**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 treat external data as untrusted until proven safe. Whether it comes from a user, file, or network, input should be checked for format, length, and content to prevent unexpected behavior or vulnerabilities like injection attacks. |
| 1. Heed Compiler Warnings | Compilers often generate warnings that can indicate poor practices or dangerous operations. Paying attention to these warnings and resolving them proactively helps catch bugs early and improves the overall safety of the code. |
| 1. Architect and Design for Security Policies | Security shouldn't be an afterthought. From the start of development, the system architecture and design should include decisions that support confidentiality, integrity, and availability. This makes it harder for attackers to exploit the system later on. |
| 1. Keep It Simple | Complex systems are harder to understand and easier to break. Writing simple, clear code with straightforward logic makes it easier to review, test, and secure, reducing the risk of hidden vulnerabilities. |
| 1. Default Deny | If access is not explicitly granted, it should be denied. This principle ensures that only intended users or processes are allowed to interact with data or perform sensitive actions, minimizing the attack surface by default. |
| 1. Adhere to the Principle of Least Privilege | Each component, process, or user should operate with only the minimum permissions necessary. Limiting access in this way reduces the potential damage from compromised parts of the system or misused privileges. |
| 1. Sanitize Data Sent to Other Systems | Any data that leaves your application and is used by another system, such as a database, browser, or file handler, should be cleaned and validated to prevent injection attacks, corruption, or misuse. |
| 1. Practice Defense in Depth | Relying on one layer of protection is risky. Instead, use multiple layers, like input validation, firewalls, authentication, and logging, so that if one layer fails, others can still protect the system. |
| 1. Use Effective Quality Assurance Techniques | Security bugs can hide in plain sight, so use thorough testing methods such as code reviews, unit tests, and static analysis to find and fix them early. QA isn't just for functionality, it's essential for security too. |
| 1. Adopt a Secure Coding Standard | Following a consistent set of coding rules helps prevent common errors that lead to security flaws. Secure coding standards like those from SEI CERT provide guidance on safe patterns to use and pitfalls to avoid. |

### 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** | **Avoid mixing incompatible data types in arithmetic operations** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Mixing different data types, such as signed and unsigned integers, in arithmetic operations can lead to unexpected behavior or incorrect results. For example, subtracting a larger unsigned value from a smaller signed value may result in a large positive number due to wrap-around. Ensuring operands are of compatible types helps prevent subtle bugs and improves the safety and reliability of the code. |

| **Noncompliant Code** |
| --- |
| This code mixes a signed integer with an unsigned value in a subtraction operation, which can lead to wrap-around and incorrect results. |
| #include <iostream>  void printDifference(int userInput) {  unsigned int threshold = 10;  int diff = userInput - threshold;  std::cout << "Difference: " << diff << std::endl;  } |

| **Compliant Code** |
| --- |
| The compliant version ensures both operands are of the same signed type before performing subtraction, avoiding unexpected wrap-around or sign errors. |
| #include <iostream>  void printDifference(int userInput) {  int threshold = 10;  int diff = userInput - threshold;  std::cout << "Difference: " << diff << std::endl;  } |

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

| **Principles(s):**  2: Heed Compiler Warnings - Heed compiler warnings to catch sign conversions.  9: Use Effective Quality Assurance Techniques - QA gates enforce warnings as errors.  10: Adopt a Secure Coding Standard - Adopting a secure coding standard bans unsafe implicit conversions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| GCC | 13 | -Wconversion, -Wsign-conversion | Warns on signed and unsigned mixing and narrowing |
| MSVC | 19.40 | C4365 | Signed/unsigned mismatch warning |
| Clang-tidy | 18 | cppcoreguidelines-narrowing-conversions | Flags narrowing and unsafe implicit conversions. |
| |  | | --- | | Cppcheck |  |  | | --- | |  | | 2.13 | mixedSignComparison | Reports mixed signed and unsigned operations. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Prevent use of hardcoded constants in logic checks** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Using hardcoded values, often called "magic numbers", in logic or configuration makes code harder to understand, maintain, and secure. When values are repeated or directly embedded into conditional logic, it becomes easy to overlook them during updates. This can introduce bugs, logic errors, or vulnerabilities if thresholds or limits change in the future. By defining constants with meaningful names, developers ensure consistency and improve clarity throughout the codebase. |

