Green Pace

Green Pace Secure Development Policy

Gonzalo Patino

Contents

Green Pace

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://).

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. Validate Input Data	Accept nothing by default. Define strict schemas and validate type, length, range, format, and encoding before use. Normalize data first, then apply allowlists to block injection, overflow, traversal, and deserialization attacks
2. Heed Compiler Warnings	Turn on the highest warning level and treat warnings as errors. Warnings often indicate real defects like truncation, sign or size mismatches, uninitialized reads, and undefined behavior that attackers can exploit.
3. Architect and Design for Security Policies	Design from the start with explicit security policies: authentication, authorization, accounting, data handling, and error handling. Use threat modeling and clear trust boundaries to minimize the attack surface and fail safely.
4. Keep It Simple	Prefer simple, readable, and proven constructs over clever or complex ones. Simplicity cuts bug rate, eases review and testing, and reduces undefined behavior that can become vulnerabilities.
5. Default Deny	Deny access by default and grant only what is explicitly allowed. Inputs are rejected until validated, features are disabled until needed, and network or file access is blocked unless a rule permits it.
6. Adhere to the Principle of Least Privilege	Run code and processes with the minimum permissions required. Drop privileges as early as possible, scope tokens and keys tightly, and restrict resource access to limit blast radius if something goes wrong.
7. Sanitize Data Sent to Other Systems	Before emitting data, encode or escape per destination context and protocol. Use parameterized queries, safe formatters, and allowlisted transformations to prevent SQL, shell, HTML, XML, JSON, CSV, and log injection.

Principles	Write a short paragraph explaining each of the 10 principles of security.
8. Practice Defense in Depth	Layer independent controls so one failure does not compromise the system. Combine validation, parameterization, memory safety mitigations, sandboxing, RBAC, rate limiting, logging, and alerting.
9. Use Effective Quality Assurance Techniques	Apply code reviews, static and dynamic analysis, unit and integration tests, fuzzing, and sanitizers consistently. Automate tests in CI to prevent regressions and measure coverage for critical paths.
10. Adopt a Secure Coding Standard	Follow an established standard such as SEI CERT C++ and make it enforceable via tooling and reviews. Define rules for memory ownership, error handling, concurrency, and APIs, and keep the standard updated.

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	Data Type Safety- Using correct and consistent data types prevents truncation, sign errors, and unexpected behavior. Enforcing strong type usage ensures that operations are predictable and reduces opportunities for overflow and underflow attacks.

Noncompliant Code

This example stores a value in an inappropriate type, causing truncation and incorrect results.

```
short s;
int i = 70000;
s = i; // Truncation occurs, value is lost
```

Compliant Code

Here, the variable is stored in a properly sized type, eliminating truncation and maintaining correct results.

```
int s;
int i = 70000;
s = i; // Safe, no truncation
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Medium	Low	P1	High

Automation

Tool	Version	Checker	Description Tool
Compiler warnings	clang++/g++ /MSVC, current	-Wall -Wextra -Wconversion -Wsign-conversion -Werror	Enable strict warnings and treat as errors to catch narrowing and sign-mismatch conversions.
clang-tidy	16+	cppcoreguidelines-narrowing-conversions, hicpp-signed-bitwise, bugprone-sizeof-expression	Flags narrowing conversions, signedness issues, and suspicious size/width use.
cppcheck	2.10+	enable warnings for type, bounds, portability	Static analysis to detect truncation, implicit casts, and type misuse.
UndefinedBehaviorSanitizer (UBSan)	in clang/gcc	-fsanitize=undefined -fno-sanitize-recover=undefined	Detects integer overflows, invalid shifts, and other UB at runtime.
CodeQL	Latest	integer and type-conversion queries	Repository scanning for truncating casts, sign/size bugs, and narrowing

Coding Standard 2

Coding Standard	Label	Name of Standard
Data Value	STD-002-CPP	Data Value Validation. All externally influenced values must be validated for range, sign, and invariants before use. Proper checks prevent division by zero, out-of-range indexing, integer overflow or underflow, and logic errors that attackers can trigger.

Noncompliant Code

Divides and indexes using unchecked input, allowing division by zero and out-of-bounds access.

```
#include <vector>
int compute(const std::vector<int>& v, int idx, int divisor) {
    // No validation
    int q = 100 / divisor;          // UB if divisor == 0
    return v[idx] + q;              // UB if idx out of range
}
```

Compliant Code

Validates divisor and index, and handles error cases safely without undefined behavior.

```
#include <vector>
#include <limits>
#include <optional>

std::optional<int> compute(const std::vector<int>& v, int idx, int
divisor) {
    if (divisor == 0) return std::nullopt;                  // prevent
divide by zero
    if (idx < 0 || static_cast<size_t>(idx) >= v.size())    // prevent
OOB
        return std::nullopt;

    // Prevent overflow in intermediate math if inputs can be large
    if (100 > std::numeric_limits<int>::max() - v[idx])     // simple
guard example
        return std::nullopt;

    int q = 100 / divisor;
    return v[idx] + q;
}
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Medium	Low	P1	High

Automation

Tool	Version	Checker	Description Tool
Compiler warnings	clang++/g++ /MSVC, current	-Wall -Wextra -Wpedantic -Werror	Treat warnings as errors to highlight suspicious conditions and unused paths that often indicate missing validation.
clang-tidy	16+	bugprone-integer-division, bugprone-narrowing-conversions, cppcoreguidelines-pro-bounds-constant-array-index, readability-simplify-boolean-expr	Finds risky integer math, narrowing, and potential out-of-bounds index issues or tautological checks.
cppcheck	2.10+	enable warnings for bounds, nullPointer, unusedFunction, unreadVariable	Detects missing range checks and dead code that can hide validation defects.
UndefinedBehaviorSanitizer (UBSan)	In clang/gcc	-fsanitize=undefined,integer -fno-sanitize-recover	Traps integer divide-by-zero, signed overflows, invalid shifts at runtime.
Fuzzing (libFuzzer or AFL++)	latest	-fsanitize=address,undefined -fsanitize-coverage=trace-pc-guard	Generates adversarial inputs to exercise edge cases and validate error handling paths automatically.

Coding Standard 3

Coding Standard	Label	Name of Standard
String Correctness	STD-003-CPP	String Handling Safety. Improper string handling leads to buffer overflows, truncation, and data corruption. Using bounded operations and safe libraries prevents memory corruption and injection vulnerabilities.

Noncompliant Code

Copies user input into a fixed buffer without length checks, risking overflow.

```
#include <cstring>

void copyUserInput(const char* input) {
    char buf[16];
    strcpy(buf, input); // Unsafe: no bounds checking
}
```

Compliant Code

Uses strncpy_s (on MSVC) or std::string for automatic bounds management.

```
#include <string>
#include <iostream>

void copyUserInput(const std::string& input) {
    std::string buf = input.substr(0, 15); // Enforces safe length
    std::cout << buf << std::endl;
}
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	High	Medium	P1	Critical

Automation

Tool	Version	Checker	Description Tool
Compiler warnings	g++/clang++ /MSVC	-Wall -Wstringop-overflow -Wformat-security	Detects risky string operations and unsafe formatting.
clang-tidy	16+	cert-err34-c, cppcoreguidelines-pro-bounds-array-to-pointer-decay, bugprone-suspicious-string-compare	Flags unsafe string handling and pointer decay.
cppcheck	2.10+	warnings for bufferAccessOutOfBounds, dangerousFunction	Finds unsafe functions like strcpy, sprintf.
AddressSanitizer (ASan)	in clang/gcc	run with -fsanitize=address	Catches buffer overflows and use-after-free at runtime.
Coverity Scan	latest	Buffer overflows, string handling defects	Enterprise-grade static analysis for buffer and string vulnerabilities.

Coding Standard 4

Coding Standard	Label	Name of Standard
SQL Injection	STD-004-CPP	SQL Injection Prevention. Directly concatenating user input into SQL queries enables attackers to inject malicious commands. Using prepared statements and parameterized queries ensures that user input is treated as data, not executable SQL.

Noncompliant Code

Builds an SQL query string by concatenating untrusted user input, allowing injection.

```
#include <string>
#include <iostream>

std::string getUserData(const std::string& user) {
   std::string query = "SELECT * FROM users WHERE name = '" + user + "';";
   // If user = "admin' OR '1'='1" => entire table returned
   return query;
}
```

Compliant Code

Uses parameterized queries with bound variables, preventing injection.

```
#include <string>
#include <sqlite3.h>

std::string getUserDataSafe(sqlite3* db, const std::string& user) {
    sqlite3_stmt* stmt;
    const char* sql = "SELECT * FROM users WHERE name = ?;";

    sqlite3_prepare_v2(db, sql, -1, &stmt, nullptr);
    sqlite3_bind_text(stmt, 1, user.c_str(), -1, SQLITE_STATIC);

    // Execute safely without injection
    // sqlite3_step(stmt);
    sqlite3_finalize(stmt);

    return "Query executed safely";
}
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Critical	High	Low	P0	Critical

Automation

Tool	Version	Checker	Description Tool
SQLMap	latest	automated injection detection	Tests endpoints for SQL injection vulnerabilities.
clang-tidy custom checks	16+	bugprone-sql-injection (custom rulesets)	Detects unsafe concatenation into query strings.
Static Application Security Testing (SAST) tools (e.g., SonarQube)	latest	SQL injection rulesets	Analyzes source code for unsafe query construction.
Dynamic Application Security Testing (DAST)	latest	injection attack simulation	Sends crafted payloads to endpoints at runtime.
Database library built-in sanitizers (e.g., SQLite, ODBC, MySQL prepared statements)	current	enforce use of prepared statements	Native API prevents unsafe string concatenation.

Coding Standard 5

Coding Standard	Label	Name of Standard
Memory Protection	STD-005-CPP	Memory Protection. Improper memory management leads to leaks, dangling pointers, double frees, and exploitable heap corruption. Using RAII (Resource Acquisition Is Initialization) and smart pointers ensures memory safety by automating cleanup.

Noncompliant Code

Allocates memory manually without freeing it, causing leaks and potential corruption.

```
#include <iostream>

void leakMemory() {
    int* arr = new int[10];
    arr[0] = 42;
    std::cout << arr[0] << std::endl;
    // Memory is never freed
}
```

Compliant Code

Uses std::unique_ptr to automatically manage memory and prevent leaks.

```
#include <iostream>
#include <memory>

void safeMemory() {
    auto arr = std::make_unique<int[]>(10);
    arr[0] = 42;
    std::cout << arr[0] << std::endl;
    // Memory automatically freed when arr goes out of scope
}
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	High	Medium	P1	Critical

Automation

Tool	Version	Checker	Description Tool
Valgrind	Latest	memcheck	Detects memory leaks, double frees, invalid reads/writes
AddressSanitizer (ASan)	in clang/gcc	-fsanitize=address	Detects heap buffer overflows, use-after-free.
clang-tidy	16+	cppcoreguidelines-owning-memory, modernize-use-unique-ptr	Enforces RAII and warns about raw new/delete usage.
cppcheck	2.10+	leakMemory, unmatchedAllocDealloc	Finds missing frees and mismatched allocation/deallocation.
Coverity Scan	Latest	resource leaks, double free	Enterprise static analysis for memory corruption and leaks.

Coding Standard 6

Coding Standard	Label	Name of Standard
Assertions	STD-006-CPP	Assertion Usage. Assertions validate assumptions during development, catching logic errors early. They must not be used for input validation in production because they can be disabled, but should enforce invariants during testing and debugging.

Noncompliant Code

Uses an assertion for user input validation, which can be bypassed when assertions are disabled.

```
#include <cassert>
#include <iostream>

void divide(int x, int y) {
   assert(y != 0); // Unsafe: input validation via assert
   std::cout << x / y << std::endl;
}
```

Compliant Code

Uses runtime checks for user input, and assertions for internal invariants only.

```
#include <cassert>
#include <iostream>

void divide(int x, int y) {
    if (y == 0) {
        std::cerr << "Invalid divisor" << std::endl;
        return; // Safe runtime check
    }
    assert(x >= 0); // Internal invariant: expect non-negative numerator
    std::cout << x / y << std::endl;
}
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Medium	Low	P2	Moderate

Automation

Tool	Version	Checker	Description Tool
clang-tidy	16+	cert-dcl03-c, cert-err33-c	Flags inappropriate assertion use for external input validation.
cppcheck	2.10+	checkAssert	Detects ineffective or redundant assertions.
Compiler	g++/clang++	-DNDEBUG build mode disables asserts	Verifies assertions vanish in release builds.
Unit testing framework (e.g., GoogleTest)	ASSERT/EXPECT macros	ASSERT/EXPECT macros	Ensures invariants via test assertions instead of production code.
Coverity Scan	latest	assertion misuse, unchecked invariants	Finds logic flaws where asserts substitute for validation.

Coding Standard 7

Coding Standard	Label	Name of Standard
Exceptions	STD-007-CPP	Exception Handling. Improper exception handling can crash programs or expose sensitive state. Exceptions should be used for error recovery, not for control flow. Catch blocks must be specific and avoid swallowing errors silently.

Noncompliant Code

Catches all exceptions broadly and ignores them, leading to undefined state and silent failures.

```
#include <iostream>

void process() {
    try {
        throw std::runtime_error("Failure");
    } catch (...) {
        // Unsafe: swallows all exceptions
        std::cout << "Error ignored" << std::endl;
    }
}
```

Compliant Code

Catches specific exceptions and handles them appropriately, preserving program stability.

```
#include <iostream>
#include <stdexcept>

void process() {
    try {
        throw std::runtime_error("Failure");
    } catch (const std::runtime_error& e) {
        std::cerr << "Runtime error: " << e.what() << std::endl;
        // Handle error safely
    } catch (const std::exception& e) {
        std::cerr << "Unexpected error: " << e.what() << std::endl;
    }
}
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Medium	Medium	P1	High

Automation

Tool	Version	Checker	Description Tool
clang-tidy	16+	bugprone-exception-escape, cppcoreguidelines-pro-type-exceptionbase	Ensures proper exception specification and inheritance from std::exception.
cppcheck	2.10+	exceptionSafety	Detects missing exception handling and unsafe constructs.
Compiler warnings	g++/clang++ /MSVC	-fno-exceptions (for embedded builds) or -Wexceptions	Verifies exception support and flags disabled features inappropriately.
Unit test framework (GoogleTest, Catch2)	Latest	EXPECT_THROW, ASSERT_THROW	Validates that exceptions are thrown and handled correctly.
Coverity Scan	Latest	exception misuse	Identifies ignored exceptions and unsafe catch-all handlers.

Coding Standard 8

Coding Standard	Label	Name of Standard
Concurrency and Thread Safety	STD-008-CPP	Concurrency and Thread Safety. Improper synchronization of shared data leads to race conditions, deadlocks, and inconsistent program state. Using C++ standard threading primitives and avoiding manual low-level locking improves correctness and security.

Noncompliant Code

Multiple threads write to a shared variable without synchronization, causing a race condition.

```
#include <thread>
#include <vector>
#include <iostream>

int counter = 0;

void increment() {
    for (int i = 0; i < 1000; ++i) {
        counter++; // Data race
    }
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);
    t1.join();
    t2.join();
    std::cout << "Counter: " << counter << std::endl;
}
```

Compliant Code

Uses std::mutex to synchronize access to shared state, preventing data races.

```
#include <thread>
#include <vector>
#include <iostream>
#include <mutex>

int counter = 0;
std::mutex mtx;
```

Compliant Code

```
void increment() {
    for (int i = 0; i < 1000; ++i) {
        std::lock_guard<std::mutex> lock(mtx);
        counter++;
    }
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);
    t1.join();
    t2.join();
    std::cout << "Counter: " << counter << std::endl;
}
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Medium	Medium	P1	High

Automation

Tool	Version	Checker	Description Tool
clang ThreadSanitizer (TSan)	in clang/gcc	run with -fsanitize=thread	Detects race conditions and data races dynamically.
cppcheck	2.10+	threadSafety	Static detection of unsafe shared-state access.
clang-tidy	16+	cppcoreguidelines-owning-memory, concurrency-mt-unsafe	Warns about unsafe constructs in multithreaded code.
Helgrind (part of Valgrind)	Latest	race condition detection	Identifies data races and deadlocks at runtime.
Coverity Scan	Latest	concurrency/race condition rules	Flags unsynchronized access to shared data.

Coding Standard 9

Coding Standard	Label	Name of Standard
Input/Output File Handling Safety	STD-009-CPP	Input/Output File Handling Safety. Unvalidated file paths and unsafe file operations can lead to path traversal, information disclosure, or time-of-check/time-of-use (TOCTOU) races. Validate paths, operate within an allowlisted base directory, and use exceptions/RAII to ensure files are opened and closed safely.

Noncompliant Code

Concatenates untrusted user input into a path and uses std::ifstream directly. A relative path like ../../etc/passwd escapes the intended directory. It also does a separate existence check that can race with open (TOCTOU).

```
#include <fstream>
#include <string>
#include <sys/stat.h> // for ::stat (TOCTOU risk)

std::string readFile(const std::string& base, const std::string&
userPath) {
    std::string full = base + "/" + userPath;           // No validation,
traversal possible

    struct stat st{};
    if (::stat(full.c_str(), &st) != 0) {               // Separate
check, racy
        return {};
    }
    std::ifstream in(full);                              // Could open
unintended file
    if (!in) return {};
    std::string data((std::istreambuf_iterator<char>(in)),
                      std::istreambuf_iterator<char>());
    return data;
}
```

Compliant Code

Canonicalizes and validates the path against an allowlisted root using std::filesystem, then opens the file with exceptions enabled, avoiding TOCTOU and traversal.

```
#include <filesystem>
#include <fstream>
#include <string>

std::string readFileSafe(const std::filesystem::path& base,
                         const std::string& userPath) {
    namespace fs = std::filesystem;

    // Canonicalize both base and target, then ensure target stays within
base
    fs::path safeBase   = fs::weakly_canonical(base);
    fs::path requested  = fs::weakly_canonical(safeBase / userPath);

    if (requested.native().rfind(safeBase.native(), 0) != 0) {
        return {}; // Reject: escapes base (path traversal)
    }

    std::ifstream in;
    in.exceptions(std::ifstream::failbit | std::ifstream::badbit);
    try {
        in.open(requested, std::ios::binary); // Single open, no TOCTOU
        std::string data((std::istreambuf_iterator<char>(in)),
                          std::istreambuf_iterator<char>());
        return data;
    } catch (const std::ios_base::failure&) {
        return {}; // Handle open/read errors safely
    }
}
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Medium	Medium	P1	High

Automation

Tool	Version	Checker	Description Tool
CodeQL (C/C++)	Latest	path traversal, unsafe file open queries	Flags user-controlled path flows into file APIs.
Semgrep	Latest	C/C++ path traversal rules (c.path-traversal.*)	Detects concatenation of untrusted input into file paths.
SonarQube / SonarCloud	Latest	Path traversal, insecure file access rules	SAST to catch unvalidated paths and TOCTOU patterns.
clang-tidy	16+	bugprone-exception-safety, bugprone-throw-keyword-missing, custom path traversal checks	Ensures exceptions are enabled/handled around file I/O and supports custom checks for unsafe path building.
cppcheck	2.10+	portability and resource handling warnings	Finds missing error handling and suspicious file usage patterns.

Coding Standard 10

Coding Standard	Label	Name of Standard
Cryptography and Sensitive Data Handling	STD-010-CPP	Cryptography and Sensitive Data Handling. Hardcoding secrets, using weak algorithms, or mishandling sensitive buffers compromises confidentiality. Use modern cryptographic libraries, avoid deprecated ciphers, and ensure secrets are wiped from memory after use.

Noncompliant Code

Stores a plaintext password in the code and uses an insecure hash (MD5).

```
#include <iostream>
#include <string>
#include <openssl/md5.h>

void insecureStore() {
    std::string password = "SuperSecret123"; // Hardcoded secret
    unsigned char digest[MD5_DIGEST_LENGTH];
    MD5(reinterpret_cast<const unsigned char*>(password.c_str()),
        password.size(), digest); // Insecure MD5
    std::cout << "Hash stored" << std::endl;
}
```

Compliant Code

Uses a secure key derivation function (PBKDF2 with SHA-256) and ensures sensitive buffers are cleared after use.

```
#include <openssl/evp.h>
#include <string>
#include <vector>
#include <iostream>

void secureStore(const std::string& password, const std::string& salt) {
    const int iterations = 100000;
    const int keyLength = 32; // 256-bit

    std::vector<unsigned char> key(keyLength);
    PKCS5_PBKDF2_HMAC(password.c_str(), password.size(),
                      reinterpret_cast<const unsigned
char*>(salt.c_str()), salt.size(),
                      iterations, EVP_sha256(), keyLength, key.data());

    std::cout << "Derived key stored securely" << std::endl;

    // Wipe sensitive data from memory
    OPENSSL_cleanse(key.data(), key.size());
}
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

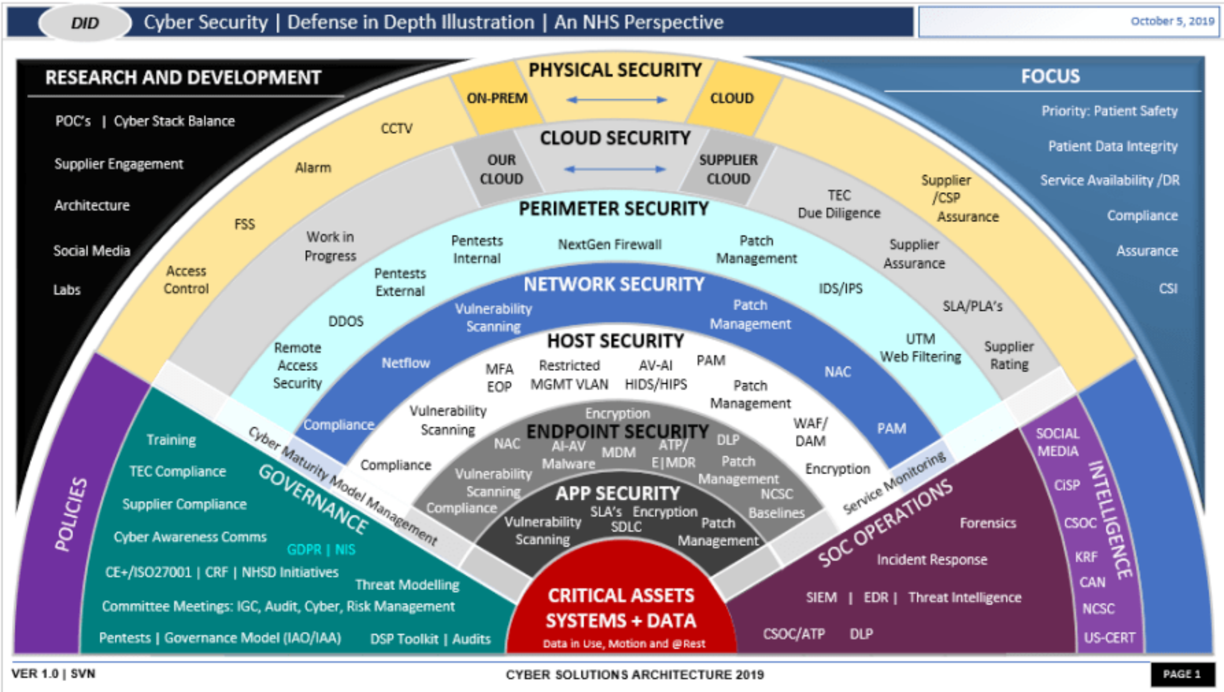
Severity	Likelihood	Remediation Cost	Priority	Level
Critical	High	Medium	P0	Critical

Automation

Tool	Version	Checker	Description Tool
CodeQL	Latest	insecure cryptography rules	Detects MD5/SHA1 usage, hardcoded secrets.
Semgrep	Latest	crypto rules (c.insecure-crypto.*)	Identifies insecure algorithms and weak random functions.
SonarQube / SonarCloud	Latest	hardcoded credentials, weak crypto rules	Flags insecure crypto API usage and embedded secrets.
truffleHog	Latest	secret scanning	Detects hardcoded passwords, API keys, and tokens in code.
clang-tidy (with custom checks)	16+	banned API list (MD5, DES)	Enforces allowlist of cryptographic primitives.

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

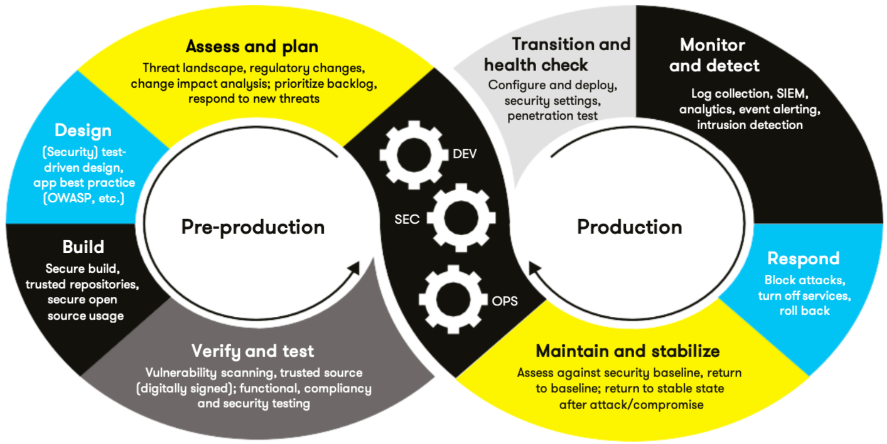
Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority,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

Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already maintains a mature DevOps infrastructure which provides a solid foundation for integrating DevSecOps practices that embed security controls directly into the continuous integration and deployment pipeline. The DevSecOps model builds upon the DevOps lifecycle (Plan, Code, Build, Test, Release, Deploy, and Operate) by inserting automated security activities into each phase. In the planning and design phase, threat modeling templates and policy-as-code linting are integrated into issue-tracking workflows so that every user story includes explicit security acceptance criteria and appropriate data classification. During coding, pre-commit and server-side Git hooks are introduced to automatically reject insecure APIs, unsafe string functions, and embedded credentials.

In the build phase, the compiler configuration is updated with strict warning flags (-Wall, -Wextra, -Wconversion, -Werror) and automated static analysis using clang-tidy and cppcheck. These scans run as part of the build job so that any violation of secure coding rules halts the pipeline. The testing phase includes sanitizers such as ASan, UBSan, and TSan, which run within automated test suites to detect memory, overflow, and concurrency defects, while fuzz testing validates parser resilience and API reliability. During packaging, software composition analysis scans dependency manifests, flags outdated or vulnerable components, and verifies license allow-lists before any artifact can be signed. The release process automatically generates a signed Software Bill of Materials and enforces CI/CD quality gates that block promotion if any critical security finding remains unresolved.

At the deployment stage, Kubernetes admission controllers validate container image signatures, enforce TLS configurations, and verify secret-management compliance. The operations environment employs runtime protection agents, log monitoring, and SIEM correlation rules to identify anomalies and security events. Incident metrics such as mean time to detect and mean time to recover are automatically fed back to engineering dashboards, creating measurable, auditable feedback loops. These additions modify Green Pace's existing DevOps model so that security tasks are not external reviews but automated, repeatable steps within the pipeline itself.

The DevSecOps diagram illustrates this integration: each ring of the DevOps infinity loop represents a continuous flow between development and operations, and DevSecOps inserts security checks within those feedback loops rather than treating them as gates or pauses. This ensures that every code change, test, and deployment passes through the same automated verification process for compliance and risk mitigation. Coding standards, encryption configurations, and Triple-A enforcement are defined as policy baselines within the CI/CD toolchain and deployed through infrastructure as code. This approach allows Green Pace to continuously verify compliance with its secure coding standards, producing a pipeline where every iteration enforces measurable security compliance (Seacord, 2013; Vehent, 2018).

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 Data Type Safety	High	Medium	Medium	High	4
STD-002-CPP Data Value Validation	High	Medium	Medium	High	4
STD-003-CPP String Handling Safety	Critical	High	Medium	P0	5
STD-004-CPP SQL Injection Prevention	Critical	High	Low	P0	5
STD-005-CPP Memory Protection	High	High	Medium	High	5
STD-006-CPP Assertion Usage	Medium	Medium	Low	P2	3
STD-007-CPP Exception Handling	High	Medium	Medium	High	4
STD-008-CPP Concurrency and Thread Safety	High	Medium	Medium	High	4
STD-009-CPP File Handling Safety	High	Medium	Medium	High	4
STD-010-CPP Cryptography and Sensitive Data	Critical	High	Medium	P0	5

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***.***

a. Explain each type of encryption, how it is used, and why and when the policy applies.
b. 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.

a. Encryption	Explain what it is and how and why the policy applies.
Encryption at rest	Encryption at rest protects stored data in databases, files, backups, volumes, and object storage. It is implemented with AES-256 in an authenticated mode like GCM using a managed key management service. Keys are never hardcoded. Access to keys is restricted using IAM roles and every access is logged. Rotation occurs at least annually or on suspected compromise. This policy applies to any medium that could hold sensitive or regulated information, including snapshots and replicas, because storage media can be lost, stolen, decommissioned, or misconfigured.
Encryption in flight	Encryption in flight protects data crossing any network boundary. It is implemented using TLS 1.2 or higher with approved cipher suites, HSTS on public endpoints, and mutual TLS for service-to-service traffic that handles sensitive data. Plaintext and obsolete protocols are disabled. This policy applies to client-to-service, service-to-service, and administrative sessions to prevent eavesdropping, tampering, and downgrade attacks during transmission.

a. Encryption	Explain what it is and how and why the policy applies.
Encryption in use	Encryption in use reduces the exposure of plaintext while data is processed in memory. Controls include minimizing the lifetime and scope of plaintext, zeroizing buffers immediately after use, avoiding paging secrets to disk, and leveraging secure enclaves or confidential computing where available. This policy applies whenever applications handle protected data in process memory because attackers may gain memory access through bugs, crashes, dumps, or insider misuse.

b. Triple-A Framework*	Explain what it is and how and why the policy applies.
Authentication	Authentication verifies the identity of users and services before any access is granted. It is implemented with a centralized identity provider, multi-factor authentication for human users, and short-lived credentials or mutual TLS for workloads. Shared accounts and static credentials are prohibited, and account lifecycle is tied to HR processes for automatic deprovisioning. This policy applies to all logins, API calls, CI jobs, administrative consoles, and machine-to-machine sessions to ensure only verified identities can interact with systems
Authorization	Authorization defines what an authenticated identity is allowed to do. It is implemented with least-privilege role-based access control and policy-as-code. Every request is evaluated against explicit allow rules and the default outcome is deny. Service accounts are narrowly scoped to a single application and environment. This policy applies at network, application, and data layers to prevent unauthorized read, write, execute, or admin actions.
Accounting	Accounting records and preserves auditable evidence of critical actions. Systems log authentication events, privilege changes, configuration edits, and access to sensitive data with timestamp, subject, object, action, result, and correlation identifiers. Logs are encrypted, tamper-evident, centralized in a SIEM, and retained according to compliance requirements. Alerts are configured for high-risk patterns like repeated failed MFA, role escalation, and denied access spikes. This policy applies to all production systems and administrative activities so investigations, compliance checks, and continuous monitoring are possible.

*****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.

Standard	Mapped Principles	Justification
STD-001-CPP Data Type Safety	Principle 1: Validate Input Data Principle 2: Heed Compiler Warnings Principle 10: Adopt a Secure Coding Standard	Using strong data types and explicit casting enforces compile-time validation and prevents overflow or truncation. Compiler warnings are treated as errors, ensuring early detection of unsafe operations. Adhering to standardized secure-coding guidelines aligns development with industry-approved best practices.
STD-002-CPP Data Value Validation	Principle 1: Validate Input Data Principle 4: Keep It Simple Principle 9: Use Effective QA	Input range checks guarantee that only legitimate data values reach the logic layer. Simplicity of validation routines prevents complexity-related defects. Continuous testing, including fuzzing and boundary analysis, ensures predictable system responses.
STD-003-CPP String Handling Safety	Principle 1: Validate Input Data Principle 6: Least Privilege Principle 9: Effective QA Principle 10: Secure Coding Standard	Validated and bounded string operations prevent buffer overflows. Access to string buffers is limited to authorized components to reduce exposure. Automated tests detect regression, and standardized library functions replace dangerous routines.
STD-004-CPP SQL Injection Prevention	Principle 1: Validate Input Data Principle 5: Default Deny Principle 8: Defense in Depth	Parameterized queries and ORM layers treat input as inert data. Default-deny rules block unvalidated statements. Multiple layers—input sanitation, prepared statements, and database permissions—jointly eliminate injection vectors.
STD-005-CPP Memory Protection	Principle 8: Defense in Depth Principle 9: Effective QA Principle 10: Secure Coding Standard	Smart pointers, ownership models, and static analysis collectively prevent leaks and dangling references. Defense-in-depth ensures failures in one control are mitigated by others. Testing tools such as ASan and Valgrind confirm runtime integrity.
STD-006-CPP Assertion Usage	Principle 3: Architect and Design for Security Principle 4: Keep It Simple Principle 9: Effective QA	Assertions confirm expected internal states while runtime checks handle user input. Simpler logic paths minimize hidden assumptions. Quality-assurance tests ensure assertions are safe and meaningful, avoiding accidental suppression of real errors.
STD-007-CPP Exception Handling	Principle 3: Architect and Design for Security Principle 4: Keep It Simple Principle 8: Defense in Depth	The architecture isolates fault domains so exceptions do not propagate unpredictably. Explicit catch clauses maintain clarity, and layered recovery mechanisms maintain system stability even when faults occur.
STD-008-CPP Concurrency and Thread Safety	Principle 3: Architect and Design for Security Principle 8: Defense in Depth Principle 9: Effective QA	Secure design mandates safe synchronization constructs and immutable shared data. Defense-in-depth requires multiple safeguards such as mutexes and atomics. Thread-sanitizer testing validates thread-safe behavior under stress.
STD-009-CPP File Handling Safety	Principle 1: Validate Input Data Principle 5: Default Deny Principle 8: Defense in Depth	Input validation ensures file paths are canonicalized. Default-deny file access blocks unlisted directories. Layered defense includes OS permissions, sandboxing, and input normalization, all of which mitigate traversal and privilege-escalation risks.
STD-010-CPP Cryptography and Sensitive Data	Principle 3: Architect and Design for Security Principle 5: Default Deny Principle 6: Least Privilege Principle 8: Defense in Depth	Secure design mandates modern cryptography and strong key management. Default-deny ensures unauthorized entities cannot access secrets. Least privilege restricts key usage, while multiple cryptographic layers protect data throughout its lifecycle.

Narrative Summary

Mapping principles to standards demonstrates the traceability between Green Pace's security philosophy and its practical coding enforcement. Each standard supports multiple principles, illustrating that secure design is holistic rather than isolated. Validation, simplicity, least privilege, and layered defense collectively ensure that vulnerabilities are mitigated at design time, verified during development, and continuously enforced through automation. This mapping also provides auditors with clear evidence that each coding rule aligns with widely recognized security foundations.

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	October 1 2025	Initial creation of Green Pace Security Policy for CS-405 Project One submission. Includes 10 coding standards, encryption and Triple-A policies, and mapped principles.	Gonzalo Patino	Gonzalo Patino
[Insert text.]	October 2nd, 2025	Added summary of risk assessments and clarified DevSecOps automation narrative.	Gonzalo Patino	Gonzalo Patino
[Insert text.]	October 12th, 2025	Completed policy version with mapped principles, encryption, and Triple-A framework. Added version control record and reflection section.	Gonzalo Patino	Gonzalo Patino

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV

Final Reflection

Developing the Green Pace Security Policy helped me understand that secure coding is not limited to writing safer functions. It is a mindset that affects every part of software design, development, and deployment. Translating abstract principles into measurable coding standards showed how each engineering decision influences overall system security. I gained a clearer appreciation for the role of input validation, least privilege, and defense in depth as practical habits that reduce real risks in code.

The project also taught me that automation is essential for lasting security. By using static analysis, secret scanning, and secure build pipelines, I saw how compliance becomes automatic and consistent instead of relying only on human review. Embedding policy-as-code and continuous integration practices transforms DevOps into DevSecOps, making security an active part of every commit, build, and release cycle.

Finally, I learned that writing security policy is just as important as coding. Documenting principles, mapping risks, and defining measurable controls help developers, auditors, and stakeholders understand the purpose behind each safeguard. This exercise strengthened my ability to think like both a software engineer and a security professional. I now see that clarity, structure, and accountability are what turn secure coding theory into reliable, verifiable practice.

References

- Vehent, J. (2018). *Securing DevOps: Security in the cloud*. Manning Publications.

- Federal Trade Commission. (2015). *Start with security: A guide for business*. https://www.ftc.gov/business-guidance/resources/start-security-guide-business

- Linford & Company. (n.d.). *Information security policies: Why they are important to your organization*. https://linfordco.com/blog/information-security-policies/

- Mend.io. (2024). *Secure coding*. https://www.mend.io/blog/secure-coding/

- Seacord, R. C. (2013). *Secure coding in C and C++* (2nd ed.). Addison-Wesley Professional.