**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 | Ensures input validation prevents security exposures such as buffer overflows, SQL injections, and cross-site scripting. By enforcing precise format rules and boundary checks, applications block malicious data manipulation. Proper validation decreases the chance of unauthorized access and safeguards data integrity. |
| 1. Heed Compiler Warnings | Compiler alerts that if ignored can lead to suppressed vulnerabilities and unsafe code patterns such as memory leaks, or undefined behavior that attacks could exploit. These act as early-warning signals that should be fixed promptly to ensure code reliability and close hidden security gaps. |
| 1. Architect and Design for Security Policies | A secure system begins from the ground up with a strong architectural foundation that aligns with clear security policies and is implemented into a system design that can prevent costly patches down the road. A proactive approach can ensure resilience against evolving cyber threats. |
| 1. Keep It Simple | Complexity creates security risks that can make it hard to identify and moderate weaknesses. A well-organized codebase is easier to identify, audit, and understand to help protect against accidental vulnerabilities. |
| 1. Default Deny | Default security posture should only deny access when explicitly granted. This principle ensures that unauthorized users and processes cannot access sensitive components and can reduce exposure to cyber-attacks. This restrictive stance largely reduces the number of potential entry points for potential attackers. |
| 1. Adhere to the Principle of Least Privilege | Users should be granted only the rights needed to perform tasks. Excessive privileges can create over-permissioned accounts that can create exploitation security risks. To reduce the chances of a breach, tighten permissions and restrict access to limit an attacker’s reach. |
| 1. Sanitize Data Sent to Other Systems | Unfiltered data can present vulnerabilities in interconnected systems through injection attacks. Any data sent to external services should be scrubbed of any dangerous characters or commands. This principle reinforces security across integrated platforms. |
| 1. Practice Defense in Depth | By implementing multiple layers of security including firewalls, encryption, access controls and continuous monitoring all help create overlapping protections. This ensures comprehensive threat mitigation within a multi-layered security approach. |
| 1. Use Effective Quality Assurance Techniques | Rigorous testing, peer views, automated static analysis improves code security by identifying weaknesses early during the development cycle. Continuous evaluation and practicing securing coding can prevent defects from making their way into late development stages. Catching flaws early stops bad code from reaching production, making the software more robust and reliable. |
| 1. Adopt a Secure Coding Standard | Following institute coding standards prevents common security flaws and provides consistency across development. Guidelines such as SEI CERT C++ Coding standard promote the best practices for memory management, input handling, and error processing. By adhering to these standards, the chances of common vulnerabilities reduce. |

### 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-CXX] | Ensures proper type safety and avoid implicit conversions |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| Double pi = 3.14  Int value = pi // implicit conversion; loss of precision |

| **Compliant Code** |
| --- |
| [Compliant description] |
| Double pi = 3.14  Int value = static\_cast<int>(pi); //explicit conversion |

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

| **Principles(s):** Validate Input data: Alows to directly align with ensuring that proper type safety and avoiding implication conversions by enforcing explicit type handling |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | Latest | Implicit Conversion Check | Detects lossy or unsafe implicit type conversions |
| CppCheck | Latest | typeConversion | Identifies narrowing and risky implicit casts |
| SonarQube | Latest | cpp:S5146 | Flags unsafe type conversions and recommends the use of ‘static\_cast’ |
| CodeSonar | Latest | MISRA Cast Rules | Detects violations of secure casting practices via CERT/MISRA guidelines |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **valid Input data ranges** | [STD-002-CXX] | This standard allows for all external or calculated values that fall within predefined safe ranges. By validating input, you can prevent issues such as divide-by-zero, overflow and improper program logic |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| int divide(int x) {  return 100 / x; //no check for x == 0 which causes a divide-by-zero error  } |

| **Compliant Code** |
| --- |
| [Compliant description] |
| int divide(int x) {  if (x == 0) {  throw std::invalid\_argument(“X cannot be zero”);  }  return 100 / x;  } |

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

| **Principles(s):** Enforce valid input ranges: ensure all numerical values used in computations will fall within expected limits that prevent divide-by-zero errors, overflows and logic errors.  Halt securely: When invalid input occurs, the system should half safely using exceptions rather then allowing incorrect computations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang static Analyzer | Latest | DivisionByZeroCheck | Detects the use of unchecked values within critical math operations |
| CppCheck | Latest | invalidArguementCheck | Will flag functions that are missing range validation for user input |
| SonarQube | Latest | Cpp:S3518 | Identifies input that is not validated before computation |
| CodeQL | Latest | Cxx/input validation | Completes deep analysis for missing validation in parameters |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CXX] | Improper string handling can lead to buffer overflows, memory corruption, and other security vulnerabilities. This standard enforces operations to prevent inadvertent behaviors. |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| Char buffer[10];  Strcpy(buffer, “This string is too long”); // risk of buffer overflow |

| **Compliant Code** |
| --- |
| [Compliant description] |
| Char buffer[10];  Strncpy(buffer, “Safe”, sizeof(buffer) – 1);  Buffer[sizeof(buffer) – 1] = ‘\0’; // ensures null termination |

