit the number of copied characters and ensure the destination buffer is not overrun. |
| char username[10];  strncpy(username, "ThisIsWayTooLong", sizeof(username) - 1);  username[sizeof(username) - 1] = '\0'; // Ensure null-termination  std::cout << "Username: " << username << std::endl; |

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

| **Principles(s):**  Validate Input Data – Prevents unsafe input lengths from corrupting memory. Keep It Simple – Safer string functions reduce the chance of subtle memory issues. Sanitize Data Sent to Other Systems – Ensures string content is correctly formed before being passed to downstream components or displayed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | Latest | cert-str31-c | Flags use of unsafe C string functions |
| SonarQube | Latest | cxx:S2755 | Detects unsafe or deprecated string handling |
| Coverity | Latest | |  | | --- | | BUFFER\_OVERFLOW |  |  | | --- | |  | | |  | | --- | |  |   Identifies potential overflows in string operations |
| Fortify | Latest | BufferOverflow.String | Analyzes unbounded string copy risks |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | SQL injection occurs when untrusted user input is included in SQL statements without validation or escaping, allowing attackers to modify or hijack queries. Using parameterized statements eliminates this risk by separating data from code. |

| **Noncompliant Code** |
| --- |
| This code builds a raw SQL query using untrusted user input, making it vulnerable to injection. |
| std::string username = getUserInput();  std::string query = "SELECT \* FROM users WHERE name = '" + username + "';";  sqlite3\_exec(db, query.c\_str(), callback, nullptr, nullptr); |

| **Compliant Code** |
| --- |
| This version uses parameterized queries with sqlite3\_prepare\_v2 and sqlite3\_bind\_text to prevent SQL injection. |
| std::string username = getUserInput();  sqlite3\_stmt\* stmt;  sqlite3\_prepare\_v2(db, "SELECT \* FROM users WHERE name = ?;", -1, &stmt, nullptr);  sqlite3\_bind\_text(stmt, 1, username.c\_str(), -1, SQLITE\_TRANSIENT);  while (sqlite3\_step(stmt) == SQLITE\_ROW) { |
| // process result |
| }  sqlite3\_finalize(stmt); |

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

| **Principles(s):**  Validate Input Data – User input is treated as data, not executable code. Sanitize Data Sent to Other Systems – Prevents user input from injecting malicious SQL. Adopt a Secure Coding Standard – Parameterized queries are a core best practice for SQL safety. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | Latest | cxx:S3649 | Detects unsafe SQL string concat |
| Fortify | Latest | SQLInjection | Flags unsafe dynamic query usage |
| Coverity | Latest | SQL\_INJECTION\_TAINTED | Traces tainted input into SQL |
| CodeQL | Latest | cpp/sql-injection | Identifies non-parameterized queries |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Improper memory access, including buffer overflows or accessing deallocated memory, can lead to crashes, security vulnerabilities, or exploitation. Using bounds-checked operations and proper allocation/deallocation practices reduces risk and enhances stability. |

| **Noncompliant Code** |
| --- |
| This code writes outside the bounds of the buffer, causing a buffer overflow. |
| char buffer[10];  strcpy(buffer, "This string is too long for the buffer"); |

| **Compliant Code** |
| --- |
| This version uses std::string instead of raw buffers, preventing overflow and handling memory safely. |
| std::string buffer = "This string is too long for the buffer";  std::cout << buffer << std::endl; |
| OR |
| std::array<char, 10> buffer;  strncpy(buffer.data(), "SafeStr", buffer.size() - 1);  buffer[buffer.size() - 1] = '\0'; |

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

| **Principles(s):**  Validate Input Data – Prevents unsafe memory operations based on unchecked input. Keep It Simple – Using STL containers reduces manual error-prone memory handling. Practice Defense in Depth – Bounds checks and safe containers help ensure one coding mistake doesn’t lead to full exploitation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | Latest | Memcheck | Detects invalid memory use |
| Clang-Tidy | Latest | cert-str32-c | Flags buffer overflows |
| Coverity | Latest | BUFFER\_OVERFLOW | Identifies writes beyond array bounds |
| Fortify | Latest | MemoryCorruption | Detects misuse of allocated memory |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions are useful for catching programming errors during development by verifying conditions that should always be true. When used appropriately, they help identify bugs early, improving code reliability and maintainability. However, they must not replace runtime checks in production code. |

| **Noncompliant Code** |
| --- |
| This code assumes a valid divisor but does not check for zero, leading to undefined behavior. |
| int divide(int x, int y) { |
| return x / y; // No check for division by zero |
| } |

| **Compliant Code** |
| --- |
| This version uses an assertion to ensure that y is not zero before performing division. |
| #include <cassert> |
| int divide(int x, int y) { |
| assert(y != 0); // Only active in debug builds  return x / y; |
| } |

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

| **Principles(s):**  Validate Input Data – Assertions help verify that critical conditions are met before logic executes. Use Effective Quality Assurance Techniques – Assertions are key in development for debugging and sanity checking internal assumptions. Keep It Simple – They provide a lightweight way to enforce invariants during development. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | Latest | cert-err33-c | Detects misuse of assert |
| Coverity | Latest | ASSERT\_SIDE\_EFFECT | Flags improper usage of assert |
| SonarQube | Latest | cxx:S730 | Ensures assertions are used correctly |
| Fortify | Latest | AssertionCheck | Analyzes assertion coverage |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Exceptions are essential for handling unexpected conditions, but misuse can lead to program crashes, memory leaks, or logic flaws. Proper handling—such as catching by reference and avoiding empty catch blocks—helps maintain stability and provides meaningful recovery paths. |

| **Noncompliant Code** |
| --- |
| This code catches exceptions by value, which can lead to object slicing and information loss. |
| try { |
| throw std::runtime\_error("Something went wrong"); |
| } catch (std::exception e) { // BAD: caught by value |
| std::cerr << e.what() << std::endl; |
| } |

| **Compliant Code** |
| --- |
| This version catches exceptions by reference, preserving the dynamic type and avoiding slicing. |
| try { |
| throw std::runtime\_error("Something went wrong"); |
| } catch (const std::exception& e) { // GOOD: caught by reference |
| std::cerr << e.what() << std::endl; |
| } |

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

| **Principles(s):**  Use Effective Quality Assurance Techniques – Exceptions help manage failure paths cleanly. Keep It Simple – Proper use of try/catch prevents undefined behavior and logic clutter. Architect and Design for Security Policies – Predictable exception handling is critical to system stability and error response. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | Latest | cert-err60-cpp | Detects exceptions not caught by reference |
| Coverity | Latest | EXCEPTION\_CLASS\_SLICED | Identifies sliced exception objects |
| SonarQube | Latest | cxx:S1149 | Ensures proper exception catching syntax |
| Fortify | Latest | ImproperExceptionHandling | Detects poor exception patterns |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Buffer Limits | [STD-008-CPP] | Accepting input without properly checking the buffer size can lead to buffer overflows, data corruption, or unintended control flow. Input should always be read with respect to its capacity and null-termination. |

| **Noncompliant Code** |
| --- |
| This code uses std::cin >> without bounds checking, which can overflow the buffer if input exceeds the limit. |
| char username[10];  std::cin >> username; // No length checking |

| **Compliant Code** |
| --- |
| This version uses std::cin.getline() with a size limit and a check for input failure. |
| char username[10];  std::cin.getline(username, sizeof(username));  if (std::cin.fail()) { |
| std::cerr << "Input too long. Try again.\n"; |
| } |

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

| **Principles(s):**  Validate Input Data – Ensures the program does not accept data that could cause memory errors. Sanitize Data Sent to Other Systems – Keeps invalid, oversized strings out of downstream systems. Default Deny – Rejects overly long input rather than accepting it and risking overflow. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | Latest | cert-str31-c | Flags unbounded string input |
| Coverity | Latest | BUFFER\_SIZE | Detects improper input lengths |
| Fortify | Latest | InputValidation | Highlights unchecked input |
| SonarQube | Latest | cxx:S5869 | Detects unsafe input handling |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Avoid Use of Dangerous C Functions | [STD-009-CPP] | Many C standard library functions do not perform bounds checking and are highly prone to buffer overflows or format string vulnerabilities. Replacing them with safer alternatives (e.g., fgets, strncpy, snprintf) improves security and reliability. |

| **Noncompliant Code** |
| --- |
| This code uses gets() which reads input without bounds, risking buffer overflow. |
| char name[20];  gets(name); // Dangerous: no bounds checking |

| **Compliant Code** |
| --- |
| This version uses fgets() with a buffer limit to avoid overflow. |
| char name[20];  fgets(name, sizeof(name), stdin); // Safe input with size checking |

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

| **Principles(s):**  Validate Input Data – Enforcing bounds prevents malformed or malicious inputs. Keep It Simple – Safe standard functions are easier to maintain and debug. Adopt a Secure Coding Standard – Directly supports SEI CERT’s recommendation to avoid dangerous APIs. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | Latest | cert-msc24-c | Detects use of unsafe C functions |
| Fortify | Latest | UnsafeFunctionUse | Flags use of gets, strcpy, sprintf |
| Coverity | Latest | STRING\_OVERFLOW | Identifies dangerous function use |
| SonarQube | Latest | cxx:S5863 | Detects use of deprecated unsafe APIs |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Avoid Use of Magic Numbers | [STD-010-CPP] | Using unnamed numeric literals (magic numbers) in code can make it difficult to understand and maintain. It also introduces risk if values need to be changed across multiple locations. Replacing them with clearly named constants improves readability, reduces errors, and supports secure, maintainable code. |

| **Noncompliant Code** |
| --- |
| This code uses hardcoded values directly in the logic. |
| if (userAge > 17) { |
| std::cout << "You are eligible to vote.\n"; |
| } |

| **Compliant Code** |
| --- |
| This version uses a named constant to represent the voting age, improving clarity and flexibility. |
| const int MIN\_VOTING\_AGE = 18;  if (userAge >= MIN\_VOTING\_AGE) { |
| std::cout << "You are eligible to vote.\n"; |
| } |

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

| **Principles(s):**  Keep It Simple – Improves code clarity and reduces the chance of mistakes. Adopt a Secure Coding Standard – Aligns with SEI CERT and other standards promoting maintainable, secure code. Use Effective Quality Assurance Techniques – Readable constants are easier to test and audit. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | Latest | readability-magic-numbers | Warns about magic number use |
| SonarQube | Latest | cxx:S109 | Flags unnamed numeric literals |
| Coverity | Latest | LITERAL\_CONSTANT | Checks for maintainability issues |
| Fortify | Latest | MagicLiteral | Scans for unannotated literals |

### 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.

At Green Pace, automation is a critical part of ensuring continuous compliance with security standards throughout the software development lifecycle. The DevSecOps diagram outlines the integration of security in each phase of the DevOps process, bridging the gap between development (DEV), Security (SEC), and Operations (OPS) teams. Automation will be implemented to enforce and monitor security policy compliance across both pre-production and production environments.

In the pre-production phase, automation begins at the “Design” and “Build” stages. Tools such as static analysis (e.g., Cppcheck, SonarQube) will be integrated into the code editor and CI pipeline to enforce secure coding practices early. Pre-commit hooks and automated linters will be configured to reject code that violates security standards such as input validation, memory safety, and SQL injection protections. Automated dependency checkers will scan open-source packages for vulnerabilities.

During the “Verify and Test” stage, automation tools will run functional, compliance, and security test with every build. Vulnerability scanning and unit testing will be triggered automatically via the CI pipeline using tools like Cppchec, Cland-Tidy, or Fortify. Additionally, integration with signed trusted sources ensures that builds rely only on validated artifacts.

As applications move to production, automation continues with “Transition and Health Check”, where infrastructure as code (IaC) templates will include security baselines. CI/CD pipelines will enforce security configurations using tools like Terraform compliance checks or Azure Policy rules.

In the “Monitor and Detect” phase, automated security information and event management (SIEM) solutions, anomaly detection, and real-time alerts will be configured. Logs from application behavior will feed into centralized logging systems (e.g., ELK Stack or Splunk) that are continuously analyzed.

Finally, during “Respond” and “Maintain and Stabilize”, automated rollback procedures, backups, and system hardening scripts will activate based on security triggers or policy violations. This ensures rapid containment and recovery, maintaining a stable and compliant production environment.

By embedding automation into every step of the DevSecOps loop – from planning and design to monitoring and incident response – Green Pace can ensure that enforcement of the secure coding standards outlined in this policy is consistent, scalable, and continuous.

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Likely | Low | Critical | 1 |
| STD-003-CPP | Medium | Likely | Medium | High | 2 |
| STD-004-CPP | Critical | Likely | High | Critical | 1 |
| STD-005-CPP | High | Possible | Medium | High | 2 |
| STD-006-CPP | Medium | Possible | Low | Medium | 3 |
| STD-007-CPP | High | Possible | Medium | High | 2 |
| STD-008-CPP | Critical | Likely | Medium | Critical | 1 |
| STD-009-CPP | High | Likely | Low | Critical | 1 |
| STD-010-CPP | Medium | Unlikely | Low | Medium | 3 |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest refers to protecting data that is stored on a disk or database from unauthorized access. This includes databases, file systems, backups, and cloud storage. At Green Pac, this policy applies to all sensitive files, including user profiles, financial data, and audit logs. Encryption algorithms such as AES-256 should be used to ensure that if data is accessed outside proper authorization, it remains unreadable. This protects against data breaches due to hardware theft or unauthorized internal access. |
| Encryption in flight | Encryption in flight refers to protecting data while it is being transmitted over a network (e.g., via HTTPS, TLS, or VPNs). This policy applies to all user inputs, API communications, and third-party integrations. All data transmissions must be encrypted to prevent interception by attackers (e.g., man-in-the-middle attacks). This ensures secure communication between clients, servers, and remote systems. |
| Encryption in use | Encryption in use involves protecting data while it is being processed in memory or used by applications. This includes keeping data encrypted within RAM or using trusted execution environments (TEEs). At Green Pace, this policy applies to processing customer PII and financial transactions. Implementing encryption in use minimizes the risk of attacks that target memory dumps or insider access during live processing. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying the identity of a user or system before granting access. At Green Pace, authentication policies require multi-factor authentication (MFA), strong password policies, and OAuth/OpenID integration for third-party services. This applies to user logins, system access, and developer tools. Ensuring only verified users gain access significantly reduces the attack surface. |
| Authorization | Authorization determines the level of access a user has after authentication. It ensures that users can only access the resources and perform the actions their role permits. At Green Pace, role-based access control (RBAC) will enforce authorization rules. This covers changes to the database, file access, and deployment privileges. It ensures least privilege is applied consistently. |
| Accounting | Accounting involved tracking user activity, including login attempts, file access, and configuration changes. At Green Pace, systems logs, audit trails, and access reports will be generated and stored securely for periodic review. Accounting applies to addition of new users, file edits, and suspicious activity detection, supporting security audits and compliance. |

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

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

### Map the Principles

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

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

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

|  |  |  |
| --- | --- | --- |
| **Principle-to-Standard Mapping** | | |
| **Standard** | **Mapped Principles (By Number)** | **Justification** |
| STD-001-CPP: Data Type | 1, 4 | (1) *Least Privilege* ensures data types are appropriately constrained to avoid overflow/underflow or unintended behavior. (4) *Fail Securely* supports using strict data types to prevent accidental data corruption or leaks. |
| STD-002-CPP: Data Value | 1, 3, 4 | (1*) Least Privilege* limits operations on variable values. (3) *Secure Defaults* ensures value limits are built in. (4) *Fail Securely* promotes bounds checking and safe handling of abnormal values. |
| STD-003-CPP: String Correctness | 2, 6 | (2) *Defense in Depth* is crucial because string errors are a major source of buffer overflows. (6) *Keep It Simple* enforces using safer string libraries or wrappers, reducing chances for error-prone manipulation. |
| STD-004-CPP: SQL Injection | 2, 4, 5 | (2*) Defense in Depth* applies to layered protections such as input-validation, parameterized queries, and sanitization. (4) *Fail Securely* ensures that SQL failures do not expose system data. (5) *Don’t Trust External Sources.* |
| STD-005-CPP: Memory Protection | 1, 2, 4 | (1) *Least Privilege* limits access to memory. (2*) Defense in Depth* supports using multiple memory safeguards. (4) *Fail Securely* ensures memory mismanagement leads to controlled failure, not system crash. |
| STD-006-CPP: Assertions | 4, 6, 7 | (4) *Fail Securely* helps ensure that failed assertions reveal no sensitive data. (6) i avoids complex assertion logic. (7) *Use Security Features* to trigger alerts or logs when assertions fail. |
| STD-007-CPP: Exceptions | 4, 7 | (4) *Fail Securely* prevents applications from crashing in ways that expose vulnerabilities. (7*) Use Security Features* includes logging or handling sensitive operations during exception management. |
| STD-008-CPP: Input Buffer Limits | 2, 3, 5 | (2*) Defense in Depth* applies by validating all inputs. (3) *Secure Defaults* enforce maximum limits by default. (5) *Don’t Trust External Sources* is critical for protecting against buffer overflows from unchecked inputs. |
| STD-009-CPP: Avoid Dangerous C Functions | 6, 7, 8 | (6*) Keep It Simple* by avoid deprecated, complex, or unsafe APIs. (7) *Use Security Features* includes modern safe alternatives (like strncpy). (8) *Minimize Attack Surface* by removing legacy, risky functions. |
| STD-010- CPP: Avoid Magic Numbers | 6, 9 | (6*) Keep It Simple* by improving readability and reducing errors. (9) *Promote Privacy* indirectly supports this by helping ensure logic tied to sensitive thresholds or states is clear and maintainable. |

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 |  |
| 1.5 | 05/25/2025 | Edited/added Principles and Coding Standards | Trinity Anderson | Aaron Demory |
| 2.0 | 06/15/2025 | Fully revised/updated security policy | Trinity Anderson | Aaron Demory |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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