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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_heading=h.y1c3jlxgfp9y)

[Purpose 2](#_heading=h.euhivy1zsxw2)

[Scope 2](#_heading=h.qz4qjccfwgj8)

[Module Three Milestone 2](#_heading=h.zhdcrpfe5t0i)

[Ten Core Security Principles 2](#_heading=h.z5ym79rjluxn)

[C/C++ Ten Coding Standards 3](#_heading=h.fhqnzyfc9eu5)

[Coding Standard 1 4](#_heading=h.i4x9bk7cwcfe)

[Coding Standard 2 5](#_heading=h.2xbyn0582kig)

[Coding Standard 3 6](#_heading=h.seqgblf4axfq)

[Coding Standard 4 7](#_heading=h.gfue85aiicqn)

[Coding Standard 5 8](#_heading=h.o0g0c2h4l5pa)

[Coding Standard 6 9](#_heading=h.gu0jwbyypixv)

[Coding Standard 7 10](#_heading=h.nwo71xc88dc8)

[Coding Standard 8 11](#_heading=h.65zpyxpdliej)

[Coding Standard 9 13](#_heading=h.baw9xdabbfl3)

[Coding Standard 10 14](#_heading=h.169h0a597fuv)

[Defense-in-Depth Illustration 15](#_heading=h.m1n06tcassrx)

[Project One 15](#_heading=h.ca1kc7l1cqms)

[1.](#_heading=h.wx1exm1afd52) Revise the C/C++ Standards 15

[2.](#_heading=h.lcvcdedzqpvi) Risk Assessment 15

[3.](#_heading=h.wpd90nry7hes) Automated Detection 15

[4.](#_heading=h.ueglrgvuuj3) Automation 15

[5.](#_heading=h.y3nrn95pkzu4) Summary of Risk Assessments 16

[6.](#_heading=h.554s0wsstvhy) Create Policies for Encryption and Triple A 16

[7.](#_heading=h.4ayyde5ddz1x) Map the Principles 17

[Audit Controls and Management 18](#_heading=h.8iieaz27f609)

[Enforcement 18](#_heading=h.ap234lwqggbz)

[Exceptions Process 18](#_heading=h.od66yx6kq36o)

[Distribution 19](#_heading=h.gtjrtdox858r)

[Policy Change Control 19](#_heading=h.t8elwbzekx92)

[Policy Version History 19](#_heading=h.pslzu61fwrn)

[Appendix A Lookups 19](#_heading=h.ogajsygy6tvy)

[Approved C/C++ Language Acronyms 19](#_heading=h.uwbl3h1nh63)

## 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 verify that every piece of data entering your system meets strict expectations: type, length, format and range before it is used. Proper validation prevents malformed or malicious inputs (e.g. SQL injection, buffer overflows) from ever reaching sensitive code paths. |
| 1. Heed Compiler Warnings | Treat compiler warnings as bugs. Enable the highest warning levels and address every warning promptly because they often flag questionable constructs such as unused variables, implicit conversions and deprecated APIs that can lead to subtle vulnerabilities if ignored. |
| 1. Architect and Design for Security Policies | Embed security considerations into your system’s architecture from the start. Define clear policies for authentication, authorization and data flow and use threat modeling to shape module boundaries, interfaces and trust levels so that security is a foundational aspect not an afterthought. |
| 1. Keep It Simple | Strive for minimalism in design and implementation. Simpler code has fewer execution paths and interactions which reduces the attack surface, makes reviews and audits more effective and lowers the chance of introducing hidden bugs. |
| 1. Default Deny | Deny all actions by default then explicitly grant only the permissions and inputs you require. Whether configuring firewalls, file access or API endpoints, a fail safe default posture ensures that unexpected or unauthorized operations are blocked. |
| 1. Adhere to the Principle of Least Privilege | Grant each user, process and component only the absolute minimum rights needed to perform its function. Limiting privileges confines the damage any compromised element can inflict and helps to contain breaches. |
| 1. Sanitize Data Sent to Other Systems | Before exporting or transmitting data in logs, over the network or to other services, cleanse and encode it appropriately such as HTML encode or escape SQL meta characters. This prevents downstream injection attacks and ensures that data is interpreted safely. |
| 1. Practice Defense in Depth | Layer multiple independent security controls such as input validation, authentication, encryption and logging so that if one control fails or is bypassed others remain to protect critical assets. Overlapping defenses make successful attacks far more difficult. |
| 1. Use Effective Quality Assurance Techniques | Employ rigorous QA processes including automated static analysis, dynamic testing, fuzzing, code reviews and security penetration testing to catch and remediate flaws early. A strong QA discipline turns code verification into a continuous, repeatable safeguard. |
| 1. Adopt a Secure Coding Standard | Follow an established peer reviewed coding standard such as SEI CERT C++ to guide consistent security focused development practices. Standards codify best practices, reduce individual judgment calls and ensure that every team member applies the same rigor. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Using platform-dependent types such as int or long can lead to inconsistent behavior—overflow, truncation, or sign-extension differences—when compiled on different architectures. By adopting fixed-width integer types from <cstdint> (e.g. int32\_t, uint64\_t), you explicitly document your size and range assumptions, improving portability and reducing subtle bugs. |