| **Noncompliant Code** |
| --- |
| This code compares a user’s points to a hardcoded value of 100, which makes the logic harder to update and understand later. |
| #include <iostream>  bool isEligible(int points) {  if (points > 100) {  return true;  }  return false;  } |

| **Compliant Code** |
| --- |
| This version replaces the hardcoded value with a named constant, making the code easier to understand and safer to modify in the future. |
| #include <iostream>  const int MAX\_POINTS = 100;  bool isEligible(int points) {  if (points > MAX\_POINTS) {  return true;  }  return false;  } |

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

| **Principles(s):**  4: Keep It Simple – Named constants make logic easy to review and maintain. 9: Use Effective Quality Assurance Techniques – QA processes verify that constants replace hardcoded values. 10: Adopt a Secure Coding Standard – The standard requires symbolic constants for thresholds. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-tidy | 18 | Readability-magic-numbers | Detects unnamed numeric literals |
| SonarQube CFamily | 10.6 | Magic numbers should not be used | Flags repeated literals in conditions |
| Cppcheck | 2.13 | Style magicnumbers | Warns about unnamed numeric literals |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Ensure proper string termination and bounds checking** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | When working with C-style strings, failing to properly null-terminate a string or allowing input to exceed the buffer size can lead to serious security issues, such as buffer overflows or undefined behavior. It is important to always allocate space for the null terminator and perform proper bounds checking when copying or reading string data to ensure the integrity and safety of the program. |

| **Noncompliant Code** |
| --- |
| This code copies user input into a fixed-size character array without checking the input length, which may lead to buffer overflow and memory corruption. |
| #include <iostream>  #include <cstring>  void storeName(const char\* input) {  char name[10];  std::strcpy(name, input); // Unsafe: no length check  std::cout << "Name: " << name << std::endl;  } |

| **Compliant Code** |
| --- |
| This version uses strncpy with a size limit to ensure that no more characters than the buffer can hold are copied, and manually null-terminates the string for safety. |
| #include <iostream>  #include <cstring>  void storeName(const char\* input) {  char name[10];  std::strncpy(name, input, sizeof(name) - 1);  name[9] = '\0'; // Ensure null termination  std::cout << "Name: " << name << std::endl;  } |

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

| **Principles(s):**  1: Validate Input Data – Validate string lengths to prevent overflow. 8: Practice Defense in Depth – Layer multiple controls against buffer overflows. 10: Adopt a Secure Coding Standard – The standard forbids unsafe string handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 18 | security.insecureAPI.strcpy and related | Flags insecure C string APIs |
| Cppcheck | 2.13 | bufferAccessOutOfBounds, invalidFunctionArgs | Detects overruns and size misuse |
| SonarQube CFamily | 10.6 | Rule set for insecure string functions | Reports unsafe APIs and missing checks |
| AddressSanitizer | 18 | ASan runtime | Finds out-of-bounds and use-after-free in tests |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Avoid constructing SQL queries with direct user input** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Directly inserting user input into SQL statements without proper validation or escaping can expose the application to SQL injection attacks. This type of vulnerability allows attackers to manipulate the structure of a query, potentially gaining unauthorized access to data or executing harmful commands. Using prepared statements or parameterized queries helps separate code from data and ensures user input is handled safely. |

| **Noncompliant Code** |
| --- |
| This code builds an SQL query by joining user input directly into the string, which can be exploited by attackers to modify the intended query logic. |
| #include <iostream>  #include <string>  void findUser(const std::string& username) {  std::string query = "SELECT \* FROM users WHERE name = '" + username + "';";  std::cout << "Executing: " << query << std::endl;  // Assume query gets passed to a database API  } |

| **Compliant Code** |
| --- |
| This version uses a parameterized approach where the SQL query is defined with placeholders and the input is bound safely, preventing injection. |
| #include <iostream>  #include <string>  // This is a mockup to illustrate parameter usage  void findUserSafe(const std::string& username) {  std::string query = "SELECT \* FROM users WHERE name = ?";  std::cout << "Preparing statement: " << query << std::endl;  std::cout << "Binding parameter: " << username << std::endl;  // Assume query and parameter are passed to a secure database interface  } |

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

| **Principles(s):**  1: Validate Input Data – Sanitize and validate all SQL parameters. 5: Default Deny – Deny database access unless explicitly allowed. 7: Sanitize Data Sent to Other Systems – Clean all outbound SQL data. 8: Practice Defense in Depth – Combine parameterization, validation, and privilege controls. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeQL | 2.17 | Cpp/sql-injection | Taint tracking from untrusted sources to SQL sinks |
| Semgrep | 1.67 | Cpp.lang.security.sql-injection | Detects string concatenation in SQL calls |
| SonarQube CFamily | 10.6 | S3649 and related injection rules | Flags concatenated SQL and missing parameters |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Release dynamically allocated memory properly** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Failing to release memory that has been allocated dynamically can lead to memory leaks, which over time may exhaust system resources and degrade application performance. On the other hand, releasing memory improperly, such as deleting the same pointer twice or accessing memory after deletion, can lead to undefined behavior or crashes. Developers should follow safe memory management practices and use smart pointers when possible to reduce these risks. |

| **Noncompliant Code** |
| --- |
| This code allocates memory with new but never deallocates it, causing a memory leak every time the function is called. |
| #include <iostream>  void allocateData() {  int\* data = new int[100];  data[0] = 42;  std::cout << "First element: " << data[0] << std::endl;  // Memory is never released  } |

| **Compliant Code** |
| --- |
| This version uses a smart pointer (std::unique\_ptr) to ensure the memory is automatically released when it goes out of scope. |
| #include <iostream>  #include <memory>  void allocateData() {  std::unique\_ptr<int[]> data(new int[100]);  data[0] = 42;  std::cout << "First element: " << data[0] << std::endl;  // Memory is released automatically  } |

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

| **Principles(s):**  3: Architect and Design for Security Policies – Design memory management for safety from the start. 8: Practice Defense in Depth – Use multiple layers of runtime and static analysis to catch leaks. 10: Adopt a Secure Coding Standard – Follow standards requiring RAII or smart pointers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 18 | [Insert text.] | Finds leaks and mismatched allocation |
| AddressSanitizer | 18 | Asan/LSan runtime | Detects leaks, double free, invalid free |
| Valgrind | 3.22 | [Insert text.] | Reports leaks and invalid memory use |
| Cppcheck | 2.13 | Memleak, doubleFree, nullPointer | Static leak and misuse patterns |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use assertions to catch invalid assumptions during development** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions are useful tools during development to verify that certain conditions hold true at specific points in the program. When used appropriately, they can detect bugs early by stopping execution if an assumption is violated. However, relying on assertions for input validation or in production code can be risky, since assertions may be disabled in release builds. They should only be used to catch logic errors and incorrect internal states during testing and debugging. |

| **Noncompliant Code** |
| --- |
| This code uses an assertion to check user input, which is unreliable because assertions may be disabled in production builds. |
| #include <cassert>  #include <iostream>  void processInput(int value) {  assert(value >= 0); // Unsafe: may be ignored in production  std::cout << "Processing value: " << value << std::endl;  } |

| **Compliant Code** |
| --- |
| This version uses a runtime check for user input and reserves the assertion for verifying internal assumptions that should never fail during development. |
| #include <cassert>  #include <iostream>  void processInput(int value) {  if (value < 0) {  std::cerr << "Error: negative value provided." << std::endl;  return;  }  assert(value >= 0); // Valid as a developer check  std::cout << "Processing value: " << value << std::endl;  } |

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

| **Principles(s):**  9: Use Effective Quality Assurance Techniques – Test assertions for correctness in development builds. 10: Adopt a Secure Coding Standard – Ensure asserts are used only for internal logic, not input checks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-tidy | 18 | Misc-assert-side-effect | Prevents side effects in assert conditions |
| SonarQube CFamily | 10.6 | Assertions should not be used for argument validation | Flags misuse of asserts for input checks |
| Semgrep | 1.67 | Custom local rule | Ban asserts on untrusted inputs across the codebase |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Avoid throwing exceptions from destructors** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Throwing exceptions from destructors can cause program termination if another exception is already active. This is especially dangerous during stack unwinding, when destructors are automatically called after an exception has been thrown. If a second exception is thrown at that time, the program may crash due to the conflict. Destructors should handle errors internally or log them instead of throwing new exceptions. |

| **Noncompliant Code** |
| --- |
| This destructor throws an exception if a cleanup step fails, which is dangerous during stack unwinding. |
| #include <iostream>  #include <stdexcept>  class ResourceHandler {  public:  ~ResourceHandler() {  // Risky: throwing in a destructor  throw std::runtime\_error("Failed to release resource");  }  }; |

| **Compliant Code** |
| --- |
| This version logs the error inside the destructor instead of throwing an exception, avoiding problems during exception handling. |
| #include <iostream>  class ResourceHandler {  public:  ~ResourceHandler() {  if (!releaseResource()) {  std::cerr << "Warning: failed to release resource" << std::endl;  }  }  private:  bool releaseResource() {  // Simulated failure  return false;  }  }; |

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

| **Principles(s):**  3: Architect and Design for Security Policies – Design destructors to handle errors without throwing. 4: Keep It Simple – Avoid complex error handling paths in destructors. 9: Use Effective Quality Assurance Techniques – Test destructor behavior to confirm no exceptions are thrown. 10: Adopt a Secure Coding Standard – Follow coding rules that forbid throwing from destructors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-tidy | 18 | cppcoreguidelines-noexcept-destructor, cert-dcl58-cpp | Recommends noexcept and flags throwing destructors |
| SonarQube CFamily | 10.6 | Destructors should not throw exceptions | Detects throw destructors |
| Cppcheck | 2.13 | exceptThrowInDestructor | Warns when destructor contains a throw |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Initialize all pointers before use** |
| --- | --- | --- |
| Memory | [STD-008-CPP] | Using uninitialized pointers can result in undefined behavior, including memory corruption and program crashes. If a pointer is not assigned a valid memory address before it is accessed or dereferenced, it may point to arbitrary or restricted memory. This can lead to difficult-to-debug errors or security vulnerabilities. Initializing pointers to a known value such as nullptr ensures they are safe to check before use and helps catch mistakes early. |

| **Noncompliant Code** |
| --- |
| This code declares a pointer but never initializes it. Dereferencing the pointer results in undefined behavior. |
| #include <iostream>  void printValue() {  int\* ptr;  \*ptr = 5; // Undefined behavior: ptr not initialized  std::cout << \*ptr << std::endl;  } |

| **Compliant Code** |
| --- |
| This version initializes the pointer to nullptr and checks it before dereferencing, preventing undefined behavior. |
| #include <iostream>  void printValue() {  int\* ptr = nullptr;  if (ptr != nullptr) {  \*ptr = 5;  std::cout << \*ptr << std::endl;  } else {  std::cout << "Pointer is null." << std::endl;  }  } |

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

| **Principles(s):**  2: Heed Compiler Warnings – Enable and respond to warnings about uninitialized use. 8: Practice Defense in Depth – Add runtime checks for pointer safety. 9: Use Effective Quality Assurance Techniques – Test pointer initialization and usage paths. 10: Adopt a Secure Coding Standard – Require initialization to nullptr before use. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 18 | core.uninitialized.Assign | Reports use of uninitialized values |
| Clang | 18 | -Wuninitialized | Compile-time warning for uninitialized use. |
| Cppcheck | 2.13 | Uninitvar, nullpointer | Detects uninitialized variables and null dereferences. |
| AddressSanitizer | 18 | ASan runtime | Catches invalid memory access during tests. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Validate numeric input ranges before processing** |
| --- | --- | --- |
| Input Validation | [STD-009-CPP] | Numeric input from users, files, or external sources should always be validated to ensure it falls within an expected and safe range. Failing to do so can lead to out-of-bounds access, division by zero, or excessive memory usage. Proper validation helps protect the program from both accidental and malicious input, improving reliability and security. |

| **Noncompliant Code** |
| --- |
| This code uses user-provided input as an array index without checking whether the value is within valid bounds, risking out-of-bounds access. |
| #include <iostream>  void accessArray(int index) {  int values[5] = {1, 2, 3, 4, 5};  std::cout << "Value: " << values[index] << std::endl; // No bounds check  } |

| **Compliant Code** |
| --- |
| This version validates the input before using it as an index, ensuring safe access within array bounds. |
| #include <iostream>  void accessArray(int index) {  int values[5] = {1, 2, 3, 4, 5};  if (index >= 0 && index < 5) {  std::cout << "Value: " << values[index] << std::endl;  } else {  std::cout << "Error: index out of bounds." << std::endl;  }  } |

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

| **Principles(s):**  1: Validate Input Data – Check numeric inputs for safe ranges. 8: Practice Defense in Depth – Combine static checks, runtime checks, and fuzzing. 9: Use Effective Quality Assurance Techniques – Test edge cases and invalid inputs. 10: Adopt a Secure Coding Standard – Enforce range validation rules. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeQL | 2.17 | cpp/array-index-out-of-bounds | Finds indexes that may exceed bounds |
| clang-tidy | 18 | cppcoreguidelines-pro-bounds-constant-array-index | Flags constant out-of-range indexes. |
| SonarQube CFamily | 10.6 | Array bounds rules set | Detects potential out-of-bounds access |
| AddressSanitizer | 18 | Asan runtime | Detects out-of-bounds at runtime in tests |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Always close file handles after use** |
| --- | --- | --- |
| Resource Management | [STD-010-CPP] | File handles and other system resources should be released promptly after use to avoid resource leaks, locking conflicts, and potential data integrity problems. Failing to close a file properly may also prevent data from being fully written to disk. Explicitly closing files ensures that system resources are returned and that files are safely flushed and finalized. |

| **Noncompliant Code** |
| --- |
| This code opens a file and writes to it, but never closes the stream, risking a resource leak or data corruption. |
| #include <fstream>  void writeFile() {  std::ofstream out("data.txt");  out << "Writing to file.";  // File is never explicitly closed  } |

| **Compliant Code** |
| --- |
| This version uses RAII with a local ofstream object, which ensures the file is properly closed when the object goes out of scope. |
| #include <fstream>  void writeFile() {  {  std::ofstream out("data.txt");  out << "Writing to file.";  } // File is closed automatically here  } |

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

| **Principles(s):**  3: Architect and Design for Security Policies – Design resource handling so files close automatically. 8: Practice Defense in Depth – Use both RAII and static leak detection. 9: Use Effective Quality Assurance Techniques – Test for proper closure in all code paths. 10: Adopt a Secure Coding Standard – Require proper closure of all file handles. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube CFamily | 10.6 | S2095 “Resources should be closed” | Reports streams and file handles not closed |
| Clang Static Analyzer | 18 | alpha.unix.Stream | Flags leaks of file descriptors and streams |
| Cppcheck | 2.13 | resourceLeak | Detects missing close on files or sockets |

### 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 will be integrated into each stage of the DevSecOps pipeline to enforce and verify compliance with all 10 coding standards.

During **Plan**, the team addresses technical security debt, trains on security tools, and establishes rule sets in tools like SonarQube and clang-tidy.

During **Create**, developers enable IDE plug-ins (SonarLint, clang-tidy) to surface violations immediately while coding.

During **Verify**, the build process automatically runs static analysis tools (Cppcheck, clang-tidy, SonarQube CFamily) and enforces quality gates. Failing any gate tied to these standards blocks the build.

In **Pre-Production**, dynamic analysis is added with fuzzing, integration tests, and AddressSanitizer/LeakSanitizer enabled.

At **Release**, software is signed, and secure artifact handling is verified.

In **Prevent** and **Detect**, runtime monitoring tools (RASP, integrity checks, penetration testing) are used to identify deviations.

During **Respond**, automated playbooks run via security orchestration tools, containing threats and reporting incidents.

In **Predict**, vulnerability trend analysis is performed using CodeQL queries and SonarQube history to target high-risk code areas.

In **Adapt**, findings are fed back into planning, updating security rules and training content.

### 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 | Medium | Possible | Low | Medium | 2 |
| STD-002-CPP | Low | Likely | Low | Medium | 2 |
| STD-003-CPP | High | Likely | Medium | High | 4 |
| STD-004-CPP | High | Possible | Low | Very High | 5 |
| STD-005-CPP | High | Likely | Medium | High | 4 |
| STD-006-CPP | Medium | Possible | Low | Medium | 2 |
| STD-007-CPP | High | Unlikely | Medium | High | 3 |
| STD-008-CPP | High | Possible | Low | High | 3 |
| STD-009-CPP | High | Likely | Low | High | 4 |
| STD-010-CPP | Medium | Possible | Low | High | 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 | Data stored on physical or virtual media, including disks, backups, and database files, must be encrypted using AES-256 or stronger algorithms. Keys must be stored securely in a Key Management System (KMS) or Hardware Security Module (HSM). This ensures that if storage media is lost, stolen, or accessed without authorization, the data remains protected. Policy applies to all storage locations containing sensitive or regulated data. |
| Encryption in flight | Data transmitted across networks, including internal and external communications, must be protected with TLS 1.3 or stronger using approved cipher suites. Mutual TLS (mTLS) should be used for service-to-service communication where feasible. This prevents interception, tampering, or replay attacks. Policy applies to all data sent between systems, services, or clients. |
| Encryption in use | Sensitive data processed in memory must be protected through secure coding practices, such as clearing secrets from memory after use, using hardware-backed secure enclaves, and applying field-level encryption for highly sensitive values. This reduces the risk from memory dumps or compromised hosts. Policy applies whenever sensitive information is actively processed in an application. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | The process of verifying the identity of a user or service. All human users must authenticate via Single Sign-On (SSO) with Multi-Factor Authentication (MFA). Service accounts must use short-lived tokens and must not embed credentials in code. All user logins must be recorded with timestamp, source IP, and identity information to provide traceability and detect suspicious activity. This ensures only verified identities can access systems, reducing the risk of unauthorized access. |
| Authorization | The process of determining what actions an authenticated identity can perform. Role-Based Access Control (RBAC) with the principle of least privilege must be enforced, and roles must be reviewed quarterly. Changes to the database, addition of new users, and modifications to a user’s level of access must be logged and approved according to security policy. Sensitive actions require elevated privileges or additional approval. This minimizes damage from compromised accounts and enforces proper separation of duties. |
| Accounting | The process of recording and reviewing actions performed by authenticated identities. All authentication events, privilege changes, file access, and critical system actions must be logged with user ID, timestamp, action details, and originating system. Files accessed by users must be monitored, especially sensitive or regulated files, to detect misuse. Logs must be centrally stored, protected from tampering, and retained per policy. This provides an auditable trail for investigations 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

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 |  |
| 2.0 | 07/20/2025 | Added Module 3 milestone content (first 10 coding standards with examples) | John Munguia |  |
| 3.0 | 08/15/2025 | Completed Project One with principles, threat levels, automation, encryption, Triple-A, and summary tables | John Munguia |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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