**Green Pace Developer: Security Policy Guide**



# 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 | Anything that comes from outside the program files, env vars, command-line args, network requests, database reads can be wrong or malicious. Check type, length, range, format, and encoding up front, and normalize it before using it. Catching bad input early stops injections, crashes, and weird logic bugs from spreading |
| 1. Heed Compiler Warnings | Treat warnings like errors. Compilers and static tools flag risky stuff, narrowing conversions, use-after-free, ignored return values, etc. Cleaning these up improves reliability and closes off easy attack paths before code ever ships. |
| 1. Architect and Design for Security Policies | Plan for security at the design stage. Define how auth, logging, data handling, and crypto should work, then build modules and data flows that enforce those rules. Clear boundaries and least privilege make it harder for a small bug to turn into a big problem. |
| 1. Keep It Simple | Complex code is harder to reason about and easier to break. Use small functions, clear ownership, and as little shared state as possible. Simple code is easier to review, test, and secure. |
| 1. Default Deny | If you’re not sure, block it. Allow only what’s explicitly approved instead of trying to block everything bad. Deny-by-default in configs, APIs, and file access limits mistakes and reduces the blast radius when something goes wrong. |
| 1. Adhere to the Principle of Least Privilege | Give code and services only the permissions they actually need. Limit file, network, and OS capabilities. Inside the code, pass only the data and handles a function must have. Less privilege means fewer ways to be exploited. |
| 1. Sanitize Data Sent to Other Systems | Before sending data to SQL, a shell, HTML, JSON, or even logs, encode/escape it for that specific context. This avoids injection, keeps logs trustworthy, and prevents your output from becoming someone else’s exploit. |
| 1. Practice Defense | No single control saves you every time. Stack layers validation, authentication, authorization, rate limiting, isolation, and monitoring, so if one layer fails, the others still protect you. |
| 1. Use Effective Quality Assurance Techniques | Mix unit tests, fuzzing, property-based tests, code reviews, SAST/DAST, and dependency scans. Focus coverage on critical paths. Good QA catches security issues the same way it catches regular bugs. |
| 1. Adopt a Secure Coding Standard | Follow a recognized standard (like SEI CERT C/C++) and make it part of your CI. Shared rules reduce confusion, guide reviews, and let automated tools enforce the basics consistently. |

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Use explicit, fixed‑width types and avoid implicit narrowing/signed‑unsigned mismatches.  Type width and signedness vary by platform. Implicit conversions cause overflow, underflow, and logic errors that lead to memory corruption or authorization flaws. Fixed‑width types and explicit casts make intent unambiguous. |

| **Noncompliant Code (implicit narrowing & sign mix‑up)** |
| --- |
| Assigns a large literal to int, then narrows to short; also compares signed int to unsigned size\_t. |
| int main() {  int big = 3000000000; // overflow on 32‑bit int  short s = big; // narrowing; implementation‑defined  size\_t n = 10;  int i = -1;  if (i < n) { /\* true on many ABIs due to promotion \*/ }  } |

| **Compliant Code (fixed‑width + explicit checks)** |
| --- |
| Uses std::uint32\_t for range, guards casts, and compares like‑typed values. |
| #include <cstdint>  #include <limits>  #include <stdexcept>  int main() {  std::uint32\_t big = 3000000000u; // defined width/range  std::uint16\_t s;  if (big > std::numeric\_limits<std::uint16\_t>::max())  throw std::overflow\_error("narrowing would overflow");  s = static\_cast<std::uint16\_t>(big);  std::size\_t n = 10;  std::size\_t j = 1; // compare like types  if (j < n) { /\* safe \*/ }  } |

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

| **Principles(s):** Heed Compiler Warnings: catches narrowing and signed/unsigned mistakes early; we treat warnings as errors so these never ship.  Keep It Simple: explicit, fixed-width types and like-typed comparisons are easier to read, review, and verify.  Use Effective Quality Assurance Techniques: unit tests + sanitizers + static analysis expose bad conversions and overflow paths.  Adopt a Secure Coding Standard: aligns with CERT INT rules and org policy so reviewers/tools are on the same page. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| |  | | --- | | GCC/Clang/MSVC |  |  | | --- | |  | | GCC 13+, Clang 17+, MSVC 19.x | -Wall -Wextra -Wconversion -Wsign-conversion -Werror (or /Wall /WX, warns C4244/C4389) | Treat warnings as errors; flags surface narrowing, sign mismatches, and implicit conversions. |
| |  | | --- | | clang-tidy |  |  | | --- | |  | | 17+ | cppcoreguidelines-narrowing-conversions, hicpp-signed-bitwise, clang-analyzer-core | Fails CI on implicit narrowing, risky signed ops, and analyzer findings. |
| cppcheck | 2.13+ | --enable=warning,portability,performance | Highlights sign/width issues and suspicious conversions across the codebase. |
| |  | | --- | | CodeQL (C/C++) |  |  | | --- | |  | | Latest pack | Integer conversion/overflow queries (e.g., overflow/integer, tainted-arithmetic) | Flags unsafe arithmetic and implicit conversions that could lead to overflow or logic bugs |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Validate value ranges and invariants at trust boundaries. Unchecked values (sizes, indices, counts) lead to OOB access, integer wrap, denial of service, or logic bypass. Validate before allocation, indexing, or arithmetic. |

| **Noncompliant Code (unchecked size → OOB/alloc issues)** |
| --- |
| Uses user‑provided length to allocate and copy without upper bounds. |
| #include <cstring>  #include <cstdlib>  void copyUser(char\* src, std::size\_t len) {  char\* buf = (char\*)std::malloc(len); // len may be huge or 0  std::memcpy(buf, src, len); // OOB if src < len  // ...  std::free(buf);  } |

| **Compliant Code (bounded, validated, fail‑fast)** |
| --- |
| Enforces sane maximums and consistency; uses std::vector for RAII. |
| #include <vector>  #include <stdexcept>  constexpr std::size\_t MAX\_COPY = 1 << 20; // 1 MiB cap  void copyUser(const char\* src, std::size\_t len) {  if (!src) throw std::invalid\_argument("null src");  if (len == 0 || len > MAX\_COPY) throw std::out\_of\_range("bad length");  std::vector<char> buf(len);  std::copy(src, src + len, buf.data());  } |

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

| **Principles(s):** Validate Input Data: check size/index/count at the boundary before use.  Keep It Simple: clear caps and guard clauses make checks obvious and reliable.  Defense in Depth: validate, cap, and re-check before allocation/copy.  Effective QA: tests, fuzzing, and sanitizers catch missed edge cases. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| GCC/Clang/MSVC | GCC 13+, Clang 17+, MSVC 19.x | -Wall -Wextra -Werror, analyzer warnings | Surfaces unchecked lengths, dead branches, and risky calls. |
| |  | | --- | | clang-tidy |  |  | | --- | |  | | 17+ | clang-analyzer-core (CallAndMessage, Dereference), bugprone-narrowing-conversions, cppcoreguidelines-pro-bounds-array-to-pointer-decay | Finds paths that can pass bad lengths/indexes to APIs. |
| cppcheck | 2.13+ | enable=warning,style,performance,portability | Flags potential out-of-bounds and size misuse. |
| |  | | --- | | CodeQL (C/C++) |  |  | | --- | |  | | Latest | cpp/tainted-arithmetic, cpp/overflow-in-arithmetic, cpp/out-of-bounds-read | Tracks untrusted sizes into allocation/copy/index sinks. |
| |  | | --- | | ASan/UBSan |  |  | | --- | |  | | Clang/GCC | |  | | --- | | -fsanitize=address,undefined |  |  | | --- | |  | | Catches OOB reads/writes and integer UB at runtime in tests. |
| Fuzzing | libFuzzer/AFL++ | N/A | Generates adversarial sizes and inputs to hit boundary mistakes. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Avoid unsafe C string APIs; prefer std::string/std::string\_view and bounded formatting. Unbounded copy/formatting functions (e.g., strcpy, sprintf) are common overflow sources. C++ string types manage size, and safe formatting prevents clobbering memory. |

| **Noncompliant Code (buffer overflow via strcpy)** |
| --- |
| Copies attacker-controlled input into a fixed buffer without length checks. |
| #include <cstring>  void greet(const char\* name) {  char buf[32];  std::strcpy(buf, name); // potential overflow  // ...  } |

| **Compliant Code (C++ strings + bounded formatting)** |
| --- |
| Uses std::string to hold input safely and snprintf to bound output. |
| #include <string>  #include <cstdio>  void greet(const std::string& name) {  char out[32];  std::snprintf(out, sizeof(out), "Hi %.\*s", (int)name.size(), name.c\_str());  // ...  } |

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

| **Principles(s):** Validate Input Data: check string length/encoding before use.  Keep It Simple: prefer std::string/string\_view and bounded APIs.  Defense in Depth: validate, bound, and encode where needed.  Adopt a Secure Coding Standard: follow CERT STR rules and enforce in CI. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| GCC/Clang/MSVC | GCC 13+, Clang 17+, MSVC 19.x | -Wformat -Wformat-security -Wstringop-overflow (GCC), Clang -Wunsafe-buffer-usage, MSVC /sdl /analyze | Flags unsafe formatting and buffer copies; enables SDL checks on MSVC. |
| |  | | --- | | Libc hardening |  |  | | --- | |  | | glibc/  uClibc | -D\_FORTIFY\_SOURCE=2 with -O2 | Detects some overflows in calls like memcpy/strcpy at build/runtime. |
| clang-tidy | 17+ | cert-str34-c (bounds for string funcs), bugprone-suspicious-string-compare, modernize-use-std-string | Warns on unsafe C string APIs and suspicious string use. |
| |  | | --- | | cppcheck |  |  | | --- | |  | | 2.13+ | --enable=warning,style,performance | Reports unsafe string copies and potential overflows. |
| |  |  | | --- | --- | | |  | | --- | |  |   CodeQL  (C/C++) |  |  | | --- | |  | | Latest | cpp/unsafe-format-string, cpp/potentially-dangerous-function | Finds unbounded sprintf/strcpy and tainted format issues. |
| ASan | |  | | --- | | Clang/GCC |  |  | | --- | |  | | -fsanitize=address | Catches buffer overflows in tests at runtime. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Use parameterized queries/prepared statements only. Concatenating SQL with user data enables injection, data exfiltration, and corruption. Prepared statements separate code from data and ensure correct encoding. |

| **Noncompliant Code (string‑built SQL)** |
| --- |
| Builds a WHERE clause by concatenating untrusted input. |
| #include <string>  std::string findUser(const std::string& user) {  std::string sql = "SELECT \* FROM users WHERE name='" + user + "'"; // vulnerable  // exec(sql.c\_str());  return sql;  } |

| **Compliant Code (prepared statement – SQLite example)** |
| --- |
| Binds the parameter instead of concatenating it. |
| #include <sqlite3.h>  void findUser(sqlite3\* db, const std::string& user) {  const char\* q = "SELECT \* FROM users WHERE name = ?1";  sqlite3\_stmt\* stmt = nullptr;  if (sqlite3\_prepare\_v2(db, q, -1, &stmt, nullptr) != SQLITE\_OK) return;  sqlite3\_bind\_text(stmt, 1, user.c\_str(), (int)user.size(), SQLITE\_TRANSIENT);  while (sqlite3\_step(stmt) == SQLITE\_ROW) {  // read columns  }  sqlite3\_finalize(stmt);  } |

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

| **Principles(s):** Validate Input Data: never trust user strings for query text.  Default Deny: block ad-hoc SQL; only allow prepared statements.  Sanitize Data Sent to Other Systems: bind values so the DB treats them as data.  Defense in Depth: pair parameterization with least-privilege DB roles and input caps.  Adopt a Secure Coding Standard: aligns with CERT SQL/STR guidance and team policy. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeQL  (C/C++) | Latest | cpp/sql-injection | Flags string-built queries and tainted data flowing into SQL APIs. |
| |  | | --- | |  |   SonarQube C++ | Current LTS | SQL injection rules (custom regex profile for concatenated SQL) | Detects concatenation into query strings; fails quality gate. |
| |  | | --- | | clang-tidy (custom  check) |  |  | | --- | |  | | 17+ | Custom matcher for SELECT/INSERT/UPDATE/DELETE in string ops | Blocks literals + +/format building queries; whitelist prepared APIs. |
| Unit/DAST tests | N/A | Test payloads: \"' OR 1=1 --\", \"'; DROP TABLE x; --\" | Proves queries are parameterized (payload yields no injection effects). |
| SCA (dependency scan) | Latest | e.g., OWASP Dependency-Check, Snyk | Ensures DB drivers/ORMs are patched to avoid known query-encoding bugs. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Eliminate manual new/delete; use RAII and smart pointers. Manual lifetime leads to leaks, double‑free, and UAF. RAII (unique\_ptr, shared\_ptr, containers) ties lifetime to scope, preventing many classes of memory bugs. |

| **Noncompliant Code (double free / early return leak)** |
| --- |
| Different control paths mishandle ownership. |
| #include <cstring>  char\* dup(const char\* s) {  char\* p = new char[std::strlen(s) + 1];  if (!p) return nullptr; // leak in other paths  std::strcpy(p, s);  delete[] p; // freed here  delete[] p; // double free  return p; // UAF  } |

| **Compliant Code (RAII with std::unique\_ptr)** |
| --- |
| Scope controls lifetime; no manual delete. |
| #include <memory>  #include <string>  std::unique\_ptr<char[]> dup(const std::string& s) {  auto p = std::make\_unique<char[]>(s.size() + 1);  std::copy(s.begin(), s.end(), p.get());  p[s.size()] = '\0';  return p; // move  } |

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

| **Principles(s):** Keep It Simple: RAII and containers reduce lifetime paths and mistakes.  Defense in Depth: pair RAII with sanitizers and runtime guards.  Effective QA: tests under sanitizers expose leaks, UAF, and double frees.  Adopt a Secure Coding Standard: follows CERT MEM/ERR/EXP guidance. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang/GCC sanitizers | Latest | -fsanitize=address,leak,undefined | Catches UAF, double free, leaks, and UB during tests. |
| clang-tidy | 17+ | cppcoreguidelines-owning-memory, modernize-make-unique, modernize-use-std-smart-ptr, clang-analyzer-cplusplus.NewDeleteLeaks | Flags manual new/delete, suggests smart pointers, detects leaks. |
| CodeQL (C/C++) | Latest | cpp/double-free, cpp/use-after-free, cpp/missing-delete | Finds lifetime bugs across code paths. |
| Valgrind Memcheck | Latest | N/A | Detects leaks, double frees, and invalid reads/writes on nightly builds. |
| cppcheck | 2.13+ | --enable=warning,style,performance | Reports leaks/uninitialized use and risky patterns. |
| Build rules | N/A | Ban APIs list | CI grep/lints to block new/delete/malloc/free in app code (allow in low-level wrappers only). |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use assert for internal invariants only; never for external input validation. assert is compiled out in release builds (NDEBUG), so relying on it for runtime checks can create production-only vulnerabilities. Validate untrusted input with real error handling. |

| **Noncompliant Code (assert used for input)** |
| --- |
| Disables validation in production, allowing bad data through. |
| #include <cassert>  int div(int a, int b) {  assert(b != 0); // removed in release  return a / b; // UB/crash in production  } |

| **Compliant Code (exceptions/error returns)** |
| --- |
| Validates at runtime; uses assert only for internal invariants. |
| #include <stdexcept>  int divChecked(int a, int b) {  if (b == 0) throw std::invalid\_argument("b cannot be 0");  return a / b;  } |

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

| **Principles(s):** Validate Input Data: real runtime checks for untrusted input; don’t rely on assert.  Keep It Simple: keep assert for internal invariants only; use clear error paths for inputs.  Effective QA: tests must pass with and without NDEBUG; sanitizers catch crashes from missed checks.  Secure Coding Standard: aligns with CERT guidance to avoid assert for external data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Build configs | N/A | -DNDEBUG (release), no -DNDEBUG (debug) | CI runs tests in both modes to prove behavior doesn’t depend on assert. |
| clang-tidy | 17+ | bugprone-assert-side-effect, misc-static-assert | Flags assert with side effects and suggests static\_assert for compile-time facts. |
| cppcheck | 2.13+ | |  | | --- | | assertWithSideEffect |  |  | | --- | |  | | Detects misuse of assert that alters program state. |
| Code review/grep | N/A | \\bassert\\s\*\\( in public/API code | Simple gate to block assert in input-handling paths; require real error handling. |
| Code review/grep | Clang/GCC | |  | | --- | | -fsanitize=address,undefined |  |  | | --- | |  | | Catches crashes/UB at runtime if a missing check slips through. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Follow safe exception practices: no throws from destructors, catch by (const) reference, and maintain strong/ basic exception safety via RAII. Throwing from destructors during stack unwinding terminates the process; incorrect catch types slice objects; lacking RAII leaks on error paths. |

| **Noncompliant Code (throwing destructor & catch by value)** |
| --- |
| Destructor throws; catch by value slices the exception. |
| #include <stdexcept>  struct Bad {  ~Bad() noexcept(false) { throw std::runtime\_error("dtor"); }  };  int main() {  try {  Bad b; // stack unwinding will call ~Bad and throw  } catch (std::exception e) { // slicing  // ...  }  } |

| **Compliant Code (noexcept dtor, catch by ref, RAII)** |
| --- |
| Destructors are noexcept; exceptions are caught by reference. |
| #include <stdexcept>  #include <memory>  struct Good {  ~Good() noexcept = default; // never throw  };  int main() {  try {  auto p = std::make\_unique<Good>();  // ...  } catch (const std::exception& e) {  // log & handle  }  } |

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

| **Principles(s):** Keep It Simple: RAII and clear ownership patterns reduce error paths during exceptions.  Defense in Depth: combine RAII, noexcept destructors, and centralized handlers so one mistake doesn’t crash the app.  Effective QA: tests exercise throw paths and verify strong/basic exception safety.  Secure Coding Standard: aligns with CERT ERR/EXP guidance (no throwing destructors, catch by reference). |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 17+ | cppcoreguidelines-noexcept-destructor, misc-throw-by-value-catch-by-reference, cert-err60-cpp | Warns on destructors that can throw, catches by value, and exception-safety issues. |
| CodeQL (C/C++) | Latest | cpp/throw-in-destructor, cpp/expensive-exception-catch-by-value | Finds throws from destructors and slicing/costly catch-by-value patterns. |
| MSVC /analyze | 19.x | C++ Core checks | Flags exception-safety and RAII violations on Windows builds. |
| Unit tests | N/A | Throw-path tests | Verify strong/basic exception guarantees; ensure cleanup via RAII. |
| Build rules | N/A | |  | | --- | | Enforce noexcept on destructors |  |  | | --- | |  | | CI lint that requires noexcept destructors in non-test code; fail on violations. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Pointer Ownership & Lifetime | [STD-008-CPP] | Document and enforce pointer ownership; prefer non‑owning T\*/std::span/std::string\_view for observers. Clear ownership prevents UAF and leaks. Distinguish owning (unique\_ptr, shared\_ptr, containers) from non‑owning observers. |

| **Noncompliant Code (ownership)** |
| --- |
| Raw pointer implies unclear lifetime; callee may delete. |
| void take(char\* p) {  delete[] p; // who owns this?  }  void f() {  char\* p = new char[4]{'t','e','s','t'};  take(p); // ownership confusion  } |

| **Compliant Code (explicit ownership & observers)** |
| --- |
| Caller retains ownership; callee uses span/view. |
| #include <memory>  #include <span>  void use(std::span<const char> view) { /\* read‑only \*/ }  void f() {  auto buf = std::make\_unique<char[]>(4);  buf[0]='t'; buf[1]='e'; buf[2]='s'; buf[3]='t';  use(std::span<const char>(buf.get(), 4));  } // buf frees automatically |

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

| **Principles(s):** Keep It Simple: clear ownership rules and observers cut lifetime mistakes.  Least Privilege: pass only what a callee needs—non-owning views for read-only access.  Defense in Depth: pair ownership rules with bounds-checked views (std::span) and reviews.  Secure Coding Standard: aligns with CERT MEM/EXP guidance on ownership and lifetime. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| clang-tidy | 17+ | cppcoreguidelines-owning-memory, modernize-use-std-smart-ptr, modernize-use-std-span, clang-analyzer-cplusplus.NewDelete | Flags raw ownership, suggests smart pointers and std::span, detects bad new/delete. |
| Clang/GCC sanitizers | Latest | -fsanitize=address,leak,undefined | Catches UAF, leaks, and UB from lifetime bugs during tests. |
| CodeQL (C/C++) | Latest | cpp/use-after-free, cpp/double-free, cpp/memory-leak | Tracks ownership flows to find frees/uses across paths. |
| Valgrind Memcheck | Latest | N/A | Nightly job to catch leaks and invalid frees on long-running tests. |
| cppcheck | 2.13+ | --enable=warning,style,performance | |  | | --- | |  |  |  | | --- | | Points out leaks and suspicious pointer  Use. | |
| Build rules | N/A | Ban list for new/delete/malloc/free in app code | |  | | --- | |  |  |  | | --- | | CI grep/lint: only permitted in isolated  allocators; prefer RAII/containers. | |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Integer Arithmetic Safety | [STD-009-CPP] | Check arithmetic for overflow/underflow before performing the operation. Signed overflow is undefined behavior; even unsigned wrap can drive buffer miscalculations and allocation errors. |

| **Noncompliant Code (size calculation overflow)** |
| --- |
| Multiplies user sizes without checks; overflow yields small allocation, then OOB write. |
| #include <cstdlib>  void\* makeBuf(std::size\_t n, std::size\_t m) {  std::size\_t bytes = n \* m; // may wrap to small value  return std::malloc(bytes);  } |

| **Compliant Code (pre‑check and safe helpers)** |
| --- |
| Validates multiplication using division check. |
| #include <limits>  #include <stdexcept>  #include <cstdlib>  void\* makeBuf(std::size\_t n, std::size\_t m) {  if (n != 0 && m > (std::numeric\_limits<std::size\_t>::max() / n))  throw std::overflow\_error("size overflow");  return std::malloc(n \* m);  } |

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

| **Principles(s):** Validate Input Data: check sizes/counts before doing arithmetic.  Keep It Simple: use clear guards and helper functions for math checks.  Effective QA: tests and fuzzing hit boundary values; sanitizers catch UB.  Secure Coding Standard: follows CERT INT rules on integer operations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| UBSan | Clang/GCC | -fsanitize=signed-integer-overflow,undefined | Traps signed overflow/UB during tests. |
| ASan | Clang/GCC | -fsanitize=address | Catches OOB writes caused by wrapped size math. |
| clang-tidy | 17+ | cert-int30-c, cert-int32-c, clang-analyzer-core.DivideZero | Warns on risky integer ops and divide-by-zero. |
| CodeQL (C/C++) | Latest | cpp/overflow-in-arithmetic, cpp/tainted-arithmetic | Tracks untrusted inputs into overflow-prone math. |
| MSVC /analyze | 19.x | C++ Core checks | Flags suspicious integer conversions/overflows on Windows builds. |
| Fuzzing | libFuzzer/AFL++ | N/A | Generates large/small sizes to exercise overflow paths. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File Path Handling & Traversal | [STD-010-CPP] | Canonicalize and constrain file paths; never trust user‑supplied paths. Relative segments (e.g., ..) and symlinks enable traversal outside intended directories, exposing secrets or config. |

| **Noncompliant Code (direct use of user path)** |
| --- |
| Opens any path provided by the user. |
| #include <fstream>  #include <string>  void readUserFile(const std::string& p) {  std::ifstream f(p); // may access sensitive files  } |

| **Compliant Code (jail to base dir + canonicalize)** |
| --- |
| Resolves to canonical path and enforces base directory prefix. |
| #include <filesystem>  #include <fstream>  void readUserFile(const std::filesystem::path& user) {  const std::filesystem::path base = "/var/app/public";  auto canon = std::filesystem::weakly\_canonical(base / user);  if (canon.native().rfind(base.native(), 0) != 0) {  throw std::runtime\_error("path traversal");  }  std::ifstream f(canon, std::ios::binary);  } |

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

| **Principles(s):** Validate Input Data: treat paths as untrusted; normalize and check before use.  Default Deny: restrict access to a fixed base directory; block anything outside it.  Defense in Depth: combine canonicalization, OS perms, and least-privilege service users.  Secure Coding Standard: follow file-I/O rules that forbid raw user paths. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeQL (C/C++) | Latest | cpp/path-injection | Flags user-controlled paths flowing into file APIs. |
| |  | | --- | | SonarQube C++ |  |  | | --- | |  | | Current LTS | Path traversal rules (custom profile) | Detects .. joins and concatenated user paths. |
| clang-tidy (custom matcher) | 17+ | Ban raw std::ifstream(std::string) with tainted input | Require wrappers that canonicalize + enforce base dir. |
| Unit/Integration tests | N/A | |  | | --- | | Traversal inputs (../, symlinks) |  |  | | --- | |  | | Prove the jail prevents escape and rejects bad paths. |
| Runtime policy | N/A | |  |  |  | | --- | --- | --- | | |  | | --- | | AppArmor/SELinux profile |  |  | | --- | |  | |  |  | | --- | |  | | |  | | --- | |  |  |  | | --- | | OS policy confines process to allowed  directories.. | |
| Build rules | N/A | |  | | --- | | Grep/lint for direct file opens |  |  | | --- | |  | | Enforce use of a vetted OpenInJail(base, userPath) helper. |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

[Insert your written explanations here.]

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Likely | Low | High | 4 |
| STD-003-CPP | High | Likely | Low | High | 4 |
| STD-004-CPP | High | Likely | Low | High | 5 |
| STD-005-CPP | High | Possible | Medium | High | 4 |
| STD-006-CPP | Medium | Possible | Low | Medium | 2 |
| STD-007-CPP | Medium | Possible | Low | Medium | 2 |
| STD-008-CPP | High | Possible | Low | High | 3 |
| STD-009-CPP | High | Possible | Low | High | 3 |
| STD-010-CPP | High | Possible | 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 sensitive data stored on disks, snapshots, and backups must be encrypted with AES-256 (or stronger) using a managed KMS. Keys are rotated at least annually and on any suspected compromise. Access to keys follows least-privilege and is logged. |
| Encryption in flight | All traffic carrying sensitive data uses TLS 1.2+ with strong ciphers. Internal service-to-service calls use mTLS. HTTP is redirected to HTTPS. Certificates are renewed automatically and pinned where applicable. |
| Encryption in use | Secrets in memory are minimized and zeroized after use. Disable core dumps in production; scrub process memory in crash handlers. For high-risk workloads, prefer hardware-backed enclaves/TEEs when available. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Require MFA for users; use short-lived tokens/claims for services (OIDC/JWT or mTLS). No shared accounts. Passwords meet org policy; secrets are vaulted. All login events are logged. |
| Authorization | Role-based access with least privilege. Every sensitive API checks authorization in the service layer (not only the UI). Changes to roles/permissions require approval and are logged |
| Accounting | Record who did what and when: user logins, file access, data reads/writes, schema changes, admin actions, and creation/deletion of users. Logs are centralized, immutable, time-synced, and retained per policy. |

**\***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.

STD-001-CPP (Data Type): 2, 4, 9, 10

Heed warnings; keep types simple/explicit; test conversions; follow the coding standard.

STD-002-CPP (Data Value): 1, 4, 8, 9

Validate inputs; keep checks straightforward; layer bounds in multiple places; test edges.

STD-003-CPP (String Correctness): 1, 4, 8, 10

Validate/normalize strings; prefer safe APIs; layer encoding rules; follow CERT STR.

STD-004-CPP (SQL Injection): 1, 5, 7, 8, 10

Validate inputs; default-deny string-built SQL; sanitize via binding; layer DB perms; follow policy.

STD-005-CPP (Memory Protection): 4, 8, 9, 10

RAII keeps it simple; combine with sanitizers; test lifetimes; align with MEM rules.

STD-006-CPP (Assertions): 1, 4, 9, 10

Real runtime validation for inputs; keep assert for internals; test with/without NDEBUG; follow standard.

STD-007-CPP (Exceptions): 4, 8, 9, 10

Simple RAII patterns; layered handling; test throw paths; follow ERR/EXP guidance.

STD-008-CPP (Pointer Ownership & Lifetime): 4, 6, 8, 10

Clear ownership; least data/handles passed; layered checks; document in the standard.

STD-009-CPP (Integer Arithmetic Safety): 1, 4, 9, 10

Validate sizes; simple guard helpers; tests/fuzzing; use CERT INT rules.

STD-010-CPP (File Path Handling & Traversal): 1, 5, 8, 10

Validate/canonicalize paths; default-deny outside base dir; layer OS policies; codify in standard.

**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 | 09/21/2025 | Project One draft: added risk assessments, automation, encryption & Triple-A, and principle mapping. | Christopher Phillips | Instructor |
| 1.2 | 10/10/2025 | Incorporated reviewer feedback; clarified tooling checkers and DevSecOps gates. | Christopher Phillips |  |
| 2.0 | 10/12/2025 | Final approved policy for audit; locked standards STD-001..STD-010 and enforcement thresholds | Christopher Phillips |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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