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

| **Principles(s):** avoid buffer overflow: Ensures that all string operations respect allocated buffer sizes that prevent memory corruption and security vulnerabilities  Fail securely: If a string operation encounters an invalid size or doesn’t meet proper termination, the system must be able to handle it safely to avoid undefined behaviors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang static Analyzer | Latest | BufferOverflowCheck | Alerts unsafe string operations that could exceed the limits of the buffer |
| CppCheck | Latest | strcpyUsage | Flags unsafe usage of functions like ‘strcpy’, ‘gets’ and ‘sprintf’ |
| SonarQube | Latest | StringOverflowCheck | Detects possible buffer overflows and improper string manipulations |
| CodeQL | Latest | Cxx/string-safety | Performs deep analysis for unsafe string manipulations |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CXX] | SQL injections are attacks that occur when an application improperly handles user-supplied data that allows attackers to manipulate queries and access unauthorized data. This standard enforces practices to mitigate injection attack risks |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| Std::string userInput = “admin’; DROP TABLE users; --";  Std::string query = “SELECT \* FROM users WHERE username = ‘ " + userInput + “ ’ ”;  executeQuery(query); //executes unsafe SQL statement that is vulnerable to injection |

| **Compliant Code** |
| --- |
| [Compliant description] |
| Std::string userInput = “admin";  Std::string query = “SELECT \* FROM users WHERE username = ?”;  PreparedStatement stmt = db.perpareStatement(query);  Stmt.setString(1, userInput);  Stmt.executeQuery(); // prevents SQL injection by safely handling user data |

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

| **Principles(s):** Parameterized queries: Ensures that user inputs are safely handled by preventing direct concatenation into SQL statements.  Failed Securely: If an invalid query attempt happens, the system should reject the operation safely rather than executing questionable commands |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SQLMap | Latest | injectionDetection | Identifies SQL vulnerabilities through automated testing |
| SonarQube | Latest | Cpp:S3649 | Flags unsafe query concatenation and lack of prepared statements |
| Fortify Static Code Analyzer | Latest | SQLinjectionCheck | Detects improper handling of dynamic queries |
| CodeQL | Latest | Cxx/sql-injection | Scans for SQL injection dangers in database query execution |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CXX] | Ensures that proper handling of memory allocation and deallocation to prevent security vulnerabilities including buffer overflows, user-after-free errors, and memory leaks. |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| void allocateMemory() {  int\* ptr = new int[100];  } // no delete statement where memory leaks can occur  void unsafeAccess() {  Int\* ptr = new int[10];  Delete[] ptr;  Ptr[0] = 42; //user-after-free error |

| **Compliant Code** |
| --- |
| [Compliant description] |
| void allocateMemory() {  std::unique\_ptr<int[]> ptr(new int[100]);  } //smart pointer that ensures cleanup  void safeAccess() {  std::vector<int> buffer(10);  buffer[0] = 42;  }//users STL containers for safe memory management |

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

| **Principles(s):** Ensure Memory Management: Prevents leaks and undefined behavior by enforcing appropriate allocation and deallocation.  Fail securely: If memory is not allocated correctly, the system must handle the error safely to prevent crashes or corrupted data |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | Latest | MemoryLeakCheck | Identifies missing deallocation that can lead to memory leaks |
| CppCheck | Latest | UAFUsageDetection | Alerts use-after-free vulnerabilities |
| Valgrind | Latest | Memcheck | Detects memory errors including leaks, invalid accesses and buffer overruns |
| AddressSanitzer | Latest | ASanHeapCheck | Detects heap corruption, out-of-bounds accesses and user-after-free errors |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CXX] | Verify conditions during runtime and ensure program integrity and prevent incorrect states. This standard enforces structured declaration usage to support debugging and error handling. |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| void processData(int value) {  assert(value > 0)  std::cout << “Processing: “ << value << std::endl;  } //may crash in production due to no fallback mechanism |

| **Compliant Code** |
| --- |
| [Compliant description] |
| Void processData(int value) {  If (value <= 0) {  Throw std::invalid\_argument(“Value needs to be greater than zero”);  }  Std::cout << “Processing: “ >> value << std::endl;  } |

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

| **Principles(s):** Use assertions safely: Ensures that assertions serve debugging purposes that don’t break execution flow in production  Fail securely: Prevent unintended failures by properly handling assertion failures through exceptions or controlled exits |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang static analyzer | Latest | AssertUsageCheck | Identifies unsafe assertions that could lead to unexpected failures |
| CppCheck | Latest | missingValidation | Identifies assertions users without proper input validation |
| SonarQube | Latest | Cpp:S5526 | Detects against misplaced assertions in production code |
| CodeQL | Latest | Cxx/assert-safety | Scans for improper assertion handling and lack of fallback logic |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CXX] | Improper exception handling can lead to unpredictable behavior, leaks, and security vulnerabilities. This standard is set to enforce structured exception management for robust error recovery and safe execution flow |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| Void processData(int value) {  Int result = 100 / value;  Std::cout << “Result: “ << result << std::endl;  } // no error handling that could cause divide-by-zero crash |

| **Compliant Code** |
| --- |
| [Compliant description] |
| Void processData(int value) {  Try {  If (value == 0) {  Throw std::invalid\_argument(“Division by zero is not prohibited”);  }  Int result = 100 / value;  Std::cout << “Result: “ << result << std::endl;  } catch (const std::exception& e) {  Std::cerr << “Error: “ << e.what() << std::endl;  }  } |

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

| **Principles(s):** Ensure proper exception handling: Prevents undefined behavior and crashes by enforcing structured error management.  Faile securely: Ensures that exceptions are handled properly rather than silently propagating failures |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang static analyzer | latest | exceptionSafetyCheck | Flags unsafe exception handling patterns |
| CppCheck | latest | uncaughtException | Identifies unhandled exceptions that could cause failures to the program |
| SonarQube | latest | Cpp:S1258 | Detects improper exception usage and missing catch blocks |
| CodeQL | latest | Cxx/exception-safety | Analyzes exception handling in order to prevent leaks and crashes |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input sanitization | [STD-008-CXX] | Improper input handling can lead to command injections, XSS, and data corruptions due to security vulnerabilities |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| Std::string userInput;  Std::cin >> userInput;  Std::string query = “SELECT \* FROM users WHERE name = ‘” + userInput + “ ‘ “  executeQuery(query) //executes raw input without sanitization that can lead to SQL injection risks |

| **Compliant Code** |
| --- |
| [Compliant description] |
| Std::string userInput;  Std::cin >> userInput;  Std::string query = “SELECT \* FROM users WHERE name = ?”  preparedStatement stmt = db.prepareStatement(query);  stmt.setString(1, sanitizeInput(userInput));  stmt.executeQuery() //sanitization prevents possible injection |

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

| **Principles(s):** validate and sanitize user input: Ensures that’s all incoming data is correctly filtered and rejects dangerous or unexpected values.  Fail securely: If an invalid input is detected, then system should reject the request instead of processing unstantized data |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SQLMap | Latest | injectionDetection | Identifies SQL vulnerabilities through automated testing tools |
| SonarQube | Latest | inputSanitizationCheck | Detects for improper user input validation in security-sensitive settings |
| Fortify static code analyzer | Latest | inputSanitizationCheck | Detects for improper user input validation in security-sensitive settings |
| CodeQL | Latest | cxx/input-validation | Performs deep analysis for unsafe input handling |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Thread Safety | [STD-009-CXX] | Improper thread synchronization can lead to race conditions, deadlocks and data corruption, This standard enforces safe multithreading practices and ensures predictable execution and data integrity |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| #include <iostream>  #include <fstream>  std::ifstream file(userInput); // Accepts raw input, allowing malicious paths  if (file.is\_open()) {  std::string line;  while (std::getline(file, line)) {  std::cout << line << std::endl;  }  } |

| **Compliant Code** |
| --- |
| [Compliant description] |
| #include <iostream>  #include <fstream>  bool isSafePath(const std::string& path) {  return path.find("..") == std::string::npos; // Blocks path traversal  }  std::ifstream file(isSafePath(userInput) ? userInput : ""); // Only open valid paths  if (file.is\_open()) {  std::string line;  while (std::getline(file, line)) {  std::cout << line << std::endl;  }  } |

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

| **Principles(s):** Enforce secure file access: ensures that all file interaction follow appropriate validation and permission restrictions that prevent unauthorized access and unintended modifications  Fail securely: when an invalid file request happens, the system must handle it safety by rejecting unsafe input instead of exposing or modifying sensitive data |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Tidy | Latest | ThreatSafetyCheck | Flags improper thread synchronization and race conditions. |
| Helgrind | Latest | ThreadAnalyzer | Detects data races and deadlocks within concurrent program |
| Coverity | Latest | ConcurrencyCheck | Scans for multithreading defects that are related to improper locking |
| ThreadSanitizer | Latest | RaceConditionCheck | Identifies thread access violations and synchronization flaws |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Flie handling security | [STD-010-CXX] | This standard enforces secure file permissions, validation and access restrictions that protects system integration. Improper file access controls can lead to unauthorized data exposure, path traversal attacks, as well as unintended file modifications |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| std::ifstream file(userInput); //accepts all input and allows for traversal attacks  if (file.is\_open()) {  std::string line;  while (std::getline(file, line)) {  std::cout << line << std::endl;  }  } |

| **Compliant Code** |
| --- |
| [Compliant description] |
| Std::string sanitizePath(const std::string& userPath) {  If (userPath.find(“…”) != std::string::npos) {  Throw std::invalid\_argument(“Invalid path traversal detected”)  }  Return userPath;  }  Std::ifstream file(sanitizePath(userInput)) // validates path before opening  If (file.is\_open()) {  Std::string line;  While (std::getline(file, line)) {  Std::cout << line << std::endl;  }  } |

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

| **Principles(s):** enforce secure file access: ensures that all file interactions follow proper validation and permission restriction.  Fail securely: if an invalid file request occurs, then the system must handle it safely instead of exposing sensitive data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang static analyzer | Latest | pathTraversalCheck | Flags improper file path handling and traversal vulnerabilities |
| CppCheck | Latest | insecureFileAccess | Detects for improper file validation and unauthorized access risks |
| SonarQube | Latest | FilePermissionsCheck | Scans for insecure file permissions that could lead to sensitive data exposed |
| CodeQL | Latest | Cxx/File-safety | Analyzes file handling to avoid access control voilations |

### 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 can be leveraged to enforce compliance within coding standards outlined with the policy. By embedding security into each phase of the software development lifecycle, Green Pace can continuously enforce the best practices without disrupting development speed. During the planning phase, security policies and threat modeling must be early defined. During development, SAST (static analysis tools) and pre-commit hooks can automatically check for coding standard violations. The building phase should implement dependency scanning and secure configuration management that can prevent vulnerabilities. DAST (dynamic automated security testing) and fuzz testing during the testing phase can validate runtime security. Secure release mechanisms, including container security scanning should be considered in the deployment phase. Lastly, continuous monitoring with SIEM and runtime security intelligence can ensure ongoing security validation during production. The DevSecOps automation model illustrates how security can be seamlessly integrated into Green Pace’s existing DevOps pipeline and ensures enforcement at every stage while upholding efficiency.

### 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 | Critical | High | Medium | Urgent | 5 |
| STD-003-CPP | High | Likely | Medium | High | 4 |
| STD-004-CPP | Critical | High | High | Urgent | 5 |
| STD-005-CPP | High | High | Medium | Critical | 5 |
| STD-006-CPP | Medium | High | Low | High | 3 |
| STD-007-CPP | High | High | Medium | Critical | 5 |
| STD-008-CPP | Critical | High | Medium | Urgent | 5 |
| STD-009-CPP | High | High | Medium | Critical | 5 |
| STD-010-CPP | High | High | Medium | Critical | 5 |
| STD-011-CPP | Medium | Likely | Low | High | 3 |
| STD-012-CPP | High | Unlikely | Medium | Medium | 2 |
| STD-013-CPP | Critical | High | High | Urgent | 5 |
| STD-014-CPP | High | Likely | Medium | High | 4 |

### 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 | Encrypts stored data files, databases, and backups to prevent unauthorized access. Applied to databases, disk storage and cloud storage using ‘AES-256’ or equivalent encryption methods |
| Encryption in flight | Encrypts data that Is transmitted between systems such as network traffic, API calls, and emails. This is enforced via TLS 1.2/1.3 VPNs and secure protocols like HTTPS to protect data from interception. |
| Encryption in use | Encrypts data actively being processed in memory or used in computations. This ensures confidential computing techniques, memory encryption, and secure enclave technologies. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Verifies user identities before granting access. Implementing this via multi-factor authentication, biometric authentication and secure login credentials. |
| Authorization | Controls user permissions specified on roles. This is applied through role-based access control, least privilege principles, and access control lists. |
| Accounting | Tracks user activities and ensures auditability and compliance. Is maintained through audit logs, database change tracking, and security monitoring tools. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 6/15/2025 | Revised for Project One that added coding standards, risks tables. And Triple-A/encryption policy | Cameron Kinney | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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