Green Pace Developer: Security Policy Guide Template



# Green Pace Secure Development Policy

# Gabriel Walls

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

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

[Project One 15](#_Toc52464070)

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

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

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

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

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

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

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

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | All input data must be validated for type, length, format, and range before it is processed. Failing to validate input can lead to vulnerabilities such as buffer overflows, SQL injection, and cross-site scripting. Input should be considered untrusted by default and should be sanitized and verified accordingly. |
| 1. Heed Compiler Warnings | Compiler warnings often indicate problematic or potentially dangerous code. Ignoring these warnings can result in undefined behavior or vulnerabilities in the application. Developers must treat all compiler warnings seriously and resolve them during development. |
| 1. Architect and Design for Security Policies | Security should be embedded into the software architecture from the beginning. This includes identifying threat models, enforcing separation of duties, implementing secure data flows, and designing systems to minimize exposure to attack surfaces. |
| 1. Keep It Simple | Simplicity in design and implementation reduces the chances of introducing errors or security flaws. Complex code is harder to test, maintain, and secure. Simple code is more transparent and more easily understood and reviewed by others. |
| 1. Default Deny | Systems should deny access by default and only allow operations or access when explicitly granted. This principle ensures that unauthorized users or processes do not gain access through oversight or misconfiguration. |
| 1. Adhere to the Principle of Least Privilege | Each component, process, or user should operate with the minimum level of access required to perform its function. This limits the scope of damage that can occur if a component is compromised or misused. |
| 1. Sanitize Data Sent to Other Systems | Before transmitting data to external systems—such as databases, APIs, or other services—data must be sanitized and encoded. This prevents the risk of injection attacks or unintentional data corruption. |
| 1. Practice Defense in Depth | A layered security approach ensures that if one security control fails, others continue to provide protection. This includes input validation, encryption, authentication, access control, and monitoring systems. |
| 1. Use Effective Quality Assurance Techniques | Security testing should be integrated into the QA process. This includes static and dynamic code analysis, fuzz testing, penetration testing, and peer code reviews to identify vulnerabilities early in the development lifecycle. |
| 1. Adopt a Secure Coding Standard | Following a consistent and recognized secure coding standard (such as SEI CERT for C/C++) helps enforce best practices across all developers. It reduces the chance of introducing common security flaws and makes security a repeatable, teachable part of the process. |

### 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 | Use Explicit and Appropriate Data Types |

| **Noncompliant Code** |
| --- |
| Using an implicit data type that may not safely store the result, leading to data truncation or overflow. |
| auto total = 3000000000 + 1; // stored as int, may overflow on 32-bit systems |

| **Compliant Code** |
| --- |
| Using a clearly defined type that matches the size and intention of the data to prevent overflow and ensure portability. |
| uint64\_t total = 3000000000ULL + 1; // explicitly using 64-bit unsigned integer |

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

| **Principles(s):**  **Adopt a Secure Coding Standard** – This standard supports the use of well-defined, portable data types, aligning with secure coding practices that minimize ambiguity and reduce platform-specific vulnerabilities. **Heed Compiler Warnings** – Using explicit data types eliminates ambiguity that often causes compiler warnings, helping developers proactively identify and resolve type-related risks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | typeMismatch | Detects improper or ambiguous type usage that may lead to overflow or truncation |
| Clang-Tidy | 17.0 | |  | | --- | |  |  |  | | --- | | clang-analyzer-core.CallAndMessage | | Highlights misuse or incorrect assumptions about return and argument types |
| SonarQube | 10.1 | cpp:S845 | Flags implicit conversions that may cause precision loss or platform issues |
| Coverity | 2023.12 | |  | | --- | |  |  |  | | --- | | UNSIGNED\_COMPARE / INTEGER\_OVERFLOW | | Detects unsafe type usage, improper comparisons, and potential overflow cases |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Validate and Constrain Data Values |

| **Noncompliant Code** |
| --- |
| Failing to validate a user-provided index value can result in out-of-bounds array access, leading to crashes or memory corruption. |
| int arr[5];  int index;  std::cin >> index;  arr[index] = 10; // no bounds checking |

| **Compliant Code** |
| --- |
| The input is checked to ensure it falls within the valid range of the array before being used, preventing out-of-bounds access. |
| int arr[5];  int index;  std::cin >> index;  if (index >= 0 && index < 5) {  arr[index] = 10;  } else {  std::cerr << "Invalid index\n";  } |

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

| **Principles(s):**  **Validate Input Data** – This standard directly reflects the principle of input validation to prevent unauthorized behavior like out-of-bounds access, which could lead to memory corruption or crashes. **Default Deny** – By rejecting values outside the valid range, the system denies unsafe input by default, allowing only what’s explicitly valid. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | arrayIndexOutOfBounds | Identifies unchecked array indexing that can lead to runtime crashes |
| Clang-Tidy | 17.0 | |  | | --- | |  |  |  | | --- | | clang-analyzer-core.NullDereference | | Warns about unchecked index and pointer dereferencing |
| SonarQube | 10.1 | cpp:S3519 | Flags possible array index violations from unchecked user input |
| Fortify SCA | 23.1 | Buffer Overflow, Strcpy | Fortify's static code analyzer flags unsafe string manipulation and recommends secure alternatives |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Ensure Safe and Bounded String Operations |

| **Noncompliant Code** |
| --- |
| Using strcpy() without bounds checking may overflow the destination buffer if the source string is too long. |
| char dest[10];  strcpy(dest, "This is way too long"); // Buffer overflow risk |

| **Compliant Code** |
| --- |
| Using strncpy() or std::string provides a safer, bounded approach to copying strings and prevents buffer overflows. |
| char dest[10];  strncpy(dest, "Safe", sizeof(dest) - 1);  dest[sizeof(dest) - 1] = '\0'; // Ensure null-termination  Or using modern C++:  std::string source = "Safe";  std::string dest = source; // Safe copy using std::string |

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

| **Principles(s):**  **Validate Input Data** – Ensuring string lengths are controlled helps validate that inputs cannot overflow buffers or corrupt memory. **Sanitize Data Sent to Other Systems** – Safe string handling also prevents unintentional leakage or injection when passing data between systems, particularly in legacy C-style functions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.10 | bufferOverrun, strFunctionUsage | Detects common string overflows and improper use of unsafe functions like strcpy |
| Clang-Tidy | 17.0 | cert-str34-c | Checks for dangerous string handling in C++ per SEI CERT guidelines |
| SonarQube | 10.1 | cpp:S3510 | Detects buffer overflows caused by unbounded string operations |
| Fortify SCA | 23.1 | Buffer Overflow, Strcpy | Fortify's static code analyzer flags unsafe string manipulation and recommends secure alternatives |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Prevent SQL Injection Using Parameterized Queries |

| **Noncompliant Code** |
| --- |
| Building an SQL query using string concatenation with user input directly, making it vulnerable to SQL injection. |
| std::string user = "admin'; DROP TABLE users; --";  std::string query = "SELECT \* FROM users WHERE username = '" + user + "';";  db.execute(query); // vulnerable |

| **Compliant Code** |
| --- |
| Using parameterized queries ensures that user input is treated as data, not executable SQL, thereby preventing injection attacks. |
| std::string query = "SELECT \* FROM users WHERE username = ?";  PreparedStatement stmt = db.prepare(query);  stmt.bind(1, user);  stmt.execute(); // safe |

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

| **Principles(s):**  **Validate Input Data** – Parameterized queries ensure user input is parsed and validated as data, not as part of the executable query string. **Sanitize Data Sent to Other Systems** – By binding variables instead of concatenating input, data sent to the database is sanitized and safe from injection. **Adopt a Secure Coding Standard** – This standard reinforces secure data handling protocols and database interaction patterns per SEI CERT rules. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.1 | cpp:S3649 | Detects SQL statements vulnerable to injection based on unsafe concatenation |
| Fortify SCA | 23.1 | SQL Injection | Flags user input passed into SQL queries without proper sanitization |
| CodeQL | 2.14 | cpp/injection/sql-injection | Identifies potential SQL injection paths in C++ source code |
| Checkmarx | 9.6 | CX-2066 | Static rule to detect unparameterized SQL queries built with string operations |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Prevent Memory Leaks and Access Violations |

| **Noncompliant Code** |
| --- |
| Manual memory management without checks can lead to memory leaks and use-after-free errors. |
| int\* data = new int[10];  // ... some operations ...  delete[] data;  delete[] data; // double delete - undefined behavior |

| **Compliant Code** |
| --- |
| Using smart pointers ensures automatic deallocation and helps prevent memory misuse. |
| #include <memory>  std::unique\_ptr<int[]> data = std::make\_unique<int[]>(10);  // memory is automatically released when `data` goes out of scope |

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

| **Principles(s):**  **Adopt a Secure Coding Standard** – Encourages the use of smart pointers and modern memory practices to reduce common C++ memory issues. **Use Effective Quality Assurance Techniques** – Memory issues like double deletes and leaks are hard to detect without thorough testing and static analysis tools. **Practice Defense in Depth** – Automating memory safety via smart pointers adds a layer of runtime protection to mitigate human error. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.21 | Memcheck | Runtime tool to detect memory leaks, invalid frees, and use-after-free |
| Cppcheck | 2.10 | |  | | --- | |  |  |  | | --- | | memoryLeak, useAfterFree | | Static analysis to detect memory management issues |
| Clang Static Analyzer | 17.0 | unix.Malloc, cplusplus.NewDelete | Detects improper memory deallocations and mismatches in allocation/deletion |
| Coverity | 2023.12 | USE\_AFTER\_FREE, MEM\_LEAK | Identifies dangerous memory access patterns and unfreed resources |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Use Assertions to Enforce Assumptions During Development |

| **Noncompliant Code** |
| --- |
| Failing to verify that a pointer is non-null before dereferencing it may lead to a crash or undefined behavior. |
| int\* ptr = getPointer();  int value = \*ptr; // no check for nullptr |

| **Compliant Code** |
| --- |
| An assertion ensures that the pointer is valid during development. This catches issues early before deployment. |
| #include <cassert>  int\* ptr = getPointer();  assert(ptr != nullptr); // assert that ptr must not be null  int value = \*ptr; |

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

| **Principles(s):**  **Use Effective Quality Assurance Techniques** – Assertions allow developers to identify invalid assumptions and edge cases during development, improving QA coverage. **Architect and Design for Security Policies** – Early assertion checks act as enforcement points to catch logical flaws before deployment, making the code more robust. **Keep It Simple** – Assertions simplify error checking by clearly expressing developer intent and expected invariants in code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17.0 | clang-analyzer-core.NullDereference | Identifies potential null pointer dereference and missing assertions |
| Cppcheck | 2.10 | nullPointer | Detects dereferencing of possibly null pointers |
| Coverity | 2023.12 | ASSERT\_SIDE\_EFFECT, NULL\_RETURNS | Highlights improper or missing use of assertions for return validation |
| PVS-Studio | 7.25 | V595, V522 | Detects missed pointer validity checks and unsafe dereferencing patterns |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Use Exceptions for Error Handling, Not Control Flow |

| **Noncompliant Code** |
| --- |
| Using exceptions to control normal logic flow is inefficient and confusing. |
| try {  throw 1;  } catch (int e) {  std::cout << "Handled value logic: " << e << std::endl;  } |

| **Compliant Code** |
| --- |
| Exceptions are reserved for actual errors. Regular control flow is managed using standard conditional statements. |
| int value = getValue();  if (value >= 0) {  std::cout << "Value logic handled: " << value << std::endl;  } else {  throw std::runtime\_error("Invalid value received.");  } |

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

| **Principles(s):**  **Keep It Simple** – Exceptions should only handle abnormal events; using them for standard flow makes code complex and harder to maintain. **Use Effective Quality Assurance Techniques** – This practice allows better static analysis and testing coverage, as exception logic is isolated and intentional. **Adopt a Secure Coding Standard** – Follows SEI CERT C++ best practices regarding structured exception use and flow control separation. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Low | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 17.0 | cert-err60-cpp | Flags use of exceptions for non-error control paths |
| Cppcheck | 2.10 | throwInTryBlock | Warns when throw/catch blocks are misused or inefficient |
| SonarQube | 10.1 | cpp:S3626 | Identifies exception misuse in regular control logic |
| Coverity | 2023.12 | CONTROL\_FLOW\_EXCEPTION | Detects logic that incorrectly relies on exceptions for standard program flow |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Logging Practices | STD-008-CPP | Sanitize Sensitive Data in Logs |

| **Noncompliant Code** |
| --- |
| Logs sensitive user credentials in plaintext, exposing them to anyone with access to the logs. |
| std::string username = "user1";  std::string password = "hunter2";  std::cout << "Login attempt: " << username << ", Password: " << password << std::endl; |

| **Compliant Code** |
| --- |
| Logs general authentication activity without exposing user passwords or sensitive data. |
| std::string username = "user1";  // authentication logic...  std::cout << "Login attempt by user: " << username << std::endl; |

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

| **Principles(s):**  **Sanitize Data Sent to Other Systems** – Logging sensitive information without sanitization exposes user credentials and violates security best practices. **Practice Defense in Depth** – Preventing sensitive data exposure through logs adds another security layer in case of a system breach. **Adopt a Secure Coding Standard** – Reinforces data classification and handling discipline to protect personally identifiable and confidential information. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.1 | cpp:S2068 | Detects logging of sensitive data like credentials, tokens, and secrets |
| Semgrep | 1.47 | logging-password, logging-secrets | Rules to detect common patterns of unsafe logging practices |
| Fortify SCA | 23.1 | Information Exposure Through Log Files | Flags sensitive data being printed to log or console |
| Logwatch/Logcheck | N/A | Pattern matching | Monitors log output for exposure of credentials or confidential information |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Concurrency and Thread Safety | STD-009-CPP | Protect Shared Data with Synchronization Primitives |

| **Noncompliant Code** |
| --- |
| Two threads accessing and modifying shared data without synchronization, leading to race conditions. |
| int counter = 0;  void increment() {  for (int i = 0; i < 1000; ++i) {  counter++; // not thread-safe  }  } |

| **Compliant Code** |
| --- |
| A mutex is used to protect access to shared data, preventing concurrent writes and race conditions. |
| #include <mutex>  int counter = 0;  std::mutex mtx;  void increment() {  for (int i = 0; i < 1000; ++i) {  std::lock\_guard<std::mutex> lock(mtx);  counter++; // thread-safe  }  } |

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

| **Principles(s):**  **Adopt a Secure Coding Standard** – Promotes the use of synchronization primitives to prevent race conditions and ensure thread safety. **Practice Defense in Depth** – Synchronization is a runtime safeguard that complements compile-time safety, catching concurrency issues during execution. **Use Effective Quality Assurance Techniques** – Static and dynamic analysis can reveal race conditions, ensuring concurrent code behaves as expected under load. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ThreadSanitizer | LLVM 17.0 | data race, mutex misuses | Runtime tool that detects data races and unsafe synchronization |
| Cppcheck | 2.10 | raceCondition | Warns when shared data is modified without proper synchronization |
| Coverity | 2023.12 | GUARDED\_BY\_VIOLATION, UNGUARDED\_WRITE | Identifies concurrency violations and mutex misapplications |
| Helgrind (Valgrind tool) | 3.21 | race conditions, thread order violations | Detects threading issues at runtime for multi-threaded applications |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File I/O Handling | STD-010-CPP | Validate File Paths and Handle File Access Errors |

| **Noncompliant Code** |
| --- |
| Fails to check if the file was opened successfully and assumes the path is safe. |
| std::ifstream file("../../../etc/passwd");  std::string line;  while (std::getline(file, line)) {  std::cout << line << std::endl;  } |

| **Compliant Code** |
| --- |
| Verifies file access and restricts paths to known-safe directories to avoid path traversal or access errors. |
| #include <fstream>  #include <iostream>  #include <filesystem>  void readSafeFile(const std::string& filename) {  std::filesystem::path safeBase = "/safe\_directory";  std::filesystem::path fullPath = safeBase / filename;  if (!fullPath.string().starts\_with(safeBase.string())) {  std::cerr << "Path traversal attempt blocked.\n";  return;  }  std::ifstream file(fullPath);  if (!file.is\_open()) {  std::cerr << "Failed to open file.\n";  return;  }  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):**  **Validate Input Data** – Ensures user-provided paths are properly vetted before allowing file access. **Architect and Design for Security Policies** – Supports secure file access by enforcing trusted directory boundaries and validating file system operations. **Practice Defense in Depth** – Prevents unauthorized access and file traversal by validating both access permissions and input format. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.1 | Cpp:S2083 | Detects potential path traversal from unvalidated file path input |
| Cppcheck | 2.10 | fileAccess, taintedPath | Warns when input paths are used without proper validation or access control |
| Coverity | 2023.12 | PATH\_TRAVERSAL, TOCTOU | Identifies unvalidated paths and unsafe file operations |
| Semgrep | 1.47 | cpp.lang.security.path-traversal | Scans for file access patterns vulnerable to directory traversal attacks |

### 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.

[Insert your written explanations here.]

### 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 | High | 4 |
| STD-003-CPP | High | Likely | Medium | High | 4 |
| STD-004-CPP | High | Likely | Medium | High | 4 |
| STD-005-CPP | High | Likely | Medium | High | 4 |
| STD-006-CPP | Medium | Possible | Low | Medium | 3 |
| STD-007-CPP | Medium | Unlikely | Low | Medium | 2 |
| STD-008-CPP | High | Likely | Low | High | 4 |
| STD-009-CPP | High | Likely | Medium | High | 4 |
| STD-010-CPP | High | Likely | Low | High | 4 |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Likely | Low | High | 4 |
| STD-003-CPP | High | Likely | Medium | High | 4 |
| STD-004-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 | Encryption at rest protects data stored on physical media (like hard drives, backups, and databases). It ensures that if a storage device is lost, stolen, or improperly decommissioned, the data remains unreadable. Green Pace uses AES-256 encryption as the default method for all customer and operational data stored at rest. This policy applies to servers, endpoints, cloud buckets, and archives. |
| Encryption in flight | Encryption in flight secures data as it travels between systems across networks (internal or external). Green Pace mandates the use of TLS 1.2 or higher for all HTTP/S services, REST APIs, email transmissions, and VPN tunnels. This ensures confidentiality and integrity during transmission, protecting against eavesdropping or tampering. |
| Encryption in use | Encryption in use refers to protecting data while it is actively being processed in memory. Green Pace requires the use of secure enclaves or hardware-level memory encryption for highly sensitive workloads (e.g., key management, financial computations). This helps prevent exposure from memory scraping, side-channel attacks, and debugging exploits. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying a user’s identity before granting access to systems or data. Green Pace requires MFA (multi-factor authentication) for all administrative access and SSO integration for employee accounts. This prevents unauthorized logins and ensures accountability. |
| Authorization | Authorization determines what authenticated users are allowed to do. Role-Based Access Control (RBAC) is enforced to ensure users only access resources necessary for their job roles. Admin rights, database write access, and system configurations are limited to approved roles. |
| Accounting | Accounting tracks what users do: logins, file access, database modifications, and changes to privileges. Logs are stored centrally and integrated with SIEM for continuous monitoring. This provides a complete audit trail for compliance and forensic investigations. |

**\***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 | 5/22/2025 | Completed Ten Core Security Principles & Coding Standards 1–10 | Gabriel Walls | [Insert text.] |
| 1.2 | 6/11/2025 | Completed Risk Assessment, Automation, Encryption, Triple-A, and Version History | Gabriel Walls | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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