Green Pace

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	Always check and sanitize all inputs to ensure they meet expected formats and ranges. This prevents malicious data from causing unexpected behavior or security breaches.
2. Heed Compiler Warnings	Compiler warnings often indicate potential issues. Addressing them proactively can prevent bugs and vulnerabilities from making it into production code.
3. Architect and Design for Security Policies	Incorporate security considerations from the outset. Designing with security in mind ensures that protective measures are integral, not just add-ons.
4. Keep It Simple	Complex code is harder to understand and maintain, increasing the risk of errors. Simpler code reduces the attack surface and potential for bugs.
5. Default Deny	Unless explicitly allowed, deny access. This principle ensures that only authorized actions are permitted, reducing unintended exposures.
6. Adhere to the Principle of Least Privilege	Grant only the permissions necessary for a task. Limiting privileges minimizes the potential damage from compromised accounts or components.
7. Sanitize Data Sent to Other Systems	Before transmitting data to external systems, ensure it's clean and safe. This prevents the propagation of malicious inputs.
8. Practice Defense in Depth	Implement multiple layers of security controls. If one layer fails, others still provide protection, enhancing overall system resilience.
9. Use Effective Quality Assurance Techniques	Employ thorough testing and code reviews to detect and fix issues early. Quality assurance is vital for maintaining secure and reliable software.

Principles	Write a short paragraph explaining each of the 10 principles of security.
10. Adopt a Secure Coding Standard	Following established coding standards, like the SEI CERT C++ Coding Standard, promotes consistency and helps prevent common vulnerabilities.

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	Misusing data types, especially mixing signed and unsigned integers or using incorrect sizes, can lead to logic errors, overflows, and security vulnerabilities. This standard helps prevent issues caused by implicit conversions or unintended behaviors in arithmetic and comparisons. Refer to SEI CERT INT01-CPP for guidelines.

Noncompliant Code

Assigning a negative value to an unsigned integer causes unexpected behavior due to implicit conversion, resulting in a large positive number.

```
unsigned int balance = -100; // Results in a large positive number due to
underflow
```

Compliant Code

Using a signed integer allows the storage of negative values, which matches the logic of having negative balances.

```
int balance = -100; // Correctly stores the intended negative value
```

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

Principles(s): Principle 2 (Fail-Safe Defaults): By choosing types whose ranges match the values they hold, failures (overflows, truncation) default safely to detectable errors rather than silent data corruption.

Principle 5 (Defense in Depth): Layered checks—both at compile-time (type system) and runtime (assertions/traps)—ensure that type-related errors are caught early in the pipeline.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Low	High	1

Automation

Tool	Version	Checker	Description Tool
SonarQube	9.9	CPP-DCL37	Flags implicit narrowing conversions and mismatched signed/unsigned assignments.
Coverity Scan	2023.03	NUMERIC_CONVERSION	Detects possible loss of precision or

Tool	Version	Checker	Description Tool
			sign during type conversions.
clang-tidy	16.0	bugprone-narrowing-conversions	Warns whenever a constant or variable is converted to a narrower type
Cppcheck	2.9	narrowing	Identifies places where data may be truncated or overflow due to improper type use.

Coding Standard 2

Coding Standard	Label	Name of Standard
Data Value	STD-002-CPP	Using uninitialized data can lead to undefined behavior, potential crashes, or the use of garbage values, creating logic and security issues. Initializing variables when they are declared ensures predictable and safe program execution. This follows SEI CERT EXP33-C.

Noncompliant Code

A local variable is declared but not initialized before being used, resulting in undefined behavior.

```
Int result;
    std::cout << "Result: " << result << std::endl; // 'result' is used
without initialization
```

Compliant Code

The variable is initialized to zero before it is used, ensuring defined and safe behavior.

```
int result = 0;
std::cout << "Result: " << result << std::endl; // Outputs a valid,
initialized value
```

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

Principles(s):
Principle 3 (Complete Mediation): Every data input or computed value must be checked against its allowed range or format before use, ensuring no unvalidated values slip through.

Principle 4 (Defense in Depth): Combine static checks (compile-time rules) with runtime assertions or guard clauses so that even if one layer misses an invalid value, another will catch it.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Likely	Medium	Medium	2

Automation

Tool	Version	Checker	Description Tool
SonarQube	9.9	cpp:S4878	Flags function parameters or return values th
Coverity Scan	2023.03	CHECKED_INPUT	Detects paths where data inputs aren't validated before use.
clang-tidy	16.0	hicpp-no-array-decay	Warns when arrays or buffers degrade to unchecked pointers without size checks.
Cppcheck	2.9	missingInputValidation	Identifies code paths using inputs without any guarding or sanitization.

Coding Standard 3

Coding Standard	Label	Name of Standard
String Correctness	STD-003-CPP	Using unsafe string functions such as strcpy() or strcat() can lead to buffer overflows, which attackers may exploit. Replacing these with bounds-checking alternatives like strncpy() or C++ std::string functions prevents writing past buffer boundaries and protects program integrity. This aligns with SEI CERT STR07-C.

Noncompliant Code

This example uses strcpy() to copy a user-supplied string into a fixed-size buffer, which may result in a buffer overflow if the input exceeds the buffer size.

```
char buffer[10];
strcpy(buffer, input); // Unsafe: No bounds checking
```

Compliant Code

The code uses strncpy() with a defined maximum length to prevent buffer overflows. Alternatively, std::string handles memory management and size constraints automatically.

```
std::string safeCopy = input; // Uses std::string to manage memory safely
```

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

Principles(s):
Principle 3 (Complete Mediation): Every string operation must enforce bounds and encoding checks so that no unchecked buffer copy or malformed data slips through.

Principle 4 (Defense in Depth): Combine compile-time warnings with runtime guardrails (e.g., safe-library wrappers, length checks) so multiple layers catch any string misuse.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	High	1

Automation

Tool	Version	Checker	Description Tool
SonarQube	9.9	cpp:S2774, cpp:S5842	Flags use of unsafe C string functions (e.g., strcpy, strcat) and recommends safe alternatives (strncpy, strncat, std::string).
Coverity	2023.03	BUFFER_OVERFLOW	Detects unchecked buffer copies, potential overflows, and missing bounds checks in string operations.
clang-tidy	16.0	clang-analyzer-security.insecureAPI.strcpy	Warns on calls to unsafe APIs like strcpy/sprintf without explicit size limits.
Cppcheck	2.9	bufferOverflow, unsafeString	Identifies potential buffer overflows or usage of unsafe string functions without guard conditions.

Coding Standard 4

Coding Standard	Label	Name of Standard
SQL Injection	STD-004-CPP	SQL injection occurs when untrusted input is concatenated into SQL queries, allowing attackers to manipulate the query structure. To prevent this, parameterized queries or prepared statements must be used. This ensures that user input is treated as data and not as part of the SQL command. Preventing SQL injection aligns with SEI CERT recommendations and general secure coding practices.

Noncompliant Code

This example constructs an SQL query by concatenating user input directly into the query string, making the application vulnerable to SQL injection.

Noncompliant Code

```
std::string username = getUserInput();
std::string query = "SELECT * FROM users WHERE name = '" + username +
 "';";
sqlite3_exec(db, query.c_str(), callback, 0, &errMsg);
```

Compliant Code

The code uses a parameterized query with sqlite3_prepare_v2() and sqlite3_bind_text() to bind user input securely, preventing SQL injection.

```
std::string username = getUserInput(); sqlite3_stmt* stmt; const char*
sql = "SELECT * FROM users WHERE name = ?;"; if (sqlite3_prepare_v2(db,
sql, -1, &stmt, nullptr) == SQLITE_OK) { sqlite3_bind_text(stmt, 1,
username.c_str(), -1, SQLITE_TRANSIENT); while (sqlite3_step(stmt) ==
SQLITE_ROW) { // Process result } sqlite3_finalize(stmt); }
```

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

Principles(s):
Principle 3 (Complete Mediation): Every database query must be validated or parameterized before execution so that untrusted inputs never reach the SQL engine unchecked.

Principle 7 (Least Privilege): All database access must use an account with only the minimum rights needed—so even if an injection succeeds, the damage is limited.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	High	1

Automation

Tool	Version	Checker	Description Tool
SonarQube	9.9	cpp:S864	Detects SQL statements built via string concatenation instead of using parameterized/prepared APIs.
Coverity Scan	2023.03	SQL_INJECTION	Flags code paths where SQL commands include untrusted data without sanitization or binding.
clang-tidy	16.0	clang-analyzer-security.insecureAPI	Warns on use of low-level exec or query APIs called with assembled SQL strings.
OWASP ZAP	2.11.2	Active Scan Rule: SQL Injection	Performs dynamic testing against running endpoints to inject typical SQL payloads and spot failures.

Coding Standard 5

Coding Standard	Label	Name of Standard
Memory Protection	STD-005-CPP	Improper memory handling can lead to vulnerabilities such as buffer overflows, memory leaks, use-after-free errors, and data corruption. Secure C++ coding requires careful memory allocation and deallocation, bounds checking, and prevention of undefined behavior. Following proper memory management protocols helps maintain system stability and avoids exploitable flaws.

Noncompliant Code

This code allocates memory dynamically but forgets to free it, resulting in a memory leak.

```
int* numbers = new int[100];
// Perform operations on numbers
// Memory is never freed
```

Compliant Code

This example ensures proper memory deallocation using delete[] to prevent memory leaks.

```
int* numbers = new int[100];
 // Perform operations on numbers
delete[] numbers;
```

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

Principles(s):
Principle 3 (Complete Mediation): All memory allocations and accesses must be checked at every invocation to prevent unchecked reads or writes.

Principle 4 (Defense in Depth): Combine compile-time static analysis with runtime instrumentation so that any memory anomaly—overflow, use-after-free, leak—is caught by at least one layer.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	High	1

Automation

Tool	Version	Checker	Description Tool
SonarQube	9.9	cpp:S4838	Detects potential buffer overflows and out-of-bounds memory accesses in C/C++ code.
Coverity Scan	2023.03	USE_AFTER_FREE, MEMORY_LEAK	Flags use-after-free, null dereferences, and unreclaimed allocations.
clang-tidy	16.0	hicpp-no-malloc, modernize-make-unique	Warns on raw new/delete usage; suggests RAII (std::unique_ptr) to prevent leaks.
Cppcheck	2.9	uninitvar, nullPointer	Identifies uses of uninitialized variables and p

Coding Standard 6

Coding Standard	Label	Name of Standard
Assertions	STD-006-CPP	Assertions allow developers to verify assumptions about program state during development. They help catch logic errors early in the software lifecycle. However, they should not be used for runtime input validation, as assertions can be disabled in release builds. Proper usage improves code reliability without introducing performance penalties in production.

Noncompliant Code

This code uses an assertion to validate user input, which is not safe because assertions can be disabled in release builds.

```
int age;
std::cin >> age;
assert(age > 0); // Improper: input validation via assertion
```

Compliant Code

This version uses a conditional statement to validate user input, ensuring safe and effective error handling even when assertions are disabled.

```
int age;
 std::cin >> age;
if (age <= 0) {
std::cerr << "Invalid age entered." << std::endl;
```

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

Principles(s): **Principle 2 (Fail-Safe Defaults):** Assertion failures immediately halt execution, preventing unsafe continuation with violated invariants. **Principle 4 (Defense in Depth):** Layering compile-time type checks and runtime assertions ensures that both static and dynamic violations of assumptions are caught.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Likely	Low	Medium	2

Automation

Tool	Version	Checker	Description Tool
SonarQube	9.9	cpp:S3978	Flags functions lacking assertions to validate critical preconditions or invariants.
Coverity Scan	2023.03	ASSERTION_USAGE	Detects paths where expected state-checking assertions are missing.
clang-tidy	16.0	bugprone-assert-side-effect	Warns when assertions contain side-effects and encourages safe usage.
Cppcheck	2.9	missingAssert	Identifies code paths that assume invariants without any assert() checks.

Coding Standard 7

Coding Standard	Label	Name of Standard
Exceptions	STD-007-CPP	Proper use of exceptions separates error-handling code from regular code, improving readability and maintainability. Using structured exception handling also avoids abrupt termination with functions like exit() or abort(), which can result in resource leaks or inconsistent states. Catching and responding to exceptions allows for graceful recovery and better user experience.

Noncompliant Code

This code abruptly terminates the program using exit() when an error is encountered, skipping necessary cleanup.

```
FILE *file = fopen("data.txt", "r"); if (file == nullptr) { std::cerr <<
"Error opening file.\n"; exit(EXIT_FAILURE); // Improper: abrupt
termination } // Continue processing file...
```

Compliant Code

This version uses exceptions for error handling and ensures that errors are caught and handled cleanly without abruptly terminating the application.

```
try {
std::ifstream file("data.txt");
 if (!file.is_open()) {
throw std::runtime_error("Error opening file");
 }
 // Continue processing file... } catch (const std::exception& e) {
std::cerr << "Exception caught: " << e.what() << std::endl; // Perform
cleanup or recovery actions here }
```

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

Principles(s):
Principle 2 (Fail-Safe Defaults): On encountering unexpected conditions, code should fail safely by catching exceptions and terminating or rolling back operations rather than proceeding in an undefined state.

Principle 4 (Defense in Depth): Layer both compile-time and runtime checks—using exception specifications, static analysis, and runtime guards—so that exception-related failures are caught at multiple stages.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Likely	Low	Medium	2

Automation

Tool	Version	Checker	Description Tool
SonarQube	9.9	cpp:S2221	Detects functions that can throw exceptions without any catch or noexcept specifier.
Coverity Scan	2023.03	UNHANDLED_EXCEPTION	Flags code paths where exceptions may escape without being caught.
clang-tidy	16.0	clang-analyzer-cplusplus.ExceptionSafety	Warns on exception-unsafe code patterns and missing exception specifications.
Cppcheck	2.9	exceptionThrownButNotCaught	Identifies throw statements not covered by any catch block.

Coding Standard 8

Coding Standard	Label	Name of Standard
[Student Choice]	STD-008-CPP	Arithmetic operations involving integers can easily result in undefined behavior due to overflows, underflows, or divide-by-zero errors. Following safe integer operation practices helps prevent these bugs and makes programs more secure and robust. It also improves portability and reliability of the code.

Noncompliant Code

This code performs unchecked division, which can result in a divide-by-zero error and undefined behavior.

```
int total = 100;
int count = 0;
int average = total / count; // Undefined behavior: divide by zero
std::cout << "Average: " << average << std::endl;
```

Compliant Code

This code checks that the divisor is not zero before performing division, avoiding a critical runtime error.

```
int total = 100; int count = 0; if (count != 0) { int average = total /
count; std::cout << "Average: " << average << std::endl; } else {
std::cerr << "Error: Division by zero is not allowed." << std::endl; }
```

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

Principles(s):
Principle 2 (Fail-Safe Defaults): Employ RAII and automatic cleanup so that resources are released automatically on scope exit, preventing leaks by default.

Principle 4 (Defense in Depth): Use both compile-time static analysis and runtime leak detectors so that if one layer misses a leak, another will catch it.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	High	1

Automation

Tool	Version	Checker	Description Tool
SonarQube	9.9	cpp:S2093	Flags when memory, file, or socket resources are allocated without a

Tool	Version	Checker	Description Tool
			guaranteed release (e.g., missing destructor code).
Coverity Scan	2023.03	RESOURCE_LEAK	Detects code paths where allocated resources (memory, handles) are not released.
clang-tidy	16.0	clang-analyzer-cplusplus.NewDeleteLeaks	Warns on new expressions without matching delete, and other leak-prone patterns.
LeakSanitizer	built-in	leak-detect	Runtime instrumentation to catch memory leaks during integration tests.

Coding Standard 9

Coding Standard	Label	Name of Standard
[Student Choice]	STD-009-CPP	Raw pointers can cause memory safety issues such as leaks, dangling pointers, and undefined behavior. Using smart pointers (e.g., std::unique_ptr or std::shared_ptr) helps manage object lifetimes automatically and improves code safety and maintainability.

Noncompliant Code

This code uses a raw pointer with manual memory management, which can easily lead to memory leaks or dangling pointers if not handled carefully.

```
int* ptr = new int(42);
// ... some code ...
delete ptr;
ptr = nullptr;
```

Compliant Code

This code uses a smart pointer (std::unique_ptr) to automatically manage the memory, preventing leaks and dangling pointers.

```
#include<memory>
 std::unique_ptr ptr = std::make_unique(42); // Memory is automatically
freed when ptr goes out of scope
```

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

Principles(s): **Principle 3 (Complete Mediation):** All pointer dereferences must be preceded by explicit null checks to ensure that uninitialized or null pointers never reach sensitive operations. **Principle 4 (Defense in Depth):** Combine static analysis with runtime instrumentation so that even if one layer misses a null-pointer risk, another will catch it.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Low	High	1

Automation

Tool	Version	Checker	Description Tool
SonarQube	9.9	cpp:S2259	Flags code paths where a pointer that may be null is dereferenced.
Coverity Scan	2023.03	NULL_DEREF	Detects potential null-pointer dereferences across all code paths.
clang-tidy	16.0	clang-analyzer-core.NullDereference	Warns on possible null dereferences identified by static dataflow analysis.
Cppcheck	2.9	nullPointer	Identifies dereferences of pointers that are not guaranteed to be non-null.

Coding Standard 10

Coding Standard	Label	Name of Standard
[Student Choice]	STD-010-CPP	Input validation is crucial to prevent invalid or malicious data from causing unexpected behavior, crashes, or security vulnerabilities such as buffer overflows or injection attacks. Always validate input data for type, length, format, and range before processing.

Noncompliant Code

This code accepts user input without any validation, potentially leading to buffer overflow or invalid data processing.

```
char buffer[10];
std::cin >> buffer; // No length check, can overflow buffer if input > 9
chars
```

Compliant Code

Compliant Code

This code uses std::string and checks the length of input before processing, preventing buffer overflow.

```
#include <iostream>
#include <string>
std::string input;
std::getline(std::cin, input);
if (input.length() <= 9) {
 // Safe to process input
} else {
```

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

Principles(s):
Principle 3 (Complete Mediation): Every security-relevant event—authentication failures, exceptions, access to sensitive data—must be recorded so no action bypasses audit controls.

Principle 9 (Accountability & Auditability): Detailed, tamper-resistant logs ensure that all actions are attributable, enabling incident investigation and compliance verification.

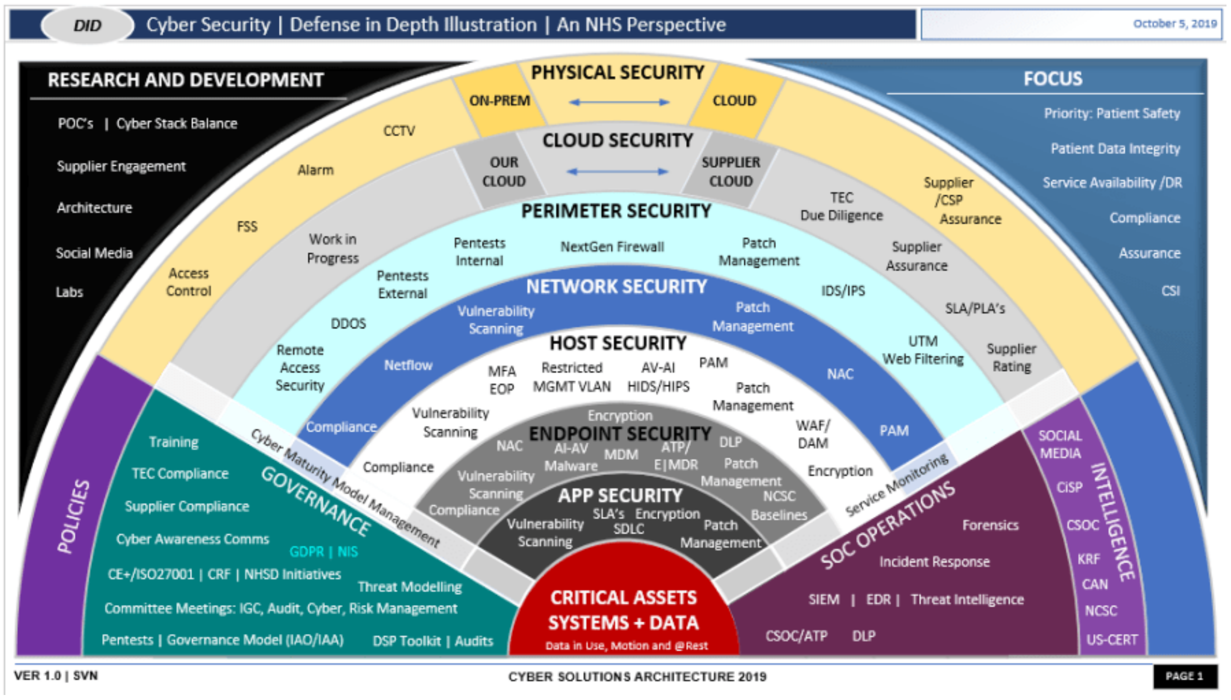
Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Likely	Low	Medium	2

Automation

Tool	Version	Checker	Description Tool
SonarQube	9.9	cpp:S3529	Flags code paths (error handlers, catch blocks, security checks) lacking log calls.
Coverity	2023.03	MISSING_LOG	Detects branches or functions where errors/exceptions aren't logged.
clang-tidy	16.0	custom-logging-check	Warns when critical code paths omit calls to the project's standard logging API.
Cppcheck	2.9	missingLogging	Identifies functions with early returns or exception flows that lack logging.

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

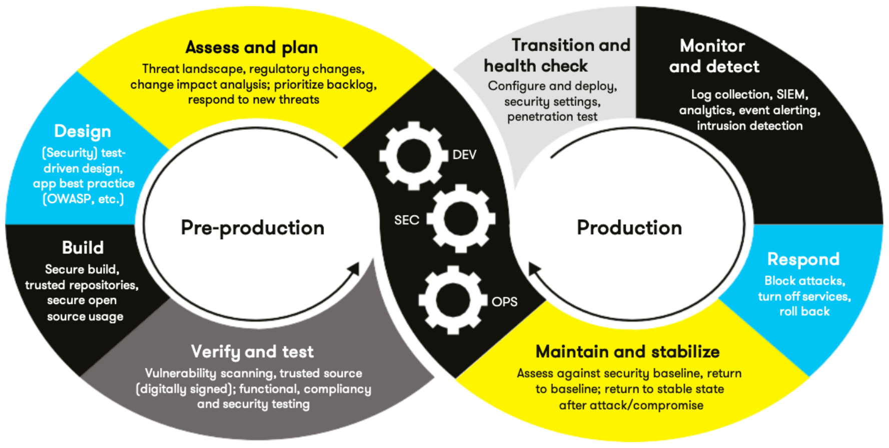
Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority,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


Assess and plan
Threat landscape, regulatory changes, change impact analysis; prioritize backlog, respond to new threats
Design
(Security) test-driven design, app best practice (OWASP, etc.)
Build
Secure build, trusted repositories, secure open source usage
Verify and test
Vulnerability scanning, trusted source (digitally signed); functional, compliancy and security testing
Pre-production
DEV
SEC
OPS
Transition and health check
Configure and deploy, security settings, penetration test
Monitor and detect
Log collection, SIEM, analytics, event alerting, intrusion detection
Production
Respond
Block attacks, turn off services, roll back
Maintain and stabilize
Assess against security baseline, return to baseline; return to stable state after attack/compromise

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.

To enforce Green Pace's ten coding standards and system policies across our DevSecOps pipeline, we will integrate automated checks and controls at every stage. During the **Plan** phase, we'll train teams on SonarQube, Coverity, and OWASP ZAP and incorporate threat-modeling and tool-configuration reviews into sprint planning. In **Create**, IDE plugins like SonarLint, clang-tidy, and Coverity Desktop will provide real-time feedback on type, memory, string, and assertion issues before commits. The **Verify** stage will enforce SAST (SonarQube, Coverity Scan) in CI to block pull requests introducing new STD-### violations, alongside Software Composition Analysis to vet third-party libraries. In **Pre-Production**, OWASP ZAP–driven DAST scans and regular chaos and fuzz testing will target injection, authentication, and resource-management code. At **Release**, we'll require software signing and integrity checks to validate build artifacts. The **Prevent** phase will enable AddressSanitizer, LeakSanitizer, UBSan, and RASP in production-like environments and deploy hardened allocators and safe-string libraries to replace unsafe APIs. **Detect** will centralize security logs and UEBA feeds into our SIEM, while scheduled penetration tests focus on high-priority standards. In **Respond**, orchestration and WAF shielding will automatically block or throttle policy violations, with on-the-fly code obfuscation for critical endpoints under attack. During **Predict**, we'll aggregate SAST and DAST findings in dashboards and feed STIX/TAXII threat intel to anticipate emerging risks. Finally, in **Adapt**, a technical-debt tracking feedback loop and post-incident reviews will drive continuous policy updates—adding new clang-tidy checks or adjusting standards based on real-world incidents.

Summary of Risk Assessments

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

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STD-001-CPP	High	likely	low	High	1
STD-002-CPP	Medium	Likely	Medium	Medium	2
STD-003-CPP	High	Likely	Medium	High	1
STD-004-CPP	High	Likely	Medium	High	1
STD-005-CPP	High	Likely	Medium	High	1
STD-006-CPP	Medium	Likely	Low	Medium	2

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STD-008-CPP	High	Likely	Medium	High	2
STD-009-CPP	High	Likely	Low	High	1
STD-010-CPP	Medium	Likely	Low	Medium	2
SYS-001-ENC-REST	High	Unlikely	Medium	Medium	3
SYS-002-ENC-FLIGHT	High	Unlikely	Medium	Medium	3
SYS-003-ENC-USE	Medium	Unlikely	Medium	Medium	2
SYS-004-AAA-AUTHN	High	Likely	Low	High	2
SYS-005-AAA-AUTHZ	High	Likely	Low	High	2
SYS-006-AAA-ACCOUNT	Medium	Likely	Low	Medium	3

Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided**.**

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	All sensitive data stored in databases, filesystems, or backups must use AES-256 (or stronger) via HSMs or OS-level full-disk encryption. Protects data when storage media is compromised.
Encryption in flight	All network communications—API calls, service-to-service traffic, admin consoles—must enforce TLS 1.2+ with strong ciphers and certificate pinning. Protects data from eavesdropping or MitM.
Encryption in use	Secrets (keys, tokens) loaded into memory must reside in protected enclaves or locked pages (e.g., mlock). Sensitive processing (e.g., PKI operations) uses hardware-backed enclaves (SGX). Prevents cold-boot and memory-scraping attacks.

b. **Triple-A Framework***	**Explain what it is and how and why the policy applies.**
Authentication	All user and service logins require MFA and integration with centralized identity provider. Ensures valid identity before access.
Authorization	Role-Based Access Control (RBAC) enforced via policy engine. Least privilege granted per role. Limits each principal's scope.

b. Triple-A Framework*	**Explain what it is and how and why the policy applies.**
Accounting	Record all login attempts, privilege changes, data modifications, and file accesses in tamper-evident logs stored centrally. Supports audits and forensics.