| **Noncompliant Code** |
| --- |
| Declares variables with generic integer types whose actual width varies by platform, risking overflow or truncation on targets where int is smaller than expected. |
| #include <iostream>  void processData() {  int count = 1000000000; // ‘int’ may be 16- or 32-bit depending on platform  long totalItems = count; // size of long also platform-dependent  std::cout << "Total items: " << totalItems << std::endl;  } |

| **Compliant Code** |
| --- |
| Uses fixed-width integer types to guarantee consistent size and behavior across all platforms. |
| #include <cstdint>  #include <iostream>  void processData() {  int32\_t count = INT32\_C(1000000000);  int64\_t totalItems = static\_cast<int64\_t>(count);  std::cout << "Total items: " << totalItems << std::endl;  } |

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

| **Principles(s):** Validate Input Data (Principle 1): Mandates checking or sanitizing every external input to ensure it meets expected formats and ranges before use.  Default Deny (Principle 5): Any data that fails validation is automatically rejected, preventing unexpected or malicious values from propagating.  Adopt a Secure Coding Standard (Principle 10): Embeds input validation as a mandatory, project-wide rule, ensuring consistency across all modules. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity Scan | 2021.12 | CWE\_20 | Flags missing input-validation (Improper Input Validation). |
| SonarQube C++ | 8.9 | cpp:S5131 | Detects uses of untrusted data without prior validation. |
| Fortify SCA | 2020.2 | invalid-input | Identifies external data not validated or sanitized. |
| CodeQL | N/A | cpp/input-validation | Queries for missing sanitization of user-controlled data. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Values supplied to variables, array indices, loop counters, or API calls must be constrained to expected ranges or domains. Failing to validate or constrain data values can lead to out-of-bounds accesses, integer overflows, resource exhaustion, or logic errors. By enforcing explicit range checks, using strongly typed enums instead of magic numbers, and handling out-of-range conditions, you ensure inputs remain within safe, predictable limits and prevent undefined or malicious behavior. |

| **Noncompliant Code** |
| --- |
| Uses an unchecked integer as an array index, permitting out-of-bounds writes when index is negative or ≥ 10. |
| #include <iostream>  void storeValue(int index, int value) {  int buffer[10] = {0};  // No validation of 'index' – may write past buffer bounds  buffer[index] = value;  std::cout << "Stored at " << index << std::endl;  } |

| **Compliant Code** |
| --- |
| Validates index against the array’s bounds and reports an error if it falls outside the valid range. |
| #include <iostream>  #include <stdexcept>  void storeValue(int index, int value) {  constexpr int SIZE = 10;  int buffer[SIZE] = {0};  if (index < 0 || index >= SIZE) {  throw std::out\_of\_range("Index " + std::to\_string(index) + " is out of valid range [0, " + std::to\_string(SIZE-1) + "]");  }  buffer[index] = value;  std::cout << "Stored at " << index << std::endl;  } |

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

| **Principles(s):** Validate Input Data (Principle 1): Ensures every index or offset is checked against container bounds before use.  Default Deny (Principle 5): Rejects any access attempt outside the valid index range.  Adopt a Secure Coding Standard (Principle 10): Embeds STR53-CPP as a mandatory rule across the codebase. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 12.0.0 | bugprone-bounds-pointer-arithmetic | Flags pointer/index arithmetic without prior bounds checks. |
| Coverity Scan | 2021.12 | CERT\_STR53\_cpp | Detects unchecked container or array element access. |
| PVS-Studio | 7.20 | V547 | Warns on potential out-of-bounds array indexing. |
| SonarQube C++ | 8.9 | cpp:S2251 | Identifies uses of operator[] on containers without range checks. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Improper handling of C-style strings (raw character arrays) often leads to buffer overflows, missing null terminators, or encoding errors. By using safe string functions with explicit length checks (e.g. strncpy, snprintf) or preferring C++ abstractions (std::string, std::string\_view), you ensure that all operations respect buffer boundaries, correctly terminate strings, and avoid undefined behavior. |

| **Noncompliant Code** |
| --- |
| Copies user input into a fixed-size buffer with strcpy, which does not check length and can overflow if src is too large. |
| #include <iostream>  #include <cstring>  void echoInput(const char\* src) {  char buffer[32];  // No bounds checking – if src ≥ 32 bytes, this overflows buffer  std::strcpy(buffer, src);  std::cout << buffer << std::endl;  } |

| **Compliant Code** |
| --- |
| Uses strncpy with an explicit maximum length and ensures null termination. Alternatively, uses std::string to manage dynamic sizing safely. |
| #include <iostream>  #include <cstring>  #include <string>  void echoInputSafe(const char\* src) {  char buffer[32];  // Copy up to buffer-1 characters and null-terminate  std::strncpy(buffer, src, sizeof(buffer) - 1);  buffer[sizeof(buffer) - 1] = '\0';  std::cout << buffer << std::endl;  }  void echoInputCpp(const std::string& src) {  // std::string manages allocation and null-termination automatically  std::string safeCopy = src;  std::cout << safeCopy << std::endl;  } |

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

| **Principles(s):** Manage Resources Securely (Principle 2): Ensures that every heap allocation or handle acquisition is paired with automatic release, preventing leaks and dangling references.  Fail Securely (Principle 7): Guarantees that on errors or exceptions, resources are still properly freed, avoiding inconsistent program state.  Adopt a Secure Coding Standard (Principle 10): Embeds RAII and smart-pointer usage as a project-wide requirement. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 13.0.0 | modernize-make-unique | Suggests replacing raw new/delete with std::make\_unique. |
| Coverity Scan | 2021.12 | RESOURCE\_LEAK | Detects allocations that lack corresponding deallocations. |
| PVS-Studio | 7.20 | V595 | Flags potential memory or handle leaks in various code paths. |
| SonarQube C++ | 8.9 | cpp:S2092 | Identifies resources (memory, files, sockets) not properly freed. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Constructing SQL commands by concatenating user-supplied strings allows attackers to inject malicious SQL that can read, modify, or delete data. By using parameterized queries (prepared statements), stored procedures, or ORM frameworks that escape inputs, you separate code from data and ensure that inputs cannot change the intended SQL structure. Additionally, validating and limiting user inputs—even when using parameterization—further hardens your application against injection attacks. |

| **Noncompliant Code** |
| --- |
| Builds a SQL query by directly embedding untrusted inputs, allowing an attacker to terminate the string and append arbitrary SQL. |
| #include <iostream>  #include <string>  #include <sqlite3.h>  bool authenticate(sqlite3\* db, const std::string& username, const std::string& password) {  // Direct concatenation of user inputs into SQL  std::string sql =  "SELECT COUNT(\*) FROM users WHERE username = '" + username +  "' AND password = '" + password + "';";  sqlite3\_stmt\* stmt = nullptr;  if (sqlite3\_prepare\_v2(db, sql.c\_str(), -1, &stmt, nullptr) != SQLITE\_OK) {  return false;  }  bool ok = (sqlite3\_step(stmt) == SQLITE\_ROW && sqlite3\_column\_int(stmt, 0) == 1);  sqlite3\_finalize(stmt);  return ok;  } |

| **Compliant Code** |
| --- |
| Uses a prepared statement with bound parameters, so input values cannot alter the SQL command’s structure. |
| #include <iostream>  #include <string>  #include <sqlite3.h>  bool authenticate(sqlite3\* db, const std::string& username, const std::string& password) {  sqlite3\_stmt\* stmt = nullptr;  constexpr char QUERY[] =  "SELECT COUNT(\*) "  "FROM users "  "WHERE username = ?1 AND password = ?2;";  // Prepare once, then bind parameters  if (sqlite3\_prepare\_v2(db, QUERY, -1, &stmt, nullptr) != SQLITE\_OK) {  return false;  }  // Bind user inputs safely  sqlite3\_bind\_text(stmt, 1, username.c\_str(), -1, SQLITE\_TRANSIENT);  sqlite3\_bind\_text(stmt, 2, password.c\_str(), -1, SQLITE\_TRANSIENT);  bool ok = false;  if (sqlite3\_step(stmt) == SQLITE\_ROW) {  int count = sqlite3\_column\_int(stmt, 0);  ok = (count == 1);  }  sqlite3\_finalize(stmt);  return ok;  } |

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

| **Principles(s):** Validate Input Data (Principle 1): Verify that all integer operands and results stay within safe, expected bounds.  Fail Securely (Principle 7): Detect and handle any overflow conditions instead of allowing wrap-around or corrupted values.  Adopt a Secure Coding Standard (Principle 10): Enforce the STR34-CPP rule across the codebase to guard against unchecked integer arithmetic. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 12.0.0 | bugprone-integer-overflow | Flags arithmetic expressions that may overflow built-in integer types. |
| Coverity Scan | 2021.12 | CWE\_190 | Detects integer operations without explicit overflow checks. |
| PVS-Studio | 7.20 | V560 | Warns on potential integer-overflow risks in expressions. |
| SonarQube C++ | 8.9 | cpp:S1935 | Identifies integer operations whose results may exceed type limits. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Out-of-bounds memory accesses, use-after-free errors, and unchecked dynamic allocations can corrupt memory or crash applications, opening the door to exploits such as buffer overflows. By validating every size or index against the actual allocation, checking the results of allocation calls, and favoring RAII and standard containers over raw pointers, you enforce strict memory safety and reduce undefined behavior. |

| **Noncompliant Code** |
| --- |
| Copies length bytes into a fixed‐size heap buffer without verifying that length is within bounds, risking overflow and memory corruption. |
| #include <cstdlib>  #include <cstring>  #include <iostream>  void copyData(const char\* src, size\_t length) {  // Allocate 64 bytes but never check that length <= 64  char\* buffer = static\_cast<char\*>(std::malloc(64));  std::memcpy(buffer, src, length);  // Potential overflow if length > 64  std::cout << buffer << std::endl;  std::free(buffer);  } |

| **Compliant Code** |
| --- |
| Ensures length does not exceed the buffer capacity and uses std::vector<char> for automatic lifetime and bounds-checked access when possible. |
| #include <vector>  #include <cstring>  #include <iostream>  #include <stdexcept>  void copyDataSafe(const char\* src, size\_t length) {  constexpr size\_t CAPACITY = 64;  if (length > CAPACITY) {  throw std::length\_error("Copy length exceeds buffer capacity");  }  // std::vector manages allocation and deallocation  std::vector<char> buffer(CAPACITY);  std::memcpy(buffer.data(), src, length);  buffer[length] = '\0'; // ensure null termination  std::cout << buffer.data() << std::endl;  } |

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

| **Principles(s):** Validate Input Data (Principle 1): Ensure that any length or size arguments passed to memory operations are checked to prevent buffer overflows  .  Keep It Simple (Principle 4): Favor simple, robust abstractions (e.g. std::vector) over manual memory management to reduce error-prone code  .  Adopt a Secure Coding Standard (Principle 10): Embed safe-memory rules into your coding policy so every developer applies the same practices  . |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 12.0.0 | hicpp-no-malloc | Flags use of low-level allocation functions (malloc/free), recommending safer abstractions like std::vector . |
| Coverity Scan | 2021.12 | CWE\_119 | Detects potential buffer-overflow and memory-corruption issues in copy operations. |
| PVS-Studio | 7.20 | V575 | Warns on suspicious use of memcpy/memmove that may overflow the destination buffer. |
| SonarQube C++ | 8.9 | cpp:S2639 | Identifies unsafe low-level memory patterns prone to buffer overflows. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Use assertions to document and enforce internal assumptions (preconditions, postconditions, and invariants) during development. Assertions help catch programmer errors early by halting execution when an assumption is violated. They are not a substitute for proper error handling of external inputs and may be disabled in release builds, so they should only verify conditions that should never occur in correct code |

| **Noncompliant Code** |
| --- |
| No check is performed on index, so out-of-range access leads to undefined behavior without a clear diagnostic. |
| #include <vector>  int getValue(const std::vector<int>& v, size\_t index) {  // No assertion – accessing v[index] may be out of bounds  return v[index];  } |

| **Compliant Code** |
| --- |
| An assert documents the invariant that index must be less than v.size(). If violated during testing, the program halts with a clear error. |
| #include <cassert>  #include <vector>  int getValue(const std::vector<int>& v, size\_t index) {  assert(index < v.size()); // verifies internal assumption  return v[index];  } |

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

| **Principles(s):**Use Effective Quality Assurance Techniques (Principle 9): Assertions catch violations of internal assumptions early during testing, making logic errors visible before release.  Keep It Simple (Principle 4): A single assert statement provides a clear, concise check of an invariant without complex error-handling code.  Adopt a Secure Coding Standard (Principle 10): Mandating assertion usage ensures all developers consistently document and verify preconditions and invariants. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 14.0.0 | misc-assert | Flags functions lacking assert checks for documented invariants. |
| PVS-Studio | 7.20 | V597 | Warns when expected preconditions are not validated via assertions. |
| SonarQube C++ | 8.9 | cpp:S1787 | Recommends using assertions to verify critical assumptions. |
| CodeQL | N/A | cpp/assert-used | Queries for presence of std::assert calls to enforce precondition checks. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Improper exception handling can lead to resource leaks, application crashes, or inadvertent information disclosure. By throwing exceptions only for truly exceptional conditions, catching them by const reference to avoid object slicing, handling specific exception types, and ensuring destructors are noexcept, you maintain clear error propagation and strong exception-safety guarantees |

| **Noncompliant Code** |
| --- |
| Throws and catches by value (slicing derived exceptions), uses a catch-all that obscures failures, and may allow resource leaks or hidden bugs. |
| #include <iostream>  #include <stdexcept>  #include <vector>  int getElement(const std::vector<int>& v, size\_t idx) {  if (idx >= v.size()) {  throw std::out\_of\_range("Index out of range");  }  return v[idx];  }  int main() {  std::vector<int> data = {1, 2, 3};  try {  int val = getElement(data, 5);  std::cout << val << std::endl;  } catch (std::exception e) { // caught by value → slicing  std::cout << "Error: " << e.what() << std::endl;  } catch (...) { // no information about the error  std::cout << "Unknown error occurred\n";  }  return 0;  } |

| **Compliant Code** |
| --- |
| Throws only for unexpected conditions, catches by const reference to preserve full exception type and message, and handles specific errors cleanly. |
| #include <iostream>  #include <stdexcept>  #include <vector>  int getElement(const std::vector<int>& v, size\_t idx) {  if (idx >= v.size()) {  throw std::out\_of\_range("Index out of range");  }  return v[idx];  }  int main() {  std::vector<int> data = {1, 2, 3};  try {  int val = getElement(data, 5);  std::cout << val << std::endl;  } catch (const std::out\_of\_range& e) { // preserves exception type  std::cerr << "Range error: " << e.what() << std::endl;  } catch (const std::exception& e) { // handles other std::exceptions  std::cerr << "Unexpected error: " << e.what() << std::endl;  }  return 0;  } |

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

| **Principles(s):** Manage Resources Securely (Principle 2): Ensures that throwing or catching exceptions cannot leak resources by relying on RAII and making destructors noexcept.  Fail Securely (Principle 7): Guarantees that error conditions propagate safely and leave the program in a consistent state, rather than hiding failures or crashing unpredictably.  Adopt a Secure Coding Standard (Principle 10): Mandates a consistent, project-wide approach to exception handling—throw only on true errors, catch by const reference, and avoid catch-all handlers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 14.0.0 | hicpp-exception-slice | Warns when exceptions are caught by value (exception slicing) instead of by reference. |
| Coverity Scan | 2021.12 | CERT\_EXC36\_cpp | Detects catch blocks that take exceptions by value rather than by (const) reference. |
| PVS-Studio | 7.20 | V702 | Flags catch(...) or catch (std::exception e) patterns that slice or obscure exception details. |
| SonarQube C++ | 8.9 | cpp:S3440 | Identifies catch clauses that should catch exceptions by (const) reference to preserve full type and data. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Concurrency Safety | STD-008-CPP | Unsynchronized access to shared data in multithreaded C++ programs leads to race conditions, data corruption, and unpredictable behavior. By using standard concurrency primitives—such as std::mutex, std::lock\_guard, and atomic types (std::atomic)—you enforce proper synchronization, ensuring that only one thread at a time can modify critical resources and that reads see a consistent state. |

| **Noncompliant Code** |
| --- |
| Two threads increment a shared counter without any synchronization. This can produce lost updates or corrupted state. |
| #include <thread>  #include <vector>  #include <iostream>  int counter = 0;  void increment() {  for (int i = 0; i < 100000; ++i) {  ++counter; // no synchronization → data race  }  }  int main() {  std::vector<std::thread> threads;  for (int i = 0; i < 4; ++i) {  threads.emplace\_back(increment);  }  for (auto& t : threads) {  t.join();  }  std::cout << "Final counter: " << counter << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| Protects the shared counter with a std::mutex and uses std::lock\_guard to ensure mutual exclusion. |
| #include <thread>  #include <vector>  #include <iostream>  #include <mutex>  int counter = 0;  std::mutex counterMutex;  void increment() {  for (int i = 0; i < 100000; ++i) {  std::lock\_guard<std::mutex> lock(counterMutex);  ++counter; // synchronized access  }  }  int main() {  std::vector<std::thread> threads;  for (int i = 0; i < 4; ++i) {  threads.emplace\_back(increment);  }  for (auto& t : threads) {  t.join();  }  std::cout << "Final counter: " << counter << std::endl;  return 0;  } |

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

| **Principles(s):** Manage Resources Securely (Principle 2): Use RAII (e.g. std::lock\_guard) for all mutexes so that locks are always released, even on exceptions.  Fail Securely (Principle 7): Ensure that, when an exception or error occurs in a synchronized section, locks are properly released and shared data remains in a consistent state.  Adopt a Secure Coding Standard (Principle 10): Codify the use of C++11 concurrency primitives (std::mutex, std::atomic, etc.) as a mandatory, project-wide practice |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang Static Analyzer | 12.0.0 | thread-safety | Static analysis using lock annotations (e.g. GUARDED\_BY) to flag unsynchronized access. |
| Thread Sanitizer (TSan) | Bundled | TSan | Dynamic race detector that reports data races at runtime. |
| Coverity Scan | 2021.12 | CONCURRENCY\_UNSAFE | Flags potential race conditions and unsynchronized accesses to shared variables. |
| SonarQube C++ | 8.9 | cpp:S2695 | Detects shared-data access without appropriate synchronization mechanisms. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secure Logging | STD-009-CPP | Logs are critical for auditing and incident response, but naïve logging of untrusted data can enable log injection, reveal sensitive information, or corrupt log integrity. By using a dedicated, thread-safe logging library and sanitizing or encoding any user-supplied content before writing to logs, you ensure that log entries remain trustworthy, cannot be forged via newline or control-character injection, and do not leak confidential data |

| **Noncompliant Code** |
| --- |
| Appends raw user input directly to a plain text log file. An attacker could embed newline (\n) or carriage-return (\r) characters to forge additional log entries or corrupt the file. |
| #include <fstream>  #include <string>  void logUserAction(const std::string& userInput) {  std::ofstream log("app.log", std::ios::app);  // Unsafe: raw input may contain CR/LF to inject fake entries  log << "User input: " << userInput << std::endl;  } |

| **Compliant Code** |
| --- |
| Removes any carriage-return or newline characters from the input, then uses a structured, thread-safe logger API to format and rotate log files. This prevents injection and preserves integrity. |
| #include <regex>  #include <string>  #include <iostream>  // Simple sanitizer: strip CR and LF before logging  static std::string sanitize(const std::string& in) {  return std::regex\_replace(in, std::regex(R"([\r\n])"), "");  }  void logUserAction(const std::string& userInput) {  static std::mutex logMutex;  std::lock\_guard<std::mutex> lock(logMutex);  std::ofstream log("app.log", std::ios::app);  if (!log) {  // handle open-failure…  return;  }  log << "User input: " << sanitize(userInput) << '\n';  } |

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

| **Principles(s):**Sanitize Data Sent to Other Systems (Principle 7): Clean or encode all user-supplied log inputs to prevent newline/CRLF injection and ensure log integrity  .  Adhere to the Principle of Least Privilege (Principle 6): Restrict which components and users may write to or read from log files, minimizing exposure of sensitive data.  Architect and Design for Security Policies (Principle 3): Build secure, tamper-evident audit trails into your system design so logging is a first-class, non-optional feature. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube C++ | 8.9 | cpp:S5847 | Flags log calls that directly include untrusted data without prior sanitization. |
| clang-tidy | 13.0.0 | misc-argument-comment | (Custom) Warns on stream-insertion of variables into log streams without sanitization. |
| CodeQL | n/a | cpp/log-injection | Queries for concatenation or interpolation of user input into logging APIs. |
| Coverity Scan | 2021.12 | CERT\_SEC30\_CPP | Detects patterns where log statements expose or allow injection of untrusted input. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Uncontrolled Format String | STD-010-CPP | Functions like printf, fprintf, and snprintf interpret format specifiers in their first argument. If an attacker can influence that argument, they can read from or write to arbitrary memory locations, leading to information disclosure or code execution. By fixing the format string and passing untrusted data only as parameters—never as the format—you eliminate this class of vulnerability. |

| **Noncompliant Code** |
| --- |
| Passes user-controlled input directly as the format string, allowing specifiers like %x or %n to leak or overwrite memory. |
| #include <cstdio>  void logMessage(const char\* userInput) {  // Unsafe: userInput may contain format specifiers  std::printf(userInput);  } |

| **Compliant Code** |
| --- |
| Uses a fixed format string and passes untrusted data as an argument, so any percent signs in userInput are printed literally rather than interpreted. |
| #include <cstdio>  void logMessage(const char\* userInput) {  // Safe: format string is constant, userInput passed separately  std::printf("%s\n", userInput);  } |

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

| **Principles(s):** Validate Input Data (Principle 1): Ensure that any format string used in printf-style calls is a constant or explicitly validated, so untrusted data cannot be interpreted as format specifiers  Keep It Simple (Principle 4): Rely on fixed format strings and pass user data as separate parameters, minimizing complexity and the risk of inadvertent parsing of malicious input  Adopt a Secure Coding Standard (Principle 10): Mandate, across the codebase, that all printf-family calls use literal format strings, preventing format-string vulnerabilities by design |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 12.0.0 | bugprone-format-security | Flags printf-family calls where the format argument is not a compile-time literal. |
| Coverity Scan | 2021.12 | CWE\_134 | Detects uncontrolled format string usage in calls like printf, sprintf, etc. |
| PVS-Studio | 7.20 | V598 | Warns when a non-literal string is passed as the format argument to printf-style APIs. |
| SonarQube C++ | 8.9 | cpp:S5847 | Identifies dynamic or unvalidated format strings in logging and I/O functions. |

### 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.

**Pre-production**

1. **Design**
   * **Policy-as-Code:** Capture encryption (TLS 1.3 in-transit, AES-256 at-rest) and IAM rules in your architecture blueprints (e.g. Terraform modules, Open Policy Agent).
   * **Security-Driven Design:** Use automated checks (e.g. OPA policy tests) against your design templates to confirm every data flow is encrypted and every service has a least-privilege role.
2. **Assess & Plan**
   * **Threat Modeling Automation:** Run scripted threat-model analyses on your Infrastructure as Code to flag any unencrypted paths or missing RBAC controls.
   * **Backlog Prioritization:** Feed identified gaps (e.g. insecure endpoints) directly into your sprint backlog via the pipeline, ranked by impact and compliance deadlines.
3. **Build**
   * **Secure Builds:** CI pipelines automatically provision KMS-managed keys and certificates, enforce IAM role attachments, and pull only signed dependencies from trusted registries.
   * **Static Scans & Secret Detection:** Tools like SonarQube and CodeQL scan every commit for hard-coded credentials or deprecated crypto APIs, failing the build on violations.
4. **Verify & Test**
   * **Automated Compliance Tests:** Unit and integration suites confirm that all service endpoints negotiate TLS and that storage volumes are encrypted.
   * **Dynamic Security Scans:** Run tools such as OWASP ZAP or Tenable Nessus against staging to validate no clear-text channels or broken auth flows remain.

**Production** 5. **Transition & Health Check**

* **Config Drift Detection:** Scheduled jobs compare live configs against the golden IaC baseline, alerting on any disabled encryption or changed IAM policies.
* **Penetration Test Orchestration:** Automate deployment of pen-test agents and collect results in your vulnerability management system before go-live.

1. **Monitor & Detect**
   * **SIEM-Driven Logging:** Centralize logs of TLS handshake failures, authentication errors, and unauthorized access attempts in your SIEM for real-time alerting.
   * **Behavioral Analytics:** Use automated anomaly detection to spot spikes in failed logins or abnormal certificate-negotiation patterns.
2. **Respond**
   * **Automated Remediation:** On detection of a compromised key or misconfiguration, workflows revoke affected credentials, rotate certificates, and invalidate sessions.
   * **Incident Playbooks:** Tools like StackStorm execute predefined containment steps (e.g. isolate instances, block offending IPs) before raising tickets for on-call teams.
3. **Maintain & Stabilize**
   * **Key & Certificate Rotation:** Cron-driven automation rotates encryption keys and SSL/TLS certificates on a regular cadence without downtime.
   * **Continuous Compliance Audits:** Nightly “compliance-as-code” jobs verify that all encryption and AAA policies remain enforced, feeding metrics to your security dashboard.

### 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 | High | Low | High | 4 |
| STD-002-CPP | High | Medium | Low | High | 4 |
| STD-003-CPP | Medium | Medium | Low | Medium | 3 |
| STD-004-CPP | High | Medium | Low | High | 4 |
| STD-005-CPP | High | Medium | Low | High | 4 |
| STD-006-CPP | Medium | Medium | Low | Medium | 3 |
| STD-007-CPP | Medium | Medium | Low | Medium | 3 |
| STD-008-CPP | High | Medium | Low | High | 4 |
| STD-009-CPP | Medium | High | Low | High | 4 |
| STD-010-CPP | High | Medium | Low | 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 | All stored data—databases, file shares, backups, and object storage—must use FIPS-approved AES-256 encryption. Keys are centrally managed in the corporate KMS and rotated every 90 days. If media or backups are lost, data remains unreadable. |
| Encryption in flight | All network traffic—client-to-server, service-to-service, and API calls—must use TLS 1.3 with strong cipher suites and certificate validation. Internal microservices require mutual TLS. This ensures data stays confidential and intact over untrusted networks. |
| Encryption in use | Sensitive data (PII, credentials, keys) in memory must run only in secure, access-controlled enclaves or protected heaps. When available, hardware memory encryption (e.g. Intel SGX, AMD SEV) must be used for key operations, preventing extraction of secrets from live processes. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | All users and services must log in via corporate SSO with MFA (at least TOTP plus password). Service accounts require certificates or short-lived tokens (< 1 hour). Plaintext credentials are not allowed.[Insert text.] |
| Authorization | We enforce RBAC with least-privilege: users, services, and processes get only the permissions they need. Access is reviewed quarterly, and any privilege escalation requires multi-level approval. |
| Accounting | We log all security events—user logins, admin actions (DB changes, user creation), privilege escalations, and file access—to a central SIEM. Logs are immutable and retained for at least 365 days for forensics. |

**\***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 |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [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 |