**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 | All input coming into a program should be treated as untrusted until proven otherwise. Attackers often exploit weak input handling to inject malicious commands or overflow buffers. By validating length, type, and format of input data before using it, developers reduce the chance of bugs, crashes, or security breaches. A strong validation layer can stop harmful input long before it reaches critical parts of t==he application. |
| 1. Heed Compiler Warnings | Compilers often detect unsafe practices, such as mismatched types, uninitialized variables, or dangerous function calls. Ignoring these warnings allows small mistakes to slip into production code, where they may become exploitable vulnerabilities. Developers should enable and review all warnings, and treat them as errors when possible, to ensure that questionable code is fixed early in the development cycle. |
| 1. Architect and Design for Security Policies | Security must be considered at the design stage, not added later as an afterthought. This means building the system around organizational security policies such as authentication rules, encryption requirements, and audit logging. By doing this, developers make sure the system enforces security by design, rather than relying on patches or ad-hoc fixes that can leave gaps. |
| 1. Keep It Simple | Complex systems are harder to secure because they have more moving parts and more opportunities for errors. Simplicity in code, architecture, and logic makes software easier to review, test, and maintain. When a feature or design is overly complicated, it becomes difficult to predict how it will behave under attack. Following this principle means choosing clarity and simplicity to reduce risk. |
| 1. Default Deny | When access is not explicitly allowed, it should be denied. A default-deny stance ensures that programs don’t accidentally allow unexpected behavior or unauthorized users. For example, firewalls and access control lists that default to blocking everything are safer than those that default to allowing everything. |
| 1. Adhere to the Principle of Least Privilege | Every user, process, or system component should run with the smallest set of permissions needed to do its job. This limits the damage if the account or process is compromised. For example, a program that only needs to read data should not have write or administrative privileges. By applying least privilege, attackers who gain access face more restrictions, which slows or stops their ability to escalate attacks. |
| 1. Sanitize Data Sent to Other Systems | Applications frequently pass data to other systems such as databases, logs, or external services. If this data is not sanitized, it may be interpreted as commands, leading to attacks such as SQL injection or log poisoning. Sanitizing ensures that outgoing data is stripped of harmful characters and encoded safely so it cannot change how the receiving system behaves. |
| 1. Practice Defense in Depth | No single layer of security is perfect, so multiple layers are needed. For example, input validation, strong authentication, error handling, and intrusion detection systems together create overlapping defenses. Even if an attacker bypasses one safeguard, others remain in place to stop them. Defense in depth accepts that some controls may fail, but ensures there is always another barrier. |
| 1. Use Effective Quality Assurance Techniques | Security and quality go hand in hand. Testing, peer reviews, static analysis, and automated scanning help catch flaws before release. Quality assurance is not just about finding bugs that cause crashes, but also about spotting weaknesses that could become vulnerabilities. By applying effective QA techniques throughout development, teams can deliver safer and more reliable code. |
| 1. Adopt a Secure Coding Standard | A secure coding standard provides developers with clear rules on how to write safe code, avoid known pitfalls, and handle common problems such as memory management and input validation. Following a standard ensures consistency across a team and helps less-experienced developers avoid mistakes. Secure coding standards, such as the SEI CERT C++ guidelines, turn best practices into enforceable habits that make the entire codebase more secure. |

### 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 Appropriate Data Types to Prevent Overflow/Underflow**  Choosing the right data type ensures that variables can hold the intended range of values without causing errors. Using a type that is too small can lead to overflows or underflows, which attackers can exploit to cause unexpected behavior or gain control of the system. Using precise and secure data types also makes code clearer and reduces logic errors. |

| **Noncompliant Code** |
| --- |
| This code uses short for a value that may exceed its maximum range, causing overflow. |
| short balance = 32000;  short deposit = 1000;  short total = balance + deposit; // Overflow happens here  std::cout << "Total: " << total << std::endl; |

| **Compliant Code** |
| --- |
| This code uses a larger data type (int or long) that safely holds the result without overflow. |
| int balance = 32000;  int deposit = 1000;  int total = balance + deposit; // No overflow  std::cout << "Total: " << total << std::endl; |

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

| **Principle: Integrity**  This coding standard supports the principle of *Integrity* by ensuring that data remains accurate and uncorrupted throughout arithmetic operations. Preventing integer overflow or underflow maintains the reliability and consistency of program calculations, which are essential to preserving system integrity and preventing tampering or corruption. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12 | integerOverflow, cert-int30-c | Flags integer overflows/underflows and risky integer conversions per CERT INT30-C. |
| Clang-Tidy | 17.0 | **cppcoreguidelines-narrowing-conversions** | Warns on narrowing casts that can truncate/overflow; good for mixing int/short/size\_t. |
| MSVC Code Analysis | VS 2022 | C26451 | “Arithmetic overflow” rule: detects operations that might overflow (e.g., mixing signed/unsigned). |
| GitHub CodeQL | latest | cpp/integer-overflow | CI security scan for arithmetic overflows in PRs/repos. *(If you don’t use CodeQL, you can omit this row.)* |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | **Validate and Constrain Data Values Before Use**  Values derived from users, files, networks, or other untrusted sources must be checked for range, type, and invariants before use. Without validation, out-of-range values can cause arithmetic errors, out-of-bounds access, or logic bypasses that attackers can exploit. |

| **Noncompliant Code** |
| --- |
| Parses a user string and uses it directly as an array index with no range checks, allowing negative or too-large indices. |
| #include <vector>  #include <string>  std::vector<int> scores(10, 0);  std::string userInput; // e.g., "-3" or "999"  std::getline(std::cin, userInput);  int idx = std::stoi(userInput); // may throw, and may be out of range  scores[idx] = 42; // no bounds check — UB / crash |

| **Compliant Code** |
| --- |
| Safely parses the input, rejects non-numeric values, and enforces [0, scores.size()-1] bounds before indexing. |
| #include <vector>  #include <string>  #include <limits>  #include <iostream>  std::vector<int> scores(10, 0);  std::string userInput;  std::getline(std::cin, userInput);  long idxLong;  try {  size\_t pos = 0;  idxLong = std::stol(userInput, &pos, 10);  if (pos != userInput.size()) throw std::invalid\_argument("trailing chars");  } catch (...) {  std::cerr << "Invalid index.\n";  // handle / return  }  if (idxLong < 0 || idxLong >= static\_cast<long>(scores.size())) {  std::cerr << "Index out of range.\n";  // handle / return  } else {  scores[static\_cast<size\_t>(idxLong)] = 42; // safe  } |

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

| Principle(s): Integrity and Defense in Depth This standard supports the principles of *Integrity* and *Defense in Depth*. Validating and constraining data ensures that values entering the system remain within safe, expected ranges, maintaining the accuracy and consistency of the application’s data. By layering validation across user input, file reads, and network data, developers prevent corrupted or malicious values from propagating deeper into the system, forming multiple lines of defense. These safeguards help ensure that one weak point, such as missing input validation, doesn’t compromise the rest of the application. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Low | P1 | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12 | uninitvar, outOfBounds, cert-str34-c | Detects uninitialized or out-of-bounds variable usage, and enforces CERT rules for validated input. |
| Clang-Tidy | 17.0 | cppcoreguidelines-pro-bounds-array-to-pointer-decay, clang-analyzer-core.NullDereference | Warns about unsafe array indexing and potential null pointer dereference from unchecked input. |
| SonarQube | 10.4 | c:S3510, c:S3656 | Detects missing input validation and unsafe conversions that could lead to injection or buffer issues. |
| Visual Studio Code Analysis | 17.x | C26481 | Flags unsafe array indexing operations and missing range checks. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | **Use Safe String Handling; Avoid Unbounded C String APIs**  Unbounded C string functions (gets, strcpy, strcat, sprintf, scanf("%s"), operator>> into char[] without a limit) can overflow fixed-size buffers and corrupt memory. Prefer std::string (or bounded copies with explicit sizes) and always enforce maximum lengths and NUL termination. |

| **Noncompliant Code** |
| --- |
| Reads user input into a fixed-size C buffer using unbounded operations, allowing buffer overflow and loss of NUL termination. |
| #include <cstdio>  #include <cstring>  #include <iostream>  char userName[16];  void readName() {  std::cout << "Enter name: ";  std::cin >> userName; // no max width; may overflow  char greet[32];  std::sprintf(greet, "Hi %s!", userName); // unbounded sprintf  std::puts(greet);  } |

| **Compliant Code** |
| --- |
| Uses std::string to collect input safely, enforces a maximum length, then copies/truncates into a legacy buffer with explicit bounds and NUL termination. |
| #include <string>  #include <cstring>  #include <iostream>  #include <algorithm>  char userName[16]; // legacy buffer (15 chars + '\0')  void readName() {  std::cout << "Enter name: ";  std::string tmp;  if (!std::getline(std::cin, tmp)) { std::cerr << "Input error\n"; return; }  if (tmp.size() > 15) {  std::cerr << "Name too long; truncating to 15 characters.\n";  }  const std::size\_t n = std::min<std::size\_t>(tmp.size(), 15);  std::memcpy(userName, tmp.data(), n);  userName[n] = '\0'; // ensure NUL termination  std::cout << "Hi " << userName << "!\n"; // safe stream formatting  } |

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

| **Principle(s): Integrity and Confidentiality**  This standard ensures that all string operations (copying, concatenation, formatting, and input) are handled safely to prevent buffer overflows, data corruption, or unauthorized access. In C and C++, unsafe string functions such as strcpy, sprintf, and strcat do not perform bounds checking, which can overwrite adjacent memory and allow attackers to execute arbitrary code or leak sensitive data.  Using safer alternatives like strncpy, snprintf, or C++ std::string objects enforces boundary limits and helps maintain data integrity and confidentiality across the system. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | High | Low | P1 | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12 | bufferAccessOutOfBounds, useStandardString | Detects unsafe string operations and recommends using std::string. |
| Clang-Tidy | 17.0 | clang-analyzer-security.insecureAPI.strcpy, clang-analyzer-security.insecureAPI.strcat | Identifies usage of unsafe C string functions and suggests secure alternatives. |
| Visual Studio Analyzer | 17.0 | C26446 | Flags unsafe or unchecked C string usage; recommends using safe standard library methods. |
| Flawfinder | 2.0 | buffer | Scans source code for known unsafe functions and marks them with severity ratings based on risk. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | **Prevent SQL Injection — Use Parameterized Queries / Prepared Statements**  Concatenating user input into SQL statements allows attackers to inject SQL that changes program logic or reads/writes unauthorized data. Always use parameterized queries / prepared statements (or proper escaping libraries) and validate inputs to ensure user data is never interpreted as SQL code. |

| **Noncompliant Code** |
| --- |
| Builds an SQL string by concatenating untrusted input. An attacker can include SQL syntax (e.g., '; DROP TABLE users; --) to manipulate the database. |
| #include <iostream>  #include <string>  // (pseudo) DB API  void runQuery(const std::string &sql);  void login(const std::string &user, const std::string &pw) {  // Unsafe: building SQL by concatenating user strings  std::string sql = "SELECT id FROM users WHERE username = '" + user +  "' AND password = '" + pw + "';";  runQuery(sql); // attacker-controlled input becomes part of SQL  } |

| **Compliant Code** |
| --- |
| Use a prepared statement API where SQL text is fixed and user values are supplied separately as bound parameters; the DB treats parameters as data, not executable SQL. |
| // Example using SQLite C API (conceptual):  #include <sqlite3.h>  #include <string>  #include <iostream>  void login(sqlite3 \*db, const std::string &user, const std::string &pw) {  const char \*sql = "SELECT id FROM users WHERE username = ? AND password = ?;";  sqlite3\_stmt \*stmt = nullptr;  if (sqlite3\_prepare\_v2(db, sql, -1, &stmt, nullptr) != SQLITE\_OK) {  std::cerr << "prepare failed\n"; return;  }  // bind parameters (1-based indices)  sqlite3\_bind\_text(stmt, 1, user.c\_str(), (int)user.size(), SQLITE\_TRANSIENT);  sqlite3\_bind\_text(stmt, 2, pw.c\_str(), (int)pw.size(), SQLITE\_TRANSIENT);  if (sqlite3\_step(stmt) == SQLITE\_ROW) {  int id = sqlite3\_column\_int(stmt, 0);  std::cout << "Logged in user id: " << id << '\n';  }  sqlite3\_finalize(stmt); // cleanup  } |

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

| Principle(s): Integrity and Confidentiality This standard protects *Integrity* by ensuring user-supplied data cannot alter the intended structure of SQL commands, and protects *Confidentiality* by preventing attackers from extracting sensitive records through crafted input. Parameterized queries (prepared statements with bound parameters) separate code from data, so input is treated strictly as values—not executable SQL. This blocks classic attacks like ' OR 1=1 -- and UNION-based data exfiltration. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | High | Low | P1 | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.x | c:S3649 (SQL injection) | Detects concatenated SQL strings and recommends parameterized queries. |
| Semgrep | latest | generic.sql-injection rules | Pattern-based detection of string-built SQL and tainted input flows. |
| CodeQL | latest | cpp/sql-injection | Data-flow queries that find unsafe SQL construction across functions/files. |
| Clang-Tidy | 17.x | clang-analyzer-security.insecureAPI (adjacent checks) | Complements SAST by flagging insecure API usage near data sinks. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | **Protect Memory: Enforce Bounds and Prefer RAII**  Out-of-bounds writes and lifetime mistakes corrupt memory and can be exploited. Enforce explicit size limits on copies and prefer RAII containers/smart pointers so buffers and lifetimes are managed safely. |

| **Noncompliant Code** |
| --- |
| Copies user data into a fixed buffer without verifying length, allowing a buffer overflow. |
| #include <cstring>  #include <iostream>  void store\_name(const char\* input) {  char name[32]; // 31 chars + '\0' expected  std::strcpy(name, input); // unbounded copy; may overflow  std::cout << "Hello, " << name << "\n";  } |

| **Compliant Code** |
| --- |
| Checks capacity, copies at most sizeof(name)-1 bytes, and guarantees NUL termination. |
| #include <cstring>  #include <algorithm>  #include <iostream>  void store\_name(const char\* input) {  char name[32]; // fixed capacity  const std::size\_t cap = sizeof(name) - 1; // leave room for '\0'  const std::size\_t n = std::min(std::strlen(input), cap);  std::memcpy(name, input, n); // bounded copy  name[n] = '\0'; // always terminated  std::cout << "Hello, " << name << "\n";  } |

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

| Principle(s): Integrity and Availability This standard preserves *Integrity* by preventing memory corruption (out-of-bounds reads/writes, use-after-free) and protects *Availability* by avoiding crashes and leaks that degrade or halt the service. Enforcing explicit bounds (e.g., std::array::at, std::span, length-checked copies) prevents overruns, while RAII (Resource Acquisition Is Initialization) uses objects like std::unique\_ptr, std::shared\_ptr, and standard containers to manage lifetime automatically—eliminating many classes of leaks and double-frees caused by manual new/delete. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium | P1 | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.18 | bufferAccessOutOfBounds, outOfBounds, memleak, resourceLeak | Flags out-of-bounds access and leaked/forgotten frees. |
| Clang-Tidy | 17.x | cppcoreguidelines-pro-bounds-array-to-pointer-decay, cppcoreguidelines-pro-bounds-constant-array-index, clang-analyzer-cplusplus.NewDelete, modernize-make-unique | Finds unsafe array/pointer use, raw new/delete, and suggests RAII (make\_unique). |
| MSVC Code Analysis | VS 2022 (17.x) | C26481, C26409 | C26481: bounds/pointer arithmetic misuse. C26409: avoid raw new/delete; prefer RAII. |
| [Insert text.] | [Insert text.] | N/A | Runtime detector for out-of-bounds, use-after-free, stack/heap corruption during tests. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | **Use Assertions for Critical Assumptions**  Assertions verify assumptions during development, like ensuring pointers aren’t null or values fall within expected ranges. Without them, hidden bugs can cause memory corruption or security issues. |

| **Noncompliant Code** |
| --- |
| No validation is performed; the function assumes ptr is valid and directly dereferences it. |
| #include <iostream>  void print\_value(int\* ptr) {  // no check; crashes if ptr is null  std::cout << \*ptr << std::endl;  } |

| **Compliant Code** |
| --- |
| Uses assert to enforce that ptr is not null before dereferencing, helping catch errors during testing. |
| #include <iostream>  #include <cassert>  void print\_value(int\* ptr) {  assert(ptr != nullptr); // runtime check in debug builds  std::cout << \*ptr << std::endl;  } |

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

| Principle(s): Integrity and Reliability Assertions document and enforce invariants (preconditions, postconditions, and “this must never happen” states) during development and testing. By failing fast in debug builds when assumptions are violated, they prevent corrupted state from propagating, improve diagnosability, and raise overall reliability. In release builds, assertions are typically compiled out, so they should **never** contain side effects or required program logic. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium-High | Medium | Low | P2 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.18 | assignmentInAssert, assertWithSideEffect | Detects assertions that modify state or call non-pure functions. |
| Cland-Tidy | 17.x | bugprone-assert-side-effect, readability-assertion-with-side-effects | Flags assertions that have side effects or depend on order of evaluation. |
| MSVC Code Analysis | VS 2022 (17.x) | C26444, C26440 (related guidelines) | Encourages side-effect-free checks and contract-style validation. |
| Static Analyzers (Clang SA) | latest | clang-analyzer-core.CallAndMessage | Surfaces unreachable paths and misuse around failed assumptions. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | **Handle Exceptions Explicitly; Don’t Catch-and-Ignore**  Swallowing exceptions hides failures and can leave resources or state inconsistent, creating security and reliability issues. Catch only exceptions you can handle, log appropriately, clean up, and either recover or rethrow. |

| **Noncompliant Code** |
| --- |
| Catches all exceptions and ignores the error, leaving the program in an unknown state and losing diagnostic information. |
| #include <fstream>  #include <string>  void save\_config(const std::string& path, const std::string& data) {  try {  std::ofstream out(path);  out << data; // may throw (bad path, disk full, etc.)  // no checks, no flush; exceptions ignored below  } catch (...) {  // do nothing — error is silently ignored  }  } |

| **Compliant Code** |
| --- |
| Catches the specific exception, logs it, ensures cleanup via RAII, and rethrows (or returns an error) so callers can react. |
| #include <fstream>  #include <iostream>  #include <stdexcept>  #include <string>  void save\_config(const std::string& path, const std::string& data) {  try {  std::ofstream out(path, std::ios::binary);  if (!out) throw std::runtime\_error("open failed");  out.write(data.data(),static\_cast<std::streamsize>(data.size()));  if (!out) throw std::runtime\_error("write failed");  out.flush();  if (!out) throw std::runtime\_error("flush failed");  } catch (const std::exception& e) {  std::cerr << "save\_config error: " << e.what() << "\n"; // log  throw; // or translate to status code/optional and return  }  } |

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

| Principle(s): Reliability and Integrity Exception handling must be explicit and meaningful. Swallowing exceptions (e.g., catch(...) {} or empty catch blocks) hides failures, lets corrupted state continue, and makes incidents impossible to diagnose. Handlers should be **specific** (catch by reference to std::exception or derived types), **log/report** enough context to act, and **either recover safely** or **fail fast**. Use narrow try scopes, prefer RAII for rollback/cleanup, avoid throwing from destructors, and use noexcept thoughtfully (don’t throw across noexcept boundaries). |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low-Medium | P1 | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.18 | throwInNoexceptFunction, unusedFunction (context), exceptThrowInDestructor\* | Flags throws in noexcept functions and common exception-safety pitfalls. |
| Clang-Tidy | 17.x | bugprone-empty-catch, misc-throw-by-value-catch-by-reference, cppcoreguidelines-noexcept-destructor, bugprone-exception-escape | Detects empty catch blocks, catch-by-value, throws from destructors, and exceptions escaping unintended boundaries. |
| MSVC Code Analysis | VS 2022 (17.x) | C26440, C26472 (related guidelines) | Encourages exception-safe patterns and proper use of noexcept. |
| SonarQube | 10.x | c:S2221, c:S3693 | Flags empty/general catch blocks and missing logging/handling. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | STD-008-CPP | **Validate All User Input Before Use**  Unvalidated input is one of the most common sources of vulnerabilities, leading to buffer overflows, injection attacks, and unexpected program behavior. By enforcing validation, the system ensures that only data within expected ranges, sizes, or formats is processed. |

| **Noncompliant Code** |
| --- |
| Accepts user input without checking its size or content, which can cause a buffer overflow or allow malicious characters. |
| #include <iostream>  #include <cstring>  int main() {  char buf[10];  std::cout << "Enter your name: ";  std::cin >> buf; // no validation — may overflow if >9 chars  std::cout << "Hello, " << buf << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| Validates input by limiting the size read into the buffer, and uses safer methods to prevent overflow. |
| #include <iostream>  #include <string>  int main() {  std::string name;  std::cout << "Enter your name (max 9 chars): ";  std::getline(std::cin, name);  if (name.size() > 9) {  std::cerr << "Error: input too long." << std::endl;  return 1;  }  std::cout << "Hello, " << name << std::endl;  return 0;  } |

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

| Principle(s): Integrity, Confidentiality, Availability All external input (UI fields, files, cookies, environment vars, CLI args, network data, APIs) must be **treated as untrusted**. Validate **type**, **length**, **range**, **format**, and **encoding** before use. Prefer **allowlists** (what is explicitly permitted) over denylists. Normalize/canonicalize paths and encodings prior to checks. Perform validation on the **server side** as the source of truth and **fail closed** (reject on validation failure). Proper input validation blocks injection (SQL/command), path traversal, XSS, buffer overflows, and business-logic abuse. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | High | Low | P1 | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Semgrep | latest | generic.tainting, cpp.lang.security.injection | Taint analysis rules that flag untrusted data reaching sinks without validation or parameterization. |
| CodeQL | latest | cpp/tainted-format-string, cpp/cleartext-transmission, cpp/path-injection | Data-flow queries tracing untrusted sources to dangerous sinks; great for path/format/injection issues. |
| SonarQube | 10.x | C/C++ rules for injection & sanitization (e.g., c:S3649, c:S5144) | Detects string-built queries/commands and missing sanitization; highlights risky APIs. |
| Clang-Tidy | 17.x | clang-analyzer-security.\* (e.g., insecureAPI.\*), bugprone-use-after-move (adjacent safety) | Flags insecure APIs and patterns commonly associated with unvalidated input reaching sinks. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | STD-009-CPP | **Apply the Principle of Least Privilege in Code**  Programs should only have the minimum access and permissions required to perform their task. This reduces the damage an attacker can cause if code is compromised, and it limits the scope of accidental misuse. |

| **Noncompliant Code** |
| --- |
| This program runs with elevated permissions unnecessarily, exposing the system if exploited. |
| #include <cstdlib>  int main() {  // Running a shell command as root — way too much privilege for a simple task  system("sudo cp file1.txt file2.txt");  return 0;  } |

| **Compliant Code** |
| --- |
| This code performs the same file copy using only the necessary permissions, without invoking a privileged shell. |
| #include <fstream>  #include <string>  int main() {  std::ifstream src("file1.txt", std::ios::binary);  std::ofstream dst("file2.txt", std::ios::binary);  if (!src || !dst) {  return 1; // error handling  }  dst << src.rdbuf(); // no extra privileges required  return 0;  } |

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

| Principle(s): Confidentiality, Integrity, Availability Every component should run with only the permissions it **strictly needs**—no more. Avoid elevated accounts, unnecessary filesystem or network permissions, broad process capabilities, and dangerous APIs. Drop privileges as soon as possible after startup tasks, scope sensitive operations narrowly, and default to deny. Least privilege limits blast radius: if an input-validation bug or memory error occurs, the attacker’s ability to read/modify files, spawn processes, or touch the network is constrained. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium-High | Medium | P1 | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Semgrep | latest | c.security.dangerous-functions.system, posix.chmod-permissive, posix.umask-zero | Flags dangerous calls (system, permissive chmod(0777), umask(0)), and other privilege-risky patterns. |
| CodeQL | latest | cpp/command-injection, cpp/hardcoded-credentials, cpp/insecure-temp-file | Data-/control-flow queries that surface privilege-escalation helpers (commands, creds, temp files). |
| SonarQube | 10.x | c:S4823 (command exec), c:S2068 (hardcoded creds), c:S5443 (file permissions) | Detects risky APIs, embedded secrets, and overly broad file permission settings. |
| Clang-Tidy | 17.x | clang-analyzer-security.insecureAPI.system, clang-analyzer-security.insecureAPI.vfork | Warns on APIs that commonly require or abuse elevated privileges. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Secure Error Handling & Logging | STD-010-CPP | Errors are inevitable, but messages and logs must not expose secrets (passwords, tokens, full queries, stack traces). Provide minimal, user-safe messages while recording only necessary diagnostics in a protected log without sensitive fields. |

| **Noncompliant Code** |
| --- |
| Leaks sensitive data in logs and to the user by printing raw credentials and internal error details. |
| #include <iostream>  #include <stdexcept>  #include <string>  bool authenticate(const std::string& user, const std::string& pass);  int main() {  std::string user, pass;  std::cin >> user >> pass;  try {  if (!authenticate(user, pass)) {  throw std::runtime\_error("db: password mismatch at row 42");  }  std::cout << "Login OK\n";  } catch (const std::exception& e) {  // Leaks secrets and internal details  std::cerr << "Login failed for user=" << user  << " pass=" << pass  << " error=" << e.what() << "\n";  std::cout << "Error: " << e.what() << "\n"; // exposes internal message to user  return 1;  }  } |

| **Compliant Code** |
| --- |
| Redacts secrets, logs only minimal context, and shows the user a generic message. Detailed diagnostics are kept internal without sensitive fields. |
| #include <iostream>  #include <stdexcept>  #include <string>  bool authenticate(const std::string& user, const std::string& pass);  // minimal internal logger (to secured sink/file in real apps)  void log\_auth\_failure(const std::string& user, const std::string& reason) {  // No passwords/tokens/PII; minimal context only  std::cerr << "[auth] user=" << user << " result=fail reason=" << reason << "\n";  }  int main() {  std::string user, pass;  std::cin >> user >> pass;  try {  if (!authenticate(user, pass)) {  // use generic reason; keep specifics out of user path  log\_auth\_failure(user, "invalid\_credentials");  std::cout << "Login failed.\n"; // generic user message  return 1;  }  std::cout << "Login OK\n";  } catch (const std::exception&) {  // unexpected error: still no secrets, no stack trace to user  log\_auth\_failure(user, "unexpected\_error");  std::cout << "An error occurred. Please try again.\n";  return 1;  }  } |

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

| **Principles(s):** **Confidentiality** (primary), plus **Integrity** and **Availability**.  Errors are inevitable, but what we *say* about them must follow need-to-know. User-facing messages should be minimal and safe (no stack traces, SQL, tokens, or credentials). Internal logs must be access-controlled and redact sensitive fields (passwords, keys, session IDs, PII). This prevents data leakage, credential replay, and gives attackers less intel for chaining exploits. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low-Medium | P1 | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Semgrep | latest | generic.secrets.detect, c.logging.no-secrets, cpp.lang.security.log-sensitive (custom/org rules) | Detects printing/logging of variables with names like pass, token, secret, and flags stack-trace/exception leaks to users. |
| CodeQL | latest | cpp/hardcoded-credentials, cpp/cleartext-logging (custom query), cpp/command-injection (for verbose error echoes) | Data-flow queries to find secrets and user-visible propagation of sensitive exception messages. |
| SonarQube | 10.x | c:S2068 (hardcoded credentials), c:S4823 (dangerous command exec), c:S6333 (exposing sensitive info) | Highlights secrets in code and risky logging of sensitive content. |
| Gitleaks | 8.x | Built-in rules for common token/secret patterns | Scans the repo history to prevent committing secrets into code or sample logs. |

### 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 ensures that Green Pace’s DevSecOps pipeline continuously enforces and validates the coding standards defined in this policy. The DevSecOps cycle shown in the diagram integrates **security checks** at every stage of software development—from planning and design to monitoring and response—so compliance becomes proactive rather than reactive.

In the **Design** and **Build** stages, automated security tools such as **SAST** (Static Application Security Testing), **SCA** (Software Composition Analysis), and **linting tools** can run in the IDE and during code commits to detect issues early. This enforces standards such as **safe data types**, **secure memory management**, and **proper input validation** before code ever reaches production.

During **Verify and Test**, automated pipelines trigger **DAST** (Dynamic Application Security Testing) and **IAST** (Interactive Application Security Testing) scans against the application in a controlled environment. These tools validate that runtime behavior aligns with the policy’s standards—such as preventing SQL injection, buffer overflow, or unsafe exception handling—and generate compliance reports.

In the **Transition and Health Check** stage, **CI/CD pipelines** automatically perform configuration validation, secret scanning, and deployment hardening (e.g., least-privilege service accounts, secure credentials injection). This ensures that each new deployment adheres to policy without manual intervention.

Finally, in **Production**, continuous monitoring tools—such as **SIEM systems**, log analyzers, and intrusion detection—automate the **Monitor and Detect** phase. They enforce standards related to secure logging, alerting on exceptions or data leaks, and initiating automated responses when deviations occur. The **Respond** phase uses these alerts to trigger rollback mechanisms, block malicious IPs, or revoke compromised credentials automatically.

Overall, automation transforms Green Pace’s DevOps cycle into a **DevSecOps framework**—embedding security gates, validation, and remediation directly into every phase. This continuous enforcement reduces human error, ensures consistent compliance with standards, and shortens the feedback loop between detection and correction.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | Critical | Likely | High | Urgent | 5 |
| STD-003-CPP | High | Possible | Medium | High | 3 |
| STD-004-CPP | Critical | Likely | High | Urgent | 5 |
| STD-005-CPP | High | Possible | Medium | High | 3 |
| STD-006-CPP | Medium | Possible | Low | Medium | 2 |
| STD-007-CPP | Medium | Possible | Low | Medium | 2 |
| STD-008-CPP | Critical | Likely | High | Urgent | 5 |
| STD-009-CPP | High | Possible | Medium | High | 3 |
| STD-010-CPP | Medium | Possible | Medium | 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 | Encryption at rest protects stored data (such as files, databases, and backups) from unauthorized access or theft when the system is not actively in use. Green Pace’s policy requires all sensitive data to be encrypted using **AES-256** or equivalent FIPS-validated algorithms before being written to disk. Full-disk encryption (BitLocker or LUKS) and database-level encryption ensure that if physical media or servers are compromised, the data remains unreadable without the encryption key. This policy applies to all production, development, and backup storage systems. |
| Encryption in flight | Encryption in flight secures data as it travels across networks. This policy requires the use of **TLS 1.3 or higher** for all communications between clients, servers, APIs, and microservices. Encryption in transit prevents eavesdropping, man-in-the-middle (MITM) attacks, and session hijacking by ensuring that data packets cannot be read or modified by unauthorized entities. Certificates must be managed through a centralized, automated certificate authority with renewal monitoring. |
| Encryption in use | Encryption in use protects data while it is being processed in memory, particularly during computations or within virtualized and cloud environments. This can include **trusted execution environments (TEE)**, **memory encryption (e.g., AMD SEV or Intel SGX)**, and application-level encryption of sensitive variables. The policy applies to applications handling personally identifiable information (PII) or financial data, ensuring that even runtime attacks or memory dumps cannot expose decrypted content. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies the identity of users, systems, or services before granting access. Green Pace enforces **multi-factor authentication (MFA)** across all environments—development, staging, and production—and integrates **Single Sign-On (SSO)** with corporate identity providers (e.g., Azure AD, Okta). All API endpoints require token-based authentication such as **OAuth 2.0** or **JWT**. This ensures only verified identities can initiate sessions, protecting systems from unauthorized access and credential-stuffing attacks. |
| Authorization | Authorization determines what authenticated users are allowed to do. Green Pace implements **role-based access control (RBAC)** and **least-privilege principles**, ensuring users, services, and applications receive only the permissions necessary for their tasks. Access control lists (ACLs) are reviewed quarterly, and sensitive actions (like code pushes or data exports) require elevated privileges with approval workflows. This minimizes the risk of privilege escalation and accidental data exposure. |
| Accounting | Accounting, also known as auditing, tracks user actions and system events to maintain accountability and enable incident response. All authentication and authorization events—logins, privilege changes, data modifications, and file access—must be logged to a **centralized SIEM** system such as Splunk or Microsoft Sentinel. Logs are time-stamped, encrypted, and retained for a minimum of one year. Accounting ensures traceability, supports compliance audits, and allows security teams to identify and respond to suspicious activity in real time. |

**\***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

### Mapping Security Principles to Coding Standards

Each of Green Pace’s ten C/C++ coding standards directly aligns with one or more fundamental security principles. These mappings demonstrate how each standard reinforces secure development practices across the organization.

1. **STD-001: Use Appropriate Data Types to Prevent Overflow/Underflow**

**Principles:** (3) *Fail Securely*, (5) *Keep It Simple*

Using correct data types enforces predictable behavior and prevents memory corruption. This supports *Fail Securely* by ensuring programs terminate safely rather than executing with corrupted data, and *Keep It Simple* by using safe, well-defined types over complex or unsafe arithmetic.

1. **STD-002: Validate Input Before Use**

**Principles:** (2) *Validate All Inputs*, (7) *Defense in Depth*

Ensuring all inputs are sanitized prevents injection and memory corruption. This supports layered defenses, protecting both the code and downstream systems.

1. **STD-003: Use Safe String Handling Functions**

**Principles:** (5) *Keep It Simple*, (8) *Least Privilege*

Restricting unsafe functions like strcpy() ensures memory safety. Simplicity and limited buffer access enforce both safety and principle of least privilege on data handling.

1. **STD-004: Prevent SQL Injection (Use Prepared Statements)**

**Principles:** (1) *Secure Defaults*, (2) *Validate All Inputs*, (7) *Defense in Depth*

Using parameterized queries prevents user input from altering SQL logic. Secure defaults and layered validation minimize the risk of injection attacks.

1. **STD-005: Protect Memory – Enforce Bounds and Prefer RAII**

**Principles:** (3) *Fail Securely*, (6) *Secure Resource Management*

RAII (Resource Acquisition Is Initialization) ensures proper memory cleanup. Programs fail gracefully and do not leak sensitive data or leave resources exposed.

1. **STD-006: Use Assertions for Critical Assumptions**

**Principles:** (9) *Assume External Systems Are Insecure*, (3) *Fail Securely*

Assertions verify internal integrity before continuing execution, preventing cascading failures and supporting secure failure behavior.

1. **STD-007: Handle Exceptions Explicitly; Don’t Catch-and-Ignore**

**Principles:** (3) *Fail Securely*, (10) *Visibility and Accountability*

Proper exception handling ensures that failures trigger logging, monitoring, and appropriate termination instead of silent errors.

1. **STD-008: Validate All User Input Before Use**

**Principles:** (2) *Validate All Inputs*, (7) *Defense in Depth*

Continuous validation prevents cross-site scripting, injection, and buffer overflow vulnerabilities, ensuring protection at every trust boundary.

1. **STD-009: Apply the Principle of Least Privilege in Code**

**Principles:** (8) *Least Privilege*, (7) *Defense in Depth*

Restricting access to necessary privileges only minimizes potential damage from exploitation, supporting layered protection across modules.

1. **STD-010: Secure Error Handling and Logging**

**Principles:** (10) *Visibility and Accountability*, (4) *Separation of Duties*

Ensuring that logs exclude sensitive information but maintain traceability enables audit readiness, supports incident response, and preserves confidentiality while maintaining visibility.

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 | 09/18/2025 | Initialized 10 core security standards | Andrew Torrez | Dewin Andujar |
| 3.0 | 10/11/2025 | Finished second half of policy | Andrew Torrez |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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