Green Pace Developer: Security Policy Guide Template



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| Principles | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. Validate Input Data | All input should be checked for correctness, length, format, and range before use. Validating input prevents injection attacks, buffer overflows, and logic errors that can compromise system security. |
| 1. Heed Compiler Warnings | Compiler warnings highlight risky or undefined behavior in code. Treating these warnings seriously helps eliminate vulnerabilities early in development before they become exploitable flaws. |
| 1. Architect and Design for Security Policies | Security should be integrated at the design stage, not added later. Planning around authentication, authorization, and error handling ensures applications are resilient against attacks from the start. |
| 1. Keep It Simple | Overly complex code is harder to review and more likely to contain hidden bugs. Writing simple, clear code reduces opportunities for mistakes and makes secure practices easier to maintain. |
| 1. Default Deny | Systems should deny access by default and only allow explicitly authorized actions. This prevents attackers from exploiting unintended permissions or oversights in access control. |
| 1. Adhere to the Principle of Least Privilege | Applications and users should only have the minimum access necessary to complete their tasks. This limits the damage if a system is compromised. |
| 1. Sanitize Data Sent to Other Systems | Before sending data across systems, ensure it is cleaned and validated. This prevents vulnerabilities such as SQL injection, cross-site scripting, and data corruption in dependent systems. |
| 1. Practice Defense in Depth | Multiple layers of security such as validation, authentication, and encryption help protect systems even if one control fails. Layered defenses make exploitation significantly harder. |
| 1. Use Effective Quality Assurance Techniques | Testing, peer reviews, and automated tools catch vulnerabilities and weaknesses before deployment. Strong QA processes are critical to secure, reliable software. |
| 1. Adopt a Secure Coding Standard | Following established secure coding guidelines helps developers avoid common mistakes and maintain consistency. Standards provide a reference point for building reliable, safe applications. |

### 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 | Correct Use of Integer Types |
| --- | --- | --- |
| Data Type | STD-001-CPP | To prevent vulnerabilities related to integer overflows and underflows, which can lead to buffer overflows or other memory corruption issues, it's critical to use the correct integer type for a given range of values. This ensures that calculations and data storage do not exceed the capacity of the assigned variable, thereby maintaining data integrity and program control flow. |

| Noncompliant Code |
| --- |
| Integer overflow can occur when the result of a calculation exceeds the maximum value that the variable type can hold, leading to a wrap-around to the minimum value. |
| cpp<br>#include <iostream><br>#include <limits><br><br>int main() {<br> int a = 2147483647; // Maximum value for a 32-bit signed int<br> int b = 1;<br> int result = a + b; // This will cause an integer overflow<br> std::cout << "Result: " << result << std::endl;<br> return 0;<br>} |

| Compliant Code |
| --- |
| Before the operation, check if the sum will exceed the maximum value to prevent overflow. |
| cpp<br>#include <iostream><br>#include <limits><br><br>int main() {<br> int a = 2147483647;<br> int b = 1;<br> if (b > 0 && a > std::numeric\_limits<int>::max() - b) {<br> // Handle error: overflow will occur<br> std::cerr << "Error: Integer overflow will occur." << std::endl;<br> } else {<br> int result = a + b;<br> std::cout << "Result: " << result << std::endl;<br> }<br> return 0;<br>} |

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

| Principles(s):  Principle 4 (Practice Defense in Depth): This standard acts as an inner layer of defense by protecting against logic errors and buffer overflows caused by integer manipulation, which would otherwise rely on external defenses to catch.  Principle 6 (Design for Secure Failure): By using range checks before operations, the code is designed to safely fail (e.g., throwing a std::overflow\_error) instead of silently wrapping around an integer's maximum value, which could lead to an exploitable state. |
| --- |

Threat Level

| Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | 2 |

Automation

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| CppCheck | 2.1+ | integerOverflow | Detects potential integer overflows or underflows, especially in loop conditions and calculations. |
| SonarQube | Latest | S137 | Flags expressions that always evaluate to true or false, often caused by overflow behavior. |
| Clang Static Analyzer | Latest | core.uninitialized.Assign | Helps catch uninitialized variables where subsequent integer overflow might occur. |
| Coverity | Latest | INTEGER\_OVERFLOW | Identifies where arithmetic operations may exceed the range of the assigned integer type. |

#### Coding Standard 2

| Coding Standard | Label | Validate All Input Data |
| --- | --- | --- |
| Data Value | STD-002-CPP | Unsanitized and unvalidated data from external sources, such as user input or a network connection, can be malicious. Failing to validate this data before use can lead to a variety of security vulnerabilities, including buffer overflows, logic errors, and denial-of-service attacks. This standard ensures all external data is checked against expected formats, ranges, and types. |

| Noncompliant Code |
| --- |
| The program directly uses user input for array indexing without any validation, which could allow a malicious user to cause an out-of-bounds access. |
| cpp<br>#include <iostream><br>#include <vector><br><br>int main() {<br> std::vector<int> data = {1, 2, 3, 4, 5};<br> int index;<br> std::cout << "Enter an index: ";<br> std::cin >> index;<br> std::cout << "Value at index " << index << ": " << data[index] << std::endl;<br> return 0;<br>} |

| Compliant Code |
| --- |
| The program validates the user-provided index to ensure it is within the valid range of the vector, preventing an out-of-bounds access. |
| cpp<br>#include <iostream><br>#include <vector><br><br>int main() {<br> std::vector<int> data = {1, 2, 3, 4, 5};<br> int index;<br> std::cout << "Enter an index: ";<br> std::cin >> index;<br> if (index >= 0 && index < data.size()) {<br> std::cout << "Value at index " << index << ": " << data[index] << std::endl;<br> } else {<br> std::cerr << "Error: Index is out of bounds." << std::endl;<br> }<br> return 0;<br>} |

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.]  Principle 1 (Validate All Input): This is the most direct application of this principle. The standard explicitly requires checking all external data (from users, networks, or files) against formats, ranges, and types to ensure it is safe before use.  Principle 3 (Minimize Attack Surface): By strictly validating input, the standard minimizes the attack surface. It reduces the opportunity for malicious data to traverse the application and execute dangerous operations (like the out-of-bounds access in the noncompliant code). |
| --- |

Threat Level

| Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- |
| Critical | Very Likely | Medium | Critical | 1 |

Automation

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| Clang Static Analyzer | Latest | security.insecureAPI.gets | Detects use of known insecure input functions (like gets) that allow unbounded input. |
| SonarQube | Latest | S5131 | Flags code that relies on user-controlled input without sanitization or validation, often leading to various injections or overflows. |
| OWASP ZAP | Latest | Active Scanner | Runtime/Dynamic tool that injects malicious payloads into all input fields to test for injection and validation failures. |
| Coverity | Latest | TAINTED\_SCALAR | Tracks unvalidated user input data (tainted data) to ensure it is not used in security-sensitive operations without checks. |

#### Coding Standard 3

| Coding Standard | Label | Use Safe String Operations and Sizing |
| --- | --- | --- |
| String Correctness | STD-003-CPP | Improper handling of C-style strings (null-terminated character arrays) is a primary cause of buffer overflows. Using functions that do not perform bounds checking, such as strcpy and sprintf, can lead to an attacker overwriting adjacent memory. Adhering to this standard ensures that string operations use functions that prevent buffer overflows by either checking bounds or by using a safer, more modern approach. |

| Noncompliant Code |
| --- |
| The strcpy function does not check the size of the destination buffer, allowing a larger source string to overwrite adjacent memory and cause a buffer overflow. |
| cpp<br>#include <iostream><br>#include <cstring><br><br>int main() {<br> char src[] = "This is a very long string that will overflow the buffer.";<br> char dest[20];<br> strcpy(dest, src); // Vulnerable: no bounds checking<br> std::cout << "Copied string: " << dest << std::endl;<br> return 0;<br>} |

| Compliant Code |
| --- |
| The strncpy function is used, which takes a size parameter to limit the number of characters copied and prevent a buffer overflow. Alternatively, C++ std::string can be used. |
| cpp<br>#include <iostream><br>#include <cstring><br><br>int main() {<br> char src[] = "This is a very long string that will overflow the buffer.";<br> char dest[sizeof(src) / sizeof(src[0])]; // Or a fixed size large enough<br> strncpy(dest, src, sizeof(dest) - 1); // Limits copy to buffer size<br> dest[sizeof(dest) - 1] = '\0'; // Ensures null termination<br> std::cout << "Copied string: " << dest << std::endl;<br> return 0;<br>} |

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.]  Principle 3 (Minimize Attack Surface): This standard minimizes the attack surface by eliminating or isolating the use of dangerous, unbounded C-style string functions. Moving to std::string removes the entire class of buffer overflow vulnerabilities associated with manual memory management.  Principle 7 (Fail Securely): When a safe function like strncpy is used, the operation stops at the buffer boundary. This ensures that the program fails gracefully (by truncating data) rather than failing insecurely by corrupting memory and crashing or allowing code injection. |
| --- |

Threat Level

| Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- |
| Critical | Likely | Medium | High | 2 |

Automation

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| CppCheck | 2.1+ | bufferAccessOutOfBounds | Identifies common dangerous string functions (strcpy, strcat) and flags potential buffer overflows. |
| Clang Static Analyzer | Latest | security.insecureAPI.strcpy | Flags all usages of known insecure C functions like strcpy, strcat, and sprintf without bounds checking. |
| SonarQube | Latest | S1313 | Highlights the use of insecure functions that can lead to buffer overflows when safer alternatives exist. |
| ASAN (AddressSanitizer) | Integrated in GCC/Clang | Runtime Flag | Runtime tool that detects heap and stack buffer overflows that occur during execution in test environments. |

#### Coding Standard 4

| Coding Standard | Label | Use Parameterized Queries for Database Access |
| --- | --- | --- |
| SQL Injection | STD-004-CPP | SQL injection occurs when user-supplied input is not properly sanitized and is directly concatenated into a SQL statement. This allows attackers to manipulate the query, leading to unauthorized data access, modification, or deletion. Using parameterized queries separates the SQL code from the user input, ensuring the input is treated as a literal value and not as executable code. |

| Noncompliant Code |
| --- |
| The code constructs a SQL query by directly concatenating an unvalidated username variable from user input, which is a classic SQL injection vulnerability. |
| cpp<br>#include <iostream><br>#include <string><br>// This is a simplified example, actual database libraries would be used<br><br>int main() {<br> std::string username = "' OR '1'='1"; // Malicious input<br> std::string password = "password";<br> std::string query = "SELECT \* FROM users WHERE username = '" + username + "' AND password = '" + password + "';";<br> std::cout << "Executing query: " << query << std::endl;<br> // Query will become: SELECT \* FROM users WHERE username = '' OR '1'='1' AND password = 'password';<br> return 0;<br>} |

| Compliant Code |
| --- |
| The code uses a parameterized query (or prepared statement) where placeholders are used for user input, preventing the input from altering the query's structure. |
| cpp<br>#include <iostream><br>#include <string><br>// This is a simplified example using placeholders<br>// In reality, a library like ODBC, libpqxx, etc., would be used<br><br>int main() {<br> std::string username = "' OR '1'='1"; // Malicious input<br> std::string password = "password";<br> // The ? symbols are placeholders for the parameters<br> std::string query = "SELECT \* FROM users WHERE username = ? AND password = ?;";<br> std::cout << "Preparing query: " << query << std::endl;<br> // The database driver will treat 'username' and 'password' as data, not code.<br> return 0;<br>} |

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.]  Principle 1 (Validate All Input): This is the most direct application. Parameterized queries enforce validation by treating all user input as pure data, not executable code, regardless of what characters it contains.  Principle 4 (Practice Defense in Depth): Using prepared statements is a robust layer of defense. Even if input sanitization (Principle 1) fails elsewhere, the database access layer still protects against code execution by ensuring data structure separation. |
| --- |

Threat Level

| Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- |
| Critical | Very Likely | Medium | Critical | 1 |

Automation

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| SonarQube | Latest | S2077 | Flags SQL queries constructed from user-controlled input that are not executed using prepared statements. |
| OWASP ZAP | Latest | Active Scanner | Runtime/Dynamic tool that actively attempts to inject malicious SQL payloads (e.g., ' OR 1=1 --) into application fields to find vulnerabilities. |
| Coverity | Latest | SQLI | Identifies vulnerable code patterns where user-supplied data (tainted data) flows into a SQL query execution sink. |
| Bandit | 1.7.5 | B608 | Detects the use of string formatting to construct database queries instead of using safe parameter binding. |

#### Coding Standard 5

| Coding Standard | Label | Deallocate Memory Correctly and Safely |
| --- | --- | --- |
| Memory Protection | STD-005-CPP | Improper memory management can lead to several security vulnerabilities. Forgetting to free dynamically allocated memory results in memory leaks, while freeing memory and then later trying to access it creates a use-after-free vulnerability. This standard ensures that all dynamically allocated memory is correctly deallocated and that pointers are nullified after delete to prevent dangling pointer issues. |

| Noncompliant Code |
| --- |
| The code deletes the ptr but the pointer itself is not nullified. The program then attempts to use the freed pointer, which is now a dangling pointer, leading to a use-after-free vulnerability and undefined behavior. |
| cpp<br>#include <iostream><br><br>int main() {<br> int\* ptr = new int(10);<br> std::cout << "Before delete: " << \*ptr << std::endl;<br> delete ptr; // Memory is freed<br> // ptr is now a dangling pointer<br> std::cout << "After delete: " << \*ptr << std::endl; // Vulnerable<br> return 0;<br>} |

| Compliant Code |
| --- |
| The code correctly deallocates the memory and then sets the pointer to nullptr. This prevents the dangling pointer from being dereferenced later, mitigating the use-after-free vulnerability. |
| cpp<br>#include <iostream><br><br>int main() {<br> int\* ptr = new int(10);<br> std::cout << "Before delete: " << \*ptr << std::endl;<br> delete ptr;<br> ptr = nullptr; // Pointer is nullified<br> if (ptr != nullptr) {<br> std::cout << "After nullify: " << \*ptr << std::endl;<br> } else {<br> std::cout << "Pointer is null." << std::endl;<br> }<br> return 0;<br>} |

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.]  Principle 3 (Minimize Attack Surface): By strictly controlling memory deallocation (or better yet, using RAII via smart pointers), this standard eliminates common classes of bugs that attackers exploit to gain arbitrary code execution.  Principle 9 (Do Not Trust Infrastructure): Memory corruption vulnerabilities can compromise runtime security mechanisms like stack canaries. This standard implements checks at the code level to prevent corruption, ensuring the code is not relying solely on external OS or compiler defenses. |
| --- |

Threat Level

| Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- |
| Critical | Likely | High | Critical | 1 |

Automation

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| CppCheck | 2.1+ | useAfterFree, memleak | Detects memory leaks, double frees, and attempts to use memory after it has been explicitly freed. |
| Clang Static Analyzer | Latest | cplusplus.NewDelete | Flags incorrect usage of new and delete and issues related to non-nullifying dangling pointers. |
| ASAN (AddressSanitizer) | Integrated in GCC/Clang | Runtime Flag | Runtime tool that detects use-after-free, double-free, and buffer overflows during testing with very low performance overhead. |
| SonarQube | Latest | S5003 | Identifies weak pointers that are not used to track dynamically allocated memory and flags raw pointer misuse. |

#### Coding Standard 6

| Coding Standard | Label | Use Assertions for Internal State Validation |
| --- | --- | --- |
| Assertions | STD-006-CPP | Assertions are a powerful debugging tool for validating assumptions about program state and function preconditions/postconditions. They are intended for detecting programmer errors, not for handling runtime errors caused by user input or unexpected external conditions. Relying on assertions for security-critical checks can be dangerous because they are typically disabled in a production build, leaving the application vulnerable. This standard dictates that assertions should only be used to validate internal program logic. |

| Noncompliant Code |
| --- |
| The code uses an assert to validate user input. When compiled in a release build (with NDEBUG defined), the assertion is removed, allowing unvalidated user input to be processed and potentially causing a buffer overflow or other vulnerability. |
| cpp<br>#include <iostream><br>#include <cassert><br>#include <vector><br><br>int main() {<br> std::vector<int> data = {1, 2, 3, 4, 5};<br> int index;<br> std::cout << "Enter an index: ";<br> std::cin >> index;<br> // Vulnerable: This assert is removed in release builds<br> assert(index >= 0 && index < data.size());<br> std::cout << "Value at index " << index << ": " << data[index] << std::endl;<br> return 0;<br>} |

| Compliant Code |
| --- |
| The code uses a standard if statement with proper error handling to validate user input. This check remains in both debug and release builds, ensuring security is maintained in production. |
| cpp<br>#include <iostream><br>#include <vector><br><br>int main() {<br> std::vector<int> data = {1, 2, 3, 4, 5};<br> int index;<br> std::cout << "Enter an index: ";<br> std::cin >> index;<br> if (index >= 0 && index < data.size()) {<br> std::cout << "Value at index " << index << ": " << data[index] << std::endl;<br> } else {<br> std::cerr << "Error: Invalid index provided." << std::endl;<br> }<br> return 0;<br>} |

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.]  Principle 1 (Validate All Input): This standard reinforces input validation. It mandates that checks for user input must use runtime logic (if/else or exceptions) that remains in production code, rather than assertions that disappear when needed most.  Principle 7 (Fail Securely): If a precondition on user data fails in a release build, the program must fail securely (by rejecting the input or throwing an exception). Relying on an assertion means the program fails *insecurely* by accepting potentially malicious data. |
| --- |

Threat Level

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

Automation

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| CppCheck | 2.1+ | assertWithSideEffect | Flags assertions that contain side effects or modify program state, which is dangerous when assertions are removed. |
| SonarQube | Latest | S5835 | Highlights instances where assertions are used to validate method parameters, indicating that runtime checks should be used instead. |
| Clang Static Analyzer | Latest | core.ErrorHandling | Identifies patterns where security-sensitive logic might be tied to debugging checks like assertions. |
| Coverity | Latest | ASSERT\_SIDE\_EFFECT | Detects logic within an assertion that impacts the program's behavior in release builds. |

#### Coding Standard 7

| Coding Standard | Label | Use Exceptions for Error Handling |
| --- | --- | --- |
| Exceptions | STD-007-CPP | Exceptions are the preferred mechanism in C++ for handling errors that cannot be dealt with locally. Using exceptions correctly ensures that the program can gracefully handle unexpected conditions, like failed resource allocation or invalid file operations, without resorting to vulnerable error-prone methods like returning special values or using global flags. This standard mandates the proper use of try, catch, and throw to maintain program state integrity and prevent security flaws. |

| Noncompliant Code |
| --- |
| The code uses return values to indicate errors. This approach can be easily ignored by the caller, leading to a failure to handle the error state and potentially causing a security vulnerability or program crash. |
| cpp<br>#include <iostream><br>#include <fstream><br><br>bool open\_file(const std::string& filename) {<br> std::ifstream file(filename);<br> if (!file.is\_open()) {<br> return false; // Error indicated by return value<br> }<br> return true;<br>}<br><br>int main() {<br> if (!open\_file("nonexistent\_file.txt")) {<br> // A developer might forget to add this check, leading to an issue<br> std::cerr << "File could not be opened." << std::endl;<br> }<br> return 0;<br>} |

| Compliant Code |
| --- |
| The code uses an exception to signal a failure to open the file. This forces the calling code to handle the error within a try-catch block, ensuring that the error state is always addressed, regardless of the developer's diligence. |
| cpp<br>#include <iostream><br>#include <fstream><br>#include <stdexcept><br><br>void open\_file\_safe(const std::string& filename) {<br> std::ifstream file(filename);<br> if (!file.is\_open()) {<br> throw std::runtime\_error("Failed to open file.");<br> }<br>}<br><br>int main() {<br> try {<br> open\_file\_safe("nonexistent\_file.txt");<br> std::cout << "File opened successfully." << std::endl;<br> } catch (const std::runtime\_error& e) {<br> std::cerr << "Caught exception: " << e.what() << std::endl;<br> }<br> return 0;<br>} |

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.]  Principle 6 (Design for Secure Failure): This standard is fundamental to secure failure. Using exceptions forces the application to stop normal execution and enter a controlled error state, preventing code from continuing with corrupt or invalid data that resulted from the initial error.  Principle 9 (Do Not Trust Infrastructure): By ensuring that resource cleanup (via RAII in the catch block) and error notification are handled explicitly in the code, the application does not rely solely on the operating system to clean up or contain a crash caused by an unhandled error. |
| --- |

Threat Level

| Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- |
| Medium | Medium | Low | Medium | 3 |

Automation

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| SonarQube | Latest | S1479 | Flags functions that use return codes to indicate an error, recommending the use of exceptions instead. |
| Clang Static Analyzer | Latest | cplusplus.ExceptionSafety | Analyzes code to ensure exception safety, particularly in destructors and constructors where exceptions can cause leaks. |
| Coverity | Latest | RETURN\_CODE\_IGNORED | Identifies cases where error return codes are not checked by the caller, a common issue when exceptions are not used. |
| Parasoft C/C++test | Latest | CERT EXP02-C | Enforces rules about handling exceptions and resource management to prevent leaks during stack unwinding. |

#### Coding Standard 8

| Coding Standard | Label | Prevent Integer Conversion Errors |
| --- | --- | --- |
| [Student Choice] | STD-008-CPP | Integer conversion errors occur when a value is converted from one integer type to another, resulting in either a loss of data (truncation) or an incorrect value. This can happen when converting a signed integer to an unsigned integer, a larger integer type to a smaller one, or when performing mixed-sign arithmetic. Such errors can lead to security vulnerabilities like buffer overflows by causing a negative or large number to be incorrectly interpreted as a valid buffer size. |

| Noncompliant Code |
| --- |
| The code converts a negative signed integer to an unsigned int to use it as a buffer size. This results in a large positive value due to the conversion, which could cause a heap buffer overflow when memory is allocated. |
| cpp<br>#include <iostream><br><br>int main() {<br> int count = -10;<br> // Vulnerable: Converting a negative signed int to unsigned int<br> unsigned int size = count;<br> char\* buffer = new char[size]; // Allocated a large, unintended size<br> std::cout << "Allocated a buffer of size: " << size << std::endl;<br> // Code continues with buffer operations<br> delete[] buffer;<br> return 0;<br>} |

| Compliant Code |
| --- |
| The code performs a check to ensure the signed integer is not negative before converting it to an unsigned int for memory allocation. This prevents the dangerous conversion and potential buffer overflow. |
| cpp<br>#include <iostream><br><br>int main() {<br> int count = -10;<br> if (count < 0) {<br> std::cerr << "Error: count cannot be negative." << std::endl;<br> return 1;<br> }<br> unsigned int size = static\_cast<unsigned int>(count);<br> char\* buffer = new char[size];<br> std::cout << "Allocated a buffer of size: " << size << std::endl;<br> // Code continues with buffer operations<br> delete[] buffer;<br> return 0;<br>} |

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.]  Principle 4 (Practice Defense in Depth): This standard is a critical low-level defense. By checking for valid range and sign before conversion, it prevents numerical manipulation from leading to memory corruption, providing a layer of defense beneath buffer checks.  Principle 6 (Design for Secure Failure): When an invalid conversion is detected (e.g., a negative number for size), the compliant code safely fails by reporting the error. This is a secure failure mode, preventing the dangerous conversion that would lead to an unintended, exploitable memory allocation. |
| --- |

Threat Level

| Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | 2 |

Automation

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| CppCheck | 2.1+ | signed-unsigned-mismatch | Detects conversions between signed and unsigned integer types in contexts that can lead to unexpected values (e.g., comparison or buffer allocation). |
| SonarQube | Latest | S1488 | Flags variables declared as signed that may hold non-negative values, promoting safe use of unsigned types where appropriate. |
| Clang Static Analyzer | Latest | core.num.Conversion | Identifies implicit or explicit conversions that result in loss of precision or unexpected sign change. |
| Coverity | Latest | IMPLICIT\_CONVERSION | Warns about implicit conversions between different integral types that can lead to unexpected security flaws. |

#### Coding Standard 9

| Coding Standard | Label | Ensure Timely Release of Resources |
| --- | --- | --- |
| [Student Choice] | STD-009-CPP | Failure to properly release resources, such as file handles, network sockets, or memory, can lead to resource exhaustion, causing a denial-of-service (DoS) vulnerability. An attacker could intentionally trigger a scenario where a resource is acquired but never released, eventually consuming all available resources and causing the application or system to become unresponsive. This standard mandates that all acquired resources are released in a timely and predictable manner. |

| Noncompliant Code |
| --- |
| The program opens a file but does not explicitly close the file stream. If a subsequent function call throws an exception, the file handle will never be released, leading to a resource leak. |
| cpp<br>#include <iostream><br>#include <fstream><br>#include <string><br><br>void process\_file(const std::string& filename) {<br> std::ifstream file(filename);<br> if (file.is\_open()) {<br> // Do work here<br> // ...<br> // No file.close();<br> }<br>}<br><br>int main() {<br> process\_file("data.txt");<br> return 0;<br>} |

| Compliant Code |
| --- |
| The code uses the Resource Acquisition Is Initialization (RAII) pattern with C++ streams. When the std::ifstream object goes out of scope (either through a return or an exception), its destructor is automatically called, which closes the file and prevents resource leaks. This is the preferred method for C++. |
| cpp<br>#include <iostream><br>#include <fstream><br>#include <string><br><br>void process\_file(const std::string& filename) {<br> std::ifstream file(filename);<br> if (file.is\_open()) {<br> // Do work here<br> // file is automatically closed when `file` object goes out of scope<br> }<br> // The file is guaranteed to be closed here, even if an exception was thrown.<br>}<br><br>int main() {<br> process\_file("data.txt");<br> return 0;<br>} |

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.]  Principle 6 (Design for Secure Failure): This is a core tenet of secure failure. The RAII pattern ensures that resources (like file handles) are automatically released, even when an exception is thrown. This guarantees that the program fails cleanly without leaking resources that could lead to system instability or a DoS condition.  Principle 3 (Minimize Attack Surface): Resource exhaustion is a form of attack. By ensuring resources are promptly released, the standard minimizes the attack surface available to an adversary trying to trigger a Denial-of-Service through resource hoarding. |
| --- |

Threat Level

| Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | 2 |

Automation

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| CppCheck | 2.1+ | resourceLeak | Detects patterns where resources (like file pointers or dynamically allocated memory) are opened/acquired but not released. |
| Clang Static Analyzer | Latest | unix.ResourceLeak | Flags non-memory resource leaks, such as unclosed file descriptors, sockets, and locks. |
| SonarQube | Latest | S2093 | Identifies streams that are not properly closed, violating the core principle of timely resource release. |
| Coverity | Latest | RESOURCE\_LEAK | Tracks the lifecycle of system resources and reports when a resource is opened but the control flow bypasses the necessary close/release function. |

#### Coding Standard 10

| Coding Standard | Label | Protect Shared Data with Mutexes |
| --- | --- | --- |
| [Student Choice] | STD-010-CPP | When multiple threads access and modify the same shared data, a race condition can occur. This is where the final result depends on the non-deterministic timing of thread execution, leading to corrupted data or unexpected program behavior. An attacker could exploit this to trigger a denial of service, cause a program crash, or gain unauthorized access. This standard requires that all shared data be protected by a mutex to ensure only one thread can access the data at a time, thereby preventing race conditions. |

| Noncompliant Code |
| --- |
| The code increments a shared variable from multiple threads without any synchronization mechanism. This is a classic race condition where the final count is likely to be incorrect due to concurrent access. |
| cpp<br>#include <iostream><br>#include <thread><br><br>int shared\_counter = 0;<br><br>void increment\_counter() {<br> for (int i = 0; i < 10000; ++i) {<br> shared\_counter++; // Vulnerable to race condition<br> }<br>}<br><br>int main() {<br> std::thread t1(increment\_counter);<br> std::thread t2(increment\_counter);<br> t1.join();<br> t2.join();<br> std::cout << "Final counter value: " << shared\_counter << std::endl; // Likely not 20000<br> return 0;<br>} |

| Compliant Code |
| --- |
| The code uses a std::mutex to protect the shared counter. The std::lock\_guard ensures that the mutex is automatically locked when entering the scope and unlocked when exiting, guaranteeing atomic access to the shared variable and preventing the race condition. |
| cpp<br>#include <iostream><br>#include <thread><br>#include <mutex><br><br>int shared\_counter = 0;<br>std::mutex counter\_mutex;<br><br>void increment\_counter() {<br> for (int i = 0; i < 10000; ++i) {<br> std::lock\_guard<std::mutex> lock(counter\_mutex);<br> shared\_counter++; // Protected by the mutex<br> }<br>}<br><br>int main() {<br> std::thread t1(increment\_counter);<br> std::thread t2(increment\_counter);<br> t1.join();<br> t2.join();<br> std::cout << "Final counter value: " << shared\_counter << std::endl; // Guaranteed to be 20000<br> return 0;<br>} |

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.]  Principle 6 (Design for Secure Failure): If shared data is corrupted due to a race condition, the application's logic becomes unpredictable. This standard enforces synchronization (using mutexes/locks) to maintain data integrity, ensuring the system operates reliably and avoids an unpredictable failure state.  Principle 4 (Practice Defense in Depth): Synchronization mechanisms serve as a deep defensive layer within the application logic. They protect internal state variables from corruption, creating a strong barrier against attacks that exploit timing flaws or unpredictable behavior to compromise data. |
| --- |

Threat Level

| Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- |
| High | Medium | High | High | 2 |

Automation

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| ThreadSanitizer (TSAN) | Integrated in GCC/Clang | Runtime Flag | Runtime tool that detects race conditions, deadlocks, and common concurrency bugs during test execution with high accuracy. |
| Coverity | Latest | DEADLOCK, RACE\_CONDITION | Identifies static concurrency issues like potential deadlocks and unprotected access to shared global variables. |
| SonarQube | Latest | S2445 | Flags code where an operation on a synchronized collection is not fully encapsulated by the appropriate lock. |
| Helix QAC | 2024.1 | SEI CERT CON54-CPP | Enforces compliance with concurrency best practices, such as consistent use of locking mechanisms for shared resources. |

### 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.

The Green Pace DevOps process must be modified to integrate automated security scanning at two critical points to enforce compliance with this coding policy, ensuring that security checks are intrinsic rather than being left until the "Verify and test" phase. The most crucial point for automation is immediately after the Build phase. As soon as source code is compiled into a secure build, automated static analysis tools (SAST), such as CppCheck and SonarQube, must run. This stage is where all ten coding standards defined in this policy including those covering Data Type, Memory Protection, String Correctness, and Concurrency will be enforced. The Continuous Integration (CI) pipeline, managed by the DEV gear in the center, will be configured to automatically pull the SAST report. Any finding flagged as Critical (Level 1) or High (Level 2) by the SAST tool that violates a standard, such as an SQL Injection pattern or a Use-After-Free risk, will trigger a failure gate in the build pipeline. This prevents noncompliant code from proceeding to the next stage, effectively "shifting left" security. Additionally, the existing phase labeled Verify and test must be expanded to include dynamic security checks, run concurrently with functional testing. Tools like ASAN (AddressSanitizer) and OWASP ZAP are integrated here. ASAN is a runtime tool that must run during the functional test suite to detect low-level Memory Protection flaws that static analysis might miss. OWASP ZAP is a dynamic tool that must be executed against the built application to dynamically test for external vulnerabilities, such as SQL Injection and improper Data Value handling, by actively attempting to inject malicious payloads. By embedding these automated checks into the Build and Verify and test loops, Green Pace ensures compliance is continuous and cost-effective, adhering to a defense-in-depth security model.

### 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) | High | Medium | Medium | High | 2 |
| STD-002-CPP (Data Value) | Critical | Very Likely | Medium | Critical | 1 |
| STD-003-CPP (String Correctness) | Critical | Likely | Medium | High | 2 |
| STD-004-CPP (SQL Injection) | Critical | Very Likely | Medium | Critical | 1 |
| STD-005-CPP (Memory Protection) | Critical | Likely | High | Critical | 1 |
| STD-006-CPP (Assertions) | High | Medium | Low | Medium | 3 |
| STD-007-CPP (Exceptions) | Medium | Medium | Low | Medium | 3 |
| STD-008-CPP (Integer Security) | High | Medium | Medium | High | 2 |
| STD-009-CPP (Resource Management) | High | Medium | Medium | High | 2 |
| STD-010-CPP (Concurrency) | High | Medium | High | High | 2 |

### Create Policies for Encryption and Triple A

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

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. Encryption | Explain what it is and how and why the policy applies. |
| --- | --- |
| Encryption at rest | Encryption applied to data when it is stored and not actively being used or transmitted. This includes databases, file storage, backups, and archives.  How it is used: Data is encrypted before being written to disk and decrypted only upon retrieval for processing. This policy applies through mandatory use of AES-256 encryption on all production databases (PostgreSQL, MySQL, etc.) and cloud storage buckets (S3, GCS).  Why/When: This policy applies to all persistent storage containing PII, financial, or proprietary Green Pace data. It protects data from physical theft, unauthorized access via misconfigured access controls, and compromised database backups. |
| Encryption in flight | Encryption applied to data as it moves between two points over a network (e.g., between a user's browser and the web server, or between microservices).  How it is used: Mandatory use of Transport Layer Security (TLS 1.2 or higher) is required for all external communication (HTTPS for web apps) and internal communication (mTLS for service-to-service communication). All network ports must be checked for TLS enforcement.  Why/When: This policy applies to all data transmission over public or private networks. It prevents eavesdropping (man-in-the-middle attacks) and ensures the integrity of the data being transferred. |
| Encryption in use | Encryption applied to data while it is actively being processed or loaded into memory (RAM). This is the most complex state, often involving specialized hardware or secure enclaves.  How it is used: Developers must use modern application-layer encryption libraries (e.g., secure computation/secure multi-party computation) when handling highly sensitive data like encryption keys or user credentials in RAM. Data should be masked or tokenized immediately after use.  Why/When: This policy applies to data that must be processed but should never be exposed in plaintext in memory. It protects against advanced memory scraping attacks, cold boot attacks, and malicious insiders viewing process memory. |

| 1. Triple-A Framework\* | Explain what it is and how and why the policy applies. |
| --- | --- |
| Authentication | The process of verifying a user's identity before granting access.  How it is used: All users must successfully complete a login process using strong credentials. The policy mandates multi-factor authentication (MFA) for all administrative and production access (user logins).  Why/When: This policy applies at the start of every session (user logins, API calls) to ensure that the user accessing the system is who they claim to be, protecting the system from unauthorized entry. |
| Authorization | The process of determining what an authenticated user is permitted to do or access within the system.  How it is used: A Role-Based Access Control (RBAC) model must be implemented, where user permissions are tied to defined roles (e.g., Admin, Read-Only, Developer). Access must adhere to the principle of least privilege, specifically controlling user level of access and which files accessed by users are visible.  Why/When: This policy applies after successful authentication and must be checked before every critical action (e.g., viewing confidential files, changes to the database, or adding of new users). It prevents legitimate users from accessing or manipulating data outside their scope. |
| Accounting | The process of tracking user activities, including system access, resource changes, and attempted actions, often called auditing or logging.  How it is used: Comprehensive, immutable audit logs must be generated for all security-relevant events, including user logins, addition of new users, changes to the database, and files accessed by users. Logs must include user ID, timestamp, and action details.  Why/When: This policy applies continuously during the user session. It provides non-repudiation, allowing Green Pace to reconstruct events for incident response, regulatory compliance, and identifying malicious or suspicious activity. |

\*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

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 | 10/10/2025 | Update to Threat Level automation | Anthony McCormack |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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