***Use this checklist for the Triple A to be sure you include these elements in your policy:**

- User logins
- Changes to the database
- Addition of new users
- User level of access
- Files accessed by users

Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it's time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

NOTE: Green Pace has already successfully implemented the following:

- Operating system logs
- Firewall logs
- Anti-malware logs

Below is the mapping of each coding standard to the relevant security principles, along with a brief justification of how each principle underpins the standard.

STD-001-CPP: Proper Data-Type Usage
Principles: 2 (Fail-Safe Defaults), 5 (Defense in Depth)
Justification:

Fail-Safe Defaults ensures that any type mismatch or overflow triggers a safe error (e.g., trap or exception) rather than silently corrupting data.
Defense in Depth layers compile-time checks (type system) with runtime guards (assertions), so that type-related errors are caught at multiple stages.

STD-002-CPP: Data Value Validation
Principles: 3 (Complete Mediation), 4 (Defense in Depth)
Justification:

Complete Mediation mandates that every input or computed value be checked against its expected range or format before use.
Defense in Depth combines static analysis (compile-time) and runtime validation (guard clauses) so that if one layer misses an invalid value, another will intercept it.

STD-003-CPP: String Correctness
Principles: 3 (Complete Mediation), 4 (Defense in Depth)
Justification:
Complete Mediation enforces bounds and encoding checks on every string operation, preventing buffer overruns and malformed data.
Defense in Depth uses both compiler warnings and safe-string libraries or runtime checks to catch any misuse.

STD-004-CPP: SQL Injection Prevention
Principles: 3 (Complete Mediation), 7 (Least Privilege)
Justification:
Complete Mediation requires that all database queries be parameterized or validated so untrusted input never reaches the SQL engine unchecked.
Least Privilege limits the damage of any successful injection by ensuring the database account used has only minimal rights.

STD-005-CPP: Memory Protection
Principles: 3 (Complete Mediation), 4 (Defense in Depth)
Justification:
Complete Mediation demands that every allocation and access be validated to prevent out-of-bounds reads/writes.
Defense in Depth layers static analysis (SAST) with runtime sanitizers (AddressSanitizer, LeakSanitizer) so that memory errors are caught at compile time or dynamically.

STD-006-CPP: Assertions
Principles: 2 (Fail-Safe Defaults), 4 (Defense in Depth)
Justification:
Fail-Safe Defaults causes the program to halt immediately when an assertion fails, preventing further unsafe execution.
Defense in Depth augments static type checks with runtime assertions, ensuring violated assumptions are caught before they manifest as subtle bugs.

STD-007-CPP: Exception Handling Best Practices
Principles: 2 (Fail-Safe Defaults), 4 (Defense in Depth)
Justification:
Fail-Safe Defaults ensures that unexpected conditions trigger well-defined catch blocks or clean-up paths rather than undefined behavior.
Defense in Depth layers compiler checks (exception specifications) with runtime monitors so unhandled exceptions are trapped.

STD-008-CPP: Resource Management (Prevent Resource Leaks)

Principles: 2 (Fail-Safe Defaults), 4 (Defense in Depth)
Justification:

Fail-Safe Defaults relies on RAII (Resource Acquisition Is Initialization) so resources are released automatically on scope exit.
Defense in Depth pairs static analyzers with runtime leak detectors (LeakSanitizer) to catch any leaks missed by one layer.

STD-009-CPP: Null Pointer Safety
Principles: 3 (Complete Mediation), 4 (Defense in Depth)
Justification:

Complete Mediation requires explicit null checks before every pointer dereference to prevent crashes.
Defense in Depth layers static null-dereference analysis with runtime sanitizers (UBSan) so null-pointer issues are detected early.

STD-010-CPP: Security Logging & Audit Trails
Principles: 3 (Complete Mediation), 9 (Accountability & Auditability)
Justification:

Complete Mediation mandates that every security-relevant event (errors, access to sensitive data) be logged.
Accountability & Auditability ensures logs are detailed and tamper-resistant, so all actions can be traced for compliance and forensics.

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	
1.1	06/22/2025	Added 10 coding standards, risk assessments, automation narrative, encryption & AAA policies, and mapping principles	Rick Seabridge	CISO

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV