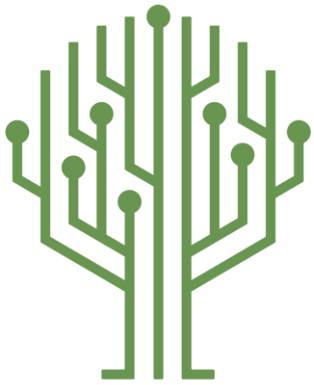
Green Pace Developer: Security Policy Guide Template



Green Pace Secure Development Policy

Contents

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.

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	Validating input data ensures that all external or user-provided inputs conform to expected formats, ranges, and types. This helps prevent common attacks such as buffer overflows, SQL injection, or other malicious exploits that rely on malformed or out-of-range data. Proper validation includes checking length limits, character sets, numeric ranges, and overall integrity before processing the input.
2. Heed Compiler Warnings	Compiler and static analysis warnings often flag dangerous coding patterns or undefined behavior. By correcting all warnings—rather than ignoring them—you reduce the chance of introducing subtle bugs or security vulnerabilities. Treating warnings as errors during development enforces stricter coding discipline and helps ensure more robust, secure applications.
3. Architect and Design for Security Policies	Security should be integrated into system architecture from the start, aligning with defined security requirements and threat models. By designing around security policies—for instance, enforcing role-based access control or encrypting sensitive data at rest—you create a robust foundation that reduces the likelihood of vulnerabilities creeping in later. This principle emphasizes a proactive design approach, rather than treating security as an afterthought.

Principles	Write a short paragraph explaining each of the 10 principles of security.
4. Keep It Simple	Complex code and intricate system designs are harder to verify and maintain, increasing the likelihood of errors or overlooked vulnerabilities. By keeping solutions as straightforward as possible—both at the code and architecture levels—you reduce the attack surface and simplify testing, code reviews, and auditing. Simplicity also helps future developers understand and securely update the system.
5. Default Deny	Under the "default deny" principle, the system initially grants no permissions or access. Users, processes, or components must then be explicitly given only the minimum permissions needed to function. This approach prevents inadvertent exposure of resources and helps ensure that newly introduced features don't accidentally open broader access than intended.
6. Adhere to the Principle of Least Privilege	Building on "default deny," each user or component operates with only the privileges strictly necessary to accomplish its task. If that component or user is compromised, the damage remains limited since it cannot access or modify resources outside its minimal scope. Least privilege applies to every layer of the system—users, processes, services, and network connections.
7. Sanitize Data Sent to Other Systems	When data leaves your system (e.g., via an API call, database query, or file output), sanitizing or encoding it prevents injection attacks and ensures it's interpreted safely by the receiving system. By escaping special characters and enforcing strict output formats, you prevent cross-site scripting (XSS), SQL injection, command injection, and other exploits that rely on unvalidated outbound data.
8. Practice Defense in Depth	Defense in Depth employs multiple layers of security controls—like input validation, access control, encryption, network segmentation, and intrusion detection—so that if one layer fails, others can still protect the system. This layered approach makes compromising the system significantly more difficult, as an attacker must bypass multiple independent defenses.
9. Use Effective Quality Assurance Techniques	Secure code is closely tied to high-quality code. Robust QA includes code reviews, static analysis tools, automated testing, unit tests, fuzz testing, and continuous integration checks. These techniques help

Principles	Write a short paragraph explaining each of the 10 principles of security.
	identify potential flaws early, reducing costs and preventing security holes from reaching production environments.
10. Adopt a Secure Coding Standard	A secure coding standard provides clear, consistent guidelines for writing safe, maintainable code. By following standards such as CERT C++ or MISRA C++, development teams reduce the likelihood of introducing known vulnerabilities. These standards codify best practices for memory handling, error checking, exception use, and other common pitfalls, ensuring a uniform and secure approach throughout the project.

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	Enforce Consistent Integer Usage

Noncompliant Code

In this example, a **signed** integer (int) is compared against an **unsigned** integer (std::size_t), which can result in unexpected wrap-around behavior if the signed value is negative.

```
    // Noncompliant code (12-point Courier New indentation)
    #include <iostream>
    #include <cstddef>

    int main() {
        int index = -5;         // signed integer
        std::size_t size = 10; // unsigned integer

        // The comparison below is problematic when 'index' is negative
        // because 'index' is implicitly converted to an unsigned type,
        // resulting in a very large value rather than -5.
        if (index < size) {
            std::cout << "Index is within range!\n";
        } else {
            std::cout << "Index is out of range!\n";
        }

        return 0;
    }
```

Compliant Code

A safer approach is to ensure both operands are of the same, appropriate type. For instance, store index as an unsigned type if negative values are never valid, or else perform explicit range checks before comparing.

```
    // Compliant code (12-point Courier New indentation)
    #include <iostream>
    #include <cstddef>

    int main() {
        // Option 1: Use a signed type consistently, and check for
negatives
        int index = -5;
```

Compliant Code

```
        int arraySize = 10;

        // Explicitly check if index is valid (not negative)
        if (index < 0 || index >= arraySize) {
            std::cout << "Index is out of range!\n";
        } else {
            std::cout << "Index is within range!\n";
        }

        // Option 2: If negatives are never valid, use unsigned
        // size_t index2 = 5;
        // size_t size2 = 10;
        // if (index2 < size2) { ... }

        return 0;
    }
```

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

Principles(s):
1. **Heed Compiler Warnings:** Many compilers generate warnings when comparing signed and unsigned values. Fixing those warnings prevents potential security risks and logic errors.
2. **Validate Input Data:** If index originates from user input, verifying it's non-negative (and within bounds) ensures valid, safe array access.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Common	Low	High	Mandatory

Automation

Tool	Version	Checker	Description Tool
clang-tidy	14.0	readability-* / bugprone-*	Flags mismatched types and suspicious comparisons.
cppcheck	2.8	warning	Detects possible type errors and implicit conversions.
SonarQube	9.0	cpp:SXXX	Identifies code smells and bugs related to type usage in C/C++

Tool	Version	Checker	Description Tool
			projects.
Visual Studio Code	2022	MSVC Analyzer	Issues warnings and suggestions for suspicious type conversions in C++ code.

Coding Standard 2

Coding Standard	Label	Name of Standard
Data Value	STD-002-CPP	Validate Data Ranges

Noncompliant Code

This code reads an integer from user input and directly uses it as an array index without checking if it is **within** valid bounds, risking **out-of-bounds** access.

```
    // Noncompliant code (12-point Courier New indentation)
    #include <iostream>

    int main() {
        int userIndex;
        int data[5] = {10, 20, 30, 40, 50};

        std::cout << "Enter an index: ";
        std::cin >> userIndex;

        // No validation performed; negative or large values cause
undefined behavior
        std::cout << "Value at index = " << data[userIndex] << std::endl;

        return 0;
    }
```

Compliant Code

Here, the program **validates** the user input, ensuring userIndex is between **0** and **4 before** using it to access the array. If the index is invalid, an error message is displayed.

```
    // Compliant code (12-point Courier New indentation)
    #include <iostream>

    int main() {
        int userIndex;
        int data[5] = {10, 20, 30, 40, 50};

        std::cout << "Enter an index: ";
        std::cin >> userIndex;

        // Check if userIndex is within the valid range for data[]
```

Compliant Code

```
        if (userIndex < 0 || userIndex >= 5) {
            std::cerr << "Error: Index out of bounds." << std::endl;
        } else {
            std::cout << "Value at index = " << data[userIndex] <<
std::endl;
        }

        return 0;
    }
```

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

Principles(s):
1. **Validate Input Data:** This standard directly enforces input validation by ensuring user-entered (or otherwise untrusted) data is checked for validity.
2. **Keep It Simple:** By adding straightforward checks, we keep the logic clear and reduce the chance of unexpected behavior from bad input.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Common	Low	High	Mandatory

Automation

Tool	Version	Checker	Description Tool
clang-tidy	14.0	bugprone-implicit-*	May warn about unverified user input usage in array or pointer expressions.
cppcheck	2.8	warning	Detects suspicious indices or unchecked user input.
Coverity	2022	UNINIT / OVERRUN	Identifies potential array overruns or unvalidated parameters.
SonarQube	9.0	cpp:SXXX	Scans for places where input is used without proper checks, flags possible OOB.

Coding Standard 3

Coding Standard	Label	Name of Standard
String Correctness	STD-003-CPP	Safe String Handling

Noncompliant Code

Here, the program uses strcpy with no length check, allowing a longer source string to overflow the destination buffer.

```
    // Noncompliant code (12-point Courier New indentation)
    #include <iostream>
    #include <cstring>

    int main() {
        char destination[10];
        const char* source = "ThisIsTooLong";

        // No boundary check, may overwrite memory beyond 'destination'
        std::strcpy(destination, source);

        std::cout << "Destination: " << destination << std::endl;
        return 0;
    }
```

Compliant Code

We switch to std::strncpy and limit the length, ensuring the destination buffer cannot be overrun. Alternatively, using std::string fully would also avoid manual buffer management.

```
    // Compliant code (12-point Courier New indentation)
    #include <iostream>
    #include <cstring>

    int main() {
        char destination[10];
        const char* source = "ThisIsTooLong";

        // Copy up to 'destination' size - 1, then manually add null
terminator
        std::strncpy(destination, source, sizeof(destination) - 1);
        destination[sizeof(destination) - 1] = '\0';
```

Compliant Code

```
        std::cout << "Destination (truncated safely): " << destination <<
std::endl;
        return 0;
    }
```

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

Principles(s):
1. **Validate Input Data:** Even with strings, ensuring we only copy as many characters as the buffer can handle is a form of validation.
2. **Heed Compiler Warnings:** Many compilers warn about unsafe functions like strcpy; listening to these warnings helps prevent this vulnerability.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Common	Low	High	Mandatory

Automation

Tool	Version	Checker	Description Tool
cppcheck	2.8	warning / memleak	Detects unsafe string operations, potential buffer overflow in C/C++.
clang-tidy	14.0	cert-* (CERT checks)	Has checks specifically warning about unsafe strcpy, etc.
SonarQube	9.0	cpp:SXXX	Finds risky string usage, flags possible out-of-bounds writes.
Visual Studio	2022	MSVC Warnings	Warns about unsafe CRT functions (strcpy, sprintf) with suggestions.

Coding Standard 4

Coding Standard	Label	Name of Standard
SQL Injection	STD-004-CPP	Avoid Direct String Concatenation for Queries

Noncompliant Code

This example builds a query using direct string concatenation with user input. An attacker could append malicious SQL to userInput, altering the query's logic.

```
    // Noncompliant code (12-point Courier New indentation)
    #include <iostream>
    #include <string>

    int main() {
        std::string userInput;
        std::cout << "Enter a username: ";
        std::getline(std::cin, userInput);

        // Dangerous: direct concatenation can lead to injection
        std::string sqlQuery = "SELECT * FROM USERS WHERE NAME = '" +
userInput + "';";

        // Imagine we pass 'sqlQuery' directly to a SQL library here...
        std::cout << "Executing Query: " << sqlQuery << std::endl;

        return 0;
    }
```

Compliant Code

Use a **parameterized** or **prepared statement** so that user input is sent to the database separately, preventing injection.

```
    // Compliant code (12-point Courier New indentation)
    #include <iostream>
    #include <string>
    // Hypothetical database API

    // Pseudocode for a parameterized query
    bool runSafeQuery(const std::string &userName) {
        // Prepare statement with placeholder
        // In a real library: "SELECT * FROM USERS WHERE NAME = ?"
        // Then bind userName to the placeholder, preventing injection
```

Compliant Code

```
        return true; // Stub
    }

    int main() {
        std::string userInput;
        std::cout << "Enter a username: ";
        std::getline(std::cin, userInput);

        // Use parameterized queries or a prepared statement
        if (!runSafeQuery(userInput)) {
            std::cerr << "Query execution failed.\n";
        }

        return 0;
    }
```

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

Principles(s):
1. **Sanitize Data Sent to Other Systems:** SQL injection is fundamentally about untrusted input. By sanitizing or parameterizing, we ensure the database receives only valid data.
2. **Practice Defense in Depth:** Even if there's some filtering upstream, using parameterized queries adds another protective layer against injection.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Critical	Very Common	Moderate	Highest	Mandatory

Automation

Tool	Version	Checker	Description Tool
SonarQube	9.0	cpp:S2076	Flags potential injection vulnerabilities when queries are built via strings.
Fortify SCA	21.x	SQL Injection	Analyzes code for untrust concatenated into SQL s
Veracode Static	2022	SQLi	Recognizes direct SQL string building from user input.

Tool	Version	Checker	Description Tool
clang-tidy	14.0	misc-*	May identify suspicious string concatenations for external commands.

Coding Standard 5

Coding Standard	Label	Name of Standard
Memory Protection	STD-005-CPP	Safe Resource Allocation & Release

Noncompliant Code

This code uses new[] to allocate an array but accidentally calls delete (not delete[]), leading to undefined behavior.

```
// Noncompliant code (12-point Courier New indentation)
#include <iostream>

int main() {
    // Allocate array with new[]
    int* numbers = new int[5];

    // ... some operations on numbers ...

    // Incorrectly use delete instead of delete[]
    delete numbers;   // Noncompliant: Should be delete[]

    return 0;
}
```

Compliant Code

We correctly match new[] with delete[], or better yet, use a smart container (e.g., std::vector) for automatic memory management.

```
// Compliant code (12-point Courier New indentation)
#include <iostream>
#include <vector>

int main() {
    // Option 1: Fix the mismatch
    int* numbers = new int[5];
    // ... use numbers ...
    delete[] numbers;  // Correct usage

    // Option 2: Use std::vector for RAII
    std::vector<int> nums(5);
    // ... safe usage with no manual delete ...

    return 0;
```

Compliant Code

```
    }
```

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

Principles(s):
1. **Keep It Simple:** RAII containers like std::vector and std::unique_ptr simplify ownership rules and reduce memory-management complexity.
2. **Validate Input Data:** Although less direct, ensuring memory usage corresponds to valid data is another line of defense against bad pointers or indexes.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Common	Low / Medium	High	Mandatory

Automation

Tool	Version	Checker	Description Tool
clang-tidy	14.0	cppcoreguidelines-*	Warns about mismatched allocation/deallocation, recommends smart pointers/containers.
ASan (AddressSanitizer)	N/A	Runtime Instrumentation	Detects memory misuse (double free, wrong delete).
cppcheck	2.8	memleak, mismatch	Identifies memory leaks, mismatch between new and delete[].
Valgrind	3.17	Memcheck	At runtime, detects invalid frees, leaks, use-after-free, etc.

Coding Standard 6

Coding Standard	Label	Name of Standard
Assertions	STD-006-CPP	Use Assertions Appropriately

Noncompliant Code

This code uses assert to validate user input (e.g., a user-supplied array index). If index is invalid, the assert triggers in production builds, potentially aborting the program rather than gracefully handling the error.

```
    // Noncompliant code (12-point Courier New indentation)
    #include <iostream>
    #include <cassert>

    int main() {
        int index;
        std::cout << "Enter an index: ";
        std::cin >> index;

        // Using an assertion for user input is noncompliant for
production
        assert(index >= 0 && index < 5 && "Index out of range");

        // If assertion fails, program aborts in debug mode
        // In release mode, the assert might be stripped out, ignoring
the error

        std::cout << "Valid index entered." << std::endl;
        return 0;
    }
```

Compliant Code

The code uses explicit checks for user input errors and handles them gracefully rather than relying on assert.

```
    // Compliant code (12-point Courier New indentation)
    #include <iostream>
    #include <stdexcept>

    int main() {
        int index;
        std::cout << "Enter an index: ";
```

Compliant Code

```
        std::cin >> index;

        if (index < 0 || index >= 5) {
            std::cerr << "Error: Index out of range. Please try again."
<< std::endl;
            return 1; // or prompt user again
        }

        std::cout << "Valid index entered." << std::endl;
        return 0;
    }
```

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

Principles(s):
1. **Architect and Design for Security Policies**: Proper error handling design ensures the system handles invalid inputs gracefully rather than crashing.
2. **Use Effective Quality Assurance Techniques**: Assertions are useful in testing or debugging to catch logic mistakes early, but not for normal runtime checks.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Common	Low	Medium	Recommended

Automation

Tool	Version	Checker	Description Tool
clang-tidy	14.0	readability-assert	Warns if assert is used for conditions that might be user-driven.
cppcheck	2.8	assertUsage	Flags suspicious or misused assertions.
SonarQube	9.0	cpp:SXXX	Can detect places where asserts conflict with production code usage.
Manual Code Review	N/A	N/A	Assertions usage often needs developer scrutiny to confirm intent.

Coding Standard 7

Coding Standard	Label	Name of Standard
Exceptions	STD-007-CPP	Proper Exception Handling

Noncompliant Code

This code throws and catches a **generic** ... exception for normal control flow, making it difficult to distinguish real errors from logic branches.

```
    // Noncompliant code (12-point Courier New indentation)
    #include <iostream>

    int main() {
        try {
            // Normal operation that isn't exceptional
            throw 1; // Using exceptions as normal flow is noncompliant
        } catch (...) {
            // Catch-all swallows every exception type, even critical
ones
            std::cout << "An exception occurred, but we are ignoring its
type.\n";
        }
        return 0;
    }
```

Compliant Code

We throw **specific** exceptions only on truly exceptional conditions and catch them in typed handlers, ensuring we handle or log the exact error meaning.

```
    // Compliant code (12-point Courier New indentation)
    #include <iostream>
    #include <stdexcept>

    void riskyOperation(bool doFail) {
        if (doFail) {
            // Throw a specific exception type
            throw std::runtime_error("Operation failed unexpectedly.");
        }
    }

    int main() {
        try {
```

Compliant Code

```
            riskyOperation(true);
        } catch (const std::runtime_error& e) {
            std::cerr << "Runtime error: " << e.what() << std::endl;
        } catch (...) {
            // Rare fallback if something else is thrown
            std::cerr << "Unknown exception occurred.\n";
        }
        return 0;
    }
```

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

Principles(s):

- **Architect and Design for Security Policies:** A well-defined exception strategy helps ensure robust error handling consistent with security policies.
- **Use Effective Quality Assurance Techniques:** Properly targeted exceptions and typed catches facilitate debugging and testing of erroneous conditions.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Occasional	Medium	Medium	Recommended

Automation

Tool	Version	Checker	Description Tool
clang-tidy	14.0	modernize-avoid-c-arrays, hicpp-exception-baseclass	Points out certain poor exception practices.
SonarQube	9.0	cpp:S112	Detects catch-all used incorrectly, recommends typed catches.
cppcheck	2.8	warning	May highlight trivial exceptions or overly broad catches.
Manual Code Review	N/A	N/A	Verifying exception usage is often done with design reviews.

Coding Standard 8

Coding Standard	Label	Name of Standard
Concurrency / Race Conditions	STD-008-CPP	Avoid Data Races & Synchronize Threads

Noncompliant Code

Two threads access and modify a shared counter without synchronization, leading to undefined behavior and possible data corruption.

```
// Noncompliant code (12-point Courier New indentation)
#include <thread>
#include <iostream>

int sharedCounter = 0;

void increment() {
    for(int i=0; i<1000; i++) {
        // No synchronization
        sharedCounter++;
    }
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);
    t1.join();
    t2.join();

    // Result is unpredictable; data race occurs
    std::cout << "Final Counter: " << sharedCounter << std::endl;
    return 0;
}
```

Compliant Code

We protect the shared counter with a mutex or use an atomic integer, ensuring increments are safely performed across threads.

```
// Compliant code (12-point Courier New indentation)
#include <thread>
#include <iostream>
#include <atomic>
```

Compliant Code

```
    std::atomic<int> sharedCounter{0};

    void increment() {
        for(int i=0; i<1000; i++) {
            // Atomic increment
            sharedCounter.fetch_add(1, std::memory_order_relaxed);
        }
    }

    int main() {
        std::thread t1(increment);
        std::thread t2(increment);
        t1.join();
        t2.join();

        // Consistent result guaranteed
        std::cout << "Final Counter: " << sharedCounter.load() <<
std::endl;
        return 0;
    }
```

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

Principles(s):
1. **Practice Defense in Depth**: Even if you have some concurrency checks, using atomic or locked operations ensures no single omission leads to a catastrophic race.
2. **Keep It Simple**: Simplifying concurrency design (e.g., using higher-level concurrency patterns) reduces race condition risk.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Common	Medium	High	Mandatory

Automation

Tool	Version	Checker	Description Tool
Thread Sanitizer	N/A	Runtime Checker	Detects data races, lock order inversions, and concurrency issues at runtime.
clang-tidy	14.0	cert-* concurrency checks	Identifies suspicious shared data usage, missing

Tool	Version	Checker	Description Tool
			synchronization.
Coverity	2022	MULTITHREAD / RACE_CONDITION	Analyzes code for possible race conditions in multi-threaded code.
cppcheck	2.8	warning (experimental)	Offers some concurrency warnings.

Coding Standard 9

Coding Standard	Label	Name of Standard
Hardcoded Secrets	STD-009-CPP	Prohibit Storing Credentials in Code

Noncompliant Code

This code includes a hardcoded database password in plain text, making it visible to anyone with repository access or compiled binary strings.

```
    // Noncompliant code (12-point Courier New indentation)
    #include <iostream>

    const char* DB_PASSWORD = "SuperSecretPass123";

    int main() {
        std::cout << "Connecting with password: " << DB_PASSWORD <<
std::endl;
        // ... connect to database ...
        return 0;
    }
```

Compliant Code

Credentials are **not** hardcoded; the code retrieves the password from an environment variable or secure configuration file at runtime.

```
    // Compliant code (12-point Courier New indentation)
    #include <iostream>
    #include <cstdlib> // for std::getenv

    int main() {
        const char* dbPass = std::getenv("DB_PASSWORD");
        if (!dbPass) {
            std::cerr << "Error: DB_PASSWORD environment variable not
set.\n";
            return 1;
        }

        std::cout << "Connecting with password: [HIDDEN]" << std::endl;
        // ... connect using dbPass ...
        return 0;
    }
```

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

Principles(s):
1. **Default Deny:** Restricting password usage to runtime-supplied secrets enforces a more secure environment.
2. **Use Effective Quality Assurance Techniques:** Tools scanning your repo can flag if any secret was accidentally committed, preventing exposure.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Critical	High	Low / Medium	Highest	Mandatory

Automation

Tool	Version	Checker	Description Tool
GitLeaks	8.0	Secrets detection	Scans repositories for common secret patterns, incl. passwords.
SonarQube	9.0	Sensitive Credentials (cpp:S2068)	Detects suspicious hardcoded strings or credentials.
truffleHog	3.x	Regex, entropy checks	Finds high-entropy or known patterns in code.
Manual Code Review	N/A	N/A	Evaluate if any new code inadvertently includes secrets.

Coding Standard 10

Coding Standard	Label	Name of Standard
Logging	STD-010-CPP	Standardize Safe Logging Practices

Noncompliant Code

This code logs **plaintext passwords** and other sensitive fields, which could be read by anyone with access to the log files.

```
    // Noncompliant code (12-point Courier New indentation)
    #include <iostream>
    #include <string>

    void loginUser(const std::string& username, const std::string&
password) {
        // Logging sensitive data
        std::cout << "[LOG] Attempting login with user=" << username
                  << " and password=" << password << std::endl;
        // ... authentication logic ...
    }

    int main() {
        loginUser("Alice", "SuperSecret123");
        return 0;
    }
```

Compliant Code

We **mask** or omit sensitive information in logs while still noting important events, preserving audit trails without exposing secrets.

```
    // Compliant code (12-point Courier New indentation)
    #include <iostream>
    #include <string>

    void loginUser(const std::string& username, const std::string&
password) {
        // Only log the fact that a login attempt happened
        std::cout << "[LOG] Login attempt for user=" << username
                  << ", password=********" << std::endl;
        // ... authentication logic ...
    }

    int main() {
```

Compliant Code

```
        loginUser("Alice", "SuperSecret123");
        return 0;
    }
```

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

Principles(s):
1. **Default Deny:** By default, deny logging sensitive data and only allow sanitized or masked fields.
2. **Use Effective Quality Assurance Techniques:** Periodic log reviews and scanning ensure no new code logs secrets.

Threat Level

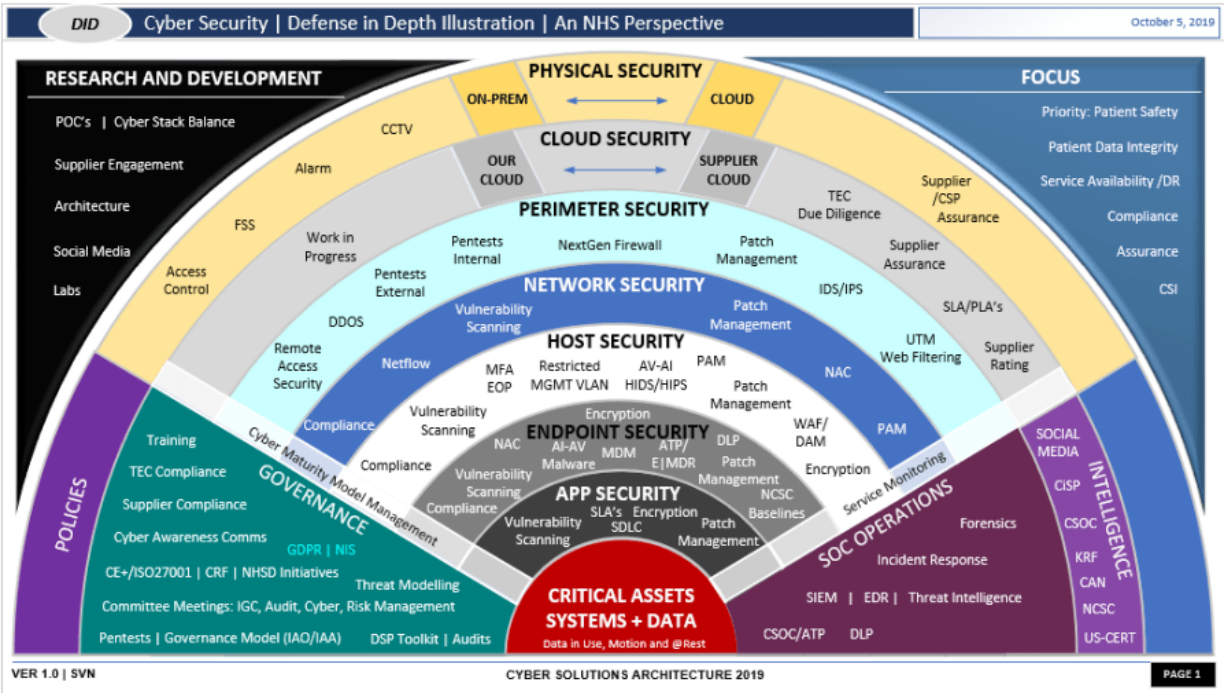
Severity	Likelihood	Remediation Cost	Priority	Level
High	Very Common	Low	High	Mandatory

Automation

Tool	Version	Checker	Description Tool
SonarQube	9.0	Security Hotspot	Flags potential logging of sensitive data.
Splunk or ELK	N/A	Log scanning scripts	Scripts that detect suspicious patterns (like "password=") in logs.
GitLeaks	8.0	Secrets detection	May catch log statements with keywords like "PASSWORD".
Manual Code Review	N/A	N/A	Validate that logs do not reve confidential info.

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.

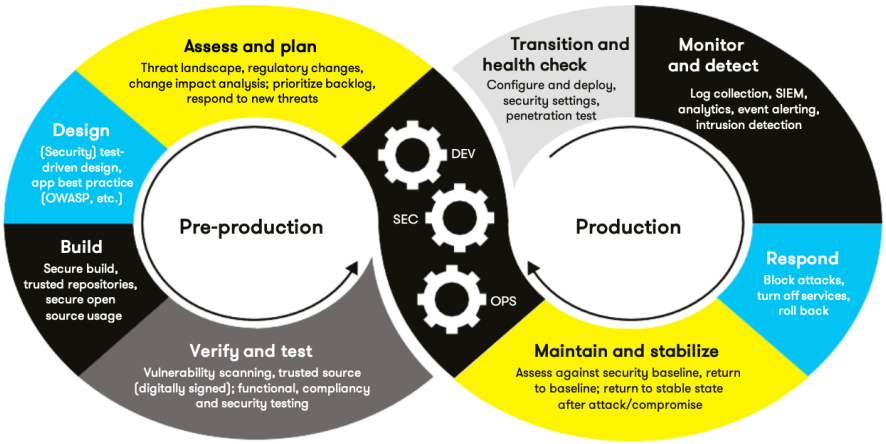
Automated Detection

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.

1. **Assess and Plan** (Pre-production):
    - **Threat Modeling:** During the design and backlog planning stage, incorporate these coding standards as acceptance criteria. For each user story or feature, specify compliance with the relevant standard (e.g., no hardcoded credentials, no unsafe string operations).
2. **Design**
    - Apply **secure design** best practices from OWASP or CERT C++ at the architecture level. For instance, ensuring **Memory Protection** standard is part of the design if the system deals with large buffers or concurrency.
3. **Build**
    - **Automated Tools:** Integrate static analysis (clang-tidy, cppcheck, SonarQube, etc.) into the **build pipeline**. If the code violates a standard (e.g., using strcpy or direct SQL concatenation), the pipeline can **fail** or generate a **warning** for the developer to fix before merging.
4. **Verify and Test**
    - **Unit Tests and Security Tests:** Write test cases that check for memory safety (AddressSanitizer), concurrency correctness (ThreadSanitizer), or injection vulnerabilities (fuzzing test harnesses).

   - Ensure the environment also runs **compliance scans** that match each coding standard.
5. **Transition and Health Check**
   - Before moving to **Production**, run final **penetration tests**, **configuration checks**, and **secure settings** reviews. Confirm that **no** outstanding coding standard violations exist in high-risk categories (e.g., memory leaks, injection points).
6. **Monitor and Detect** (Production)
   - Logs and SIEM systems watch for unusual crashes or security events that might hint at missed coding standard issues (like a buffer overflow exploit attempt).
   - If an incident is detected, feed that knowledge back into the **Assess and Plan** stage to refine or add new standards.
7. **Respond** and **Maintain**
   - If a coding standard–related security incident occurs (say, a race condition exploit), respond by patching the code **and** adjusting the DevSecOps pipeline to automatically detect similar issues in the future.
   - Over time, **update** your coding standards and automated rules as new threats emerge or as your codebase evolves.

Visual Summary

- **DEV:** Writes code following the C/C++ secure coding standards.
- **SEC:** Provides the scanning rules, threat modeling, and compliance checks.
- **OPS:** Monitors logs and ensures correct environment configuration.
- The pipeline continually cycles from **Pre-production** (plan, design, build, test) to **Production** (monitor, respond, stabilize), with the **coding standards** validated at multiple points to ensure a secure release.

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 Value)**	High	Unlikely	Medium	High	2
STD-002-CPP **(Data Value)**	High	Common	Low	High	4
STD-003-CPP **(String Correctness)**	High	Common	Low	High	5

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STD-004-CPP **(SQL Injection)**	Critical	Very Common	Medium	Highest	5
STD-005-CPP **(Memory Protection)**	High	Common	Medium	High	4
STD-006-CPP **(Assertions)**	Medium	Common	Low	Medium	3
STD-007-CPP **(Exceptions)**	Medium	Occasional	Medium	Medium	3
STD-008-CPP **(Concurrency / Race Conditions)**	High	Common	Medium	High	4
STD-009-CPP **(Hardcoded Secrets)**	Critical	High	Low	Highest	5
STD-010-CPP **(Logging)**	High	Very Common	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.

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	**Definition and Usage** • **What It Is:** Encryption of data stored on persistent media (e.g., databases, file systems, backups). • **How / Why It Applies:** Any sensitive data saved in the system must be protected to prevent unauthorized access if storage media are compromised (stolen hard drives, unauthorized backups, misconfiguration). **Policy Statement** • **Scope:** All storage locations holding sensitive or personally identifiable information (PII), including files, databases, and archive backups. • **Requirements:** 1. Use **industry-standard** encryption algorithms (e.g., AES-256). 2. Manage encryption keys securely (e.g., in a hardware security module or dedicated key vault). 3. Ensure that backups and snapshots are **equally** encrypted. 4. Periodically audit that the correct encryption is applied. **When to Apply** • **Default** for storing any sensitive data in production databases. • **Mandatory** if data is subject to compliance regulations (GDPR, HIPAA, etc.).
Encryption in flight	**Definition and Usage** • **What It Is:** Encryption of data **while being transmitted** across networks, including internal or external connections. • **How / Why It Applies:** Prevents interception or eavesdropping on data in transit, protecting confidentiality and integrity from man-in-the-middle attacks. **Policy Statement** • **Scope:** All traffic containing sensitive data (e.g., user credentials, personal information) across untrusted or semi-trusted networks. • **Requirements:** 1. Use **TLS 1.2+** or equivalent secure protocols for web (HTTPS), API calls, or microservices. 2. Disable weak ciphers and keep protocols up to date. 3. Verify certificates properly (no self-signed certs in production).

a. Encryption	Explain what it is and how and why the policy applies.
	When to Apply • **Default** for any external or internal service communication containing sensitive data. • **Always** for remote access (VPN, SSH, etc.) or client-to-server traffic.
Encryption in use	**Definition and Usage** • **What It Is:** Techniques for keeping data encrypted even in memory during computation, often referred to as "Confidential Computing." • **How / Why It Applies:** Minimizes the risk of unauthorized memory access or data leakage from running processes (e.g., through virtualization vulnerabilities, memory dumps). **Policy Statement** • **Scope:** Particularly high-security environments or specialized workloads where memory-level protections (e.g., Intel SGX, AMD SEV) are feasible. • **Requirements:** 1. Evaluate feasibility of hardware-based encryption enclaves for extremely sensitive workflows. 2. Restrict debug or dump capabilities that could expose data in memory. 3. Follow secure coding patterns to avoid writing unencrypted secrets into logs or core dumps. **When to Apply** • **Case-by-case** in high-assurance systems, especially those handling cryptographic key operations or regulated data requiring advanced in-memory protection.

b. Triple-A Framework*	Explain what it is and how and why the policy applies.
Authentication	**Definition and Usage** • **What It Is:** The process by which a user or system **proves** its identity (e.g., via passwords, tokens, biometrics). • **Why:** Ensures that only valid, recognized entities gain access to system resources. **Policy Statement** • **Scope:** All user and service logins, including administrative and system accounts.

b. Triple-A Framework*	**Explain what it is and how and why the policy applies.**
	• **Requirements**: 1. Password policies enforcing complexity, rotation, and MFA (where possible). 2. Central authentication store or Identity Provider (IdP) to unify credentials. 3. Securely handle login data (never store passwords in plaintext, limit login attempts, etc.). **When to Apply** • **Always** whenever a user or process requests system access. • **Must** track all successful/failed login attempts.
Authorization	**Definition and Usage** • **What It Is**: The process that determines **which actions** an authenticated user or process may perform (e.g., read vs. write privileges). • **Why**: Enforces the principle of least privilege, ensuring each account can only access what's necessary. **Policy Statement** • **Scope**: Setting and managing user roles, access rights to database tables, file systems, and application features. • **Requirements**: 1. Role-based or attribute-based access control mechanisms. 2. All privilege changes (new roles, newly added permissions) must be documented and approved. 3. Periodic review of roles to detect unnecessary or stale privileges. **When to Apply** • **Whenever** an authenticated user or process attempts an operation or resource request. • **Regularly** re-audit permissions to maintain compliance.
Accounting	**Definition and Usage** • **What It Is**: The process of **logging** and **tracking** user activities, changes to systems, and data access. • **Why**: Provides an audit trail for forensic analysis, compliance, and intrusion detection. **Policy Statement** • **Scope**: All user-level and system-level actions, especially administrative changes.

b. Triple-A Framework*	**Explain what it is and how and why the policy applies.**
	• **Requirements**: 1. **Log** user logins, changes to the database, addition of new users, and file access events. 2. Store logs in a tamper-evident system (e.g., centralized SIEM solution). 3. Retain logs for a specified period (e.g., 6–12 months) to meet compliance and incident response needs. **When to Apply** • **Always** in production environments, with special focus on critical systems where compliance demands thorough logging.

*****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

Standard	Relevant Principles	Justification

STD-001-CPP (Data Type)	2, 9, 10	(2) Heed compiler warnings on type mismatches. (9) QA catches type errors. (10) Aligns with secure coding best practices.
STD-002-CPP (Data Value)	1, 4, 7	(1) Validate input data ranges. (4) Keep checks simple. (7) Sanitize data that might cross system boundaries.
STD-003-CPP (String Correctness)	1, 9, 10	(1) Validate string input. (9) Use robust QA for string handling. (10) Follows secure coding guidelines to avoid buffer overflows.
STD-004-CPP (SQL Injection)	1, 7, 8	(1) Validate input to queries. (7) Sanitize data sent to databases. (8) Defense in depth via parameterized statements.
STD-005-CPP (Memory Protection)	3, 8, 9	(3) Architect for security with safe memory usage. (8) Layers of defense: address sanitizer, RAII. (9) QA finds memory leaks early.
STD-006-CPP (Assertions)	2, 9	(2) Compiler or static analysis warnings can identify misused asserts. (9) QA fosters correct usage in debug vs. production.
STD-007-CPP (Exceptions)	3, 4, 9	(3) Proper error handling design. (4) Keep it simple by not overusing exceptions. (9) Testing ensures correct catch blocks.
STD-008-CPP (Concurrency)	5, 8, 9	(5) Default deny (restrict concurrency unless safe). (8) Defense in depth with locks, atomics. (9) QA with thread sanitizers.
STD-009-CPP (Hardcoded	5, 10	(5) Default deny

Secrets)		storing secrets in code. (10) Align with secure coding best practices to keep secrets out of repos.
STD-010-CPP (Logging)	7, 8, 9	(7) Sanitize logs (avoid printing sensitive data). (8) Defense in depth using auditing. (9) QA ensures logs contain relevant info.
Encryption (At Rest, In Flight, Use)	8, 10	(8) Defense in depth for data confidentiality. (10) Secure coding extends to cryptographic handling.
Triple-A (Authentication, Authorization, Accounting)	5, 6, 8, 9	(5) Default deny is inherent in Access Control. (6) Least privilege for roles. (8) Multiple layers of identity checks. (9) Auditing.

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	01/24/2025	Comprehensive Security Policy Update (Milestone 3)	Zainab Lowe	[Insert text.]
3.0	02/14/2025	Security Policy Update for Green Pace (Project One)	Zainab Lowe	[Insert text.]

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV