**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. ValidateInput Data | All input from users, files, or external sources must be thoroughly validated to ensure it is correct, expected, and safe. This is essential to prevent vulnerabilities or unexpected behaviors caused by malicious or malformed data. |
| 1. Heed Compiler Warnings | Developers should pay close attention to compiler and static analysis warnings, treating them as potential signs of vulnerabilities. Addressing these warnings early helps prevent security flaws from reaching production. |
| 1. Architect and Design for Security Policies | Security should be a primary consideration during system architecture and design. All decisions should be guided by organizational policies that address potential threats and control access from the outset. |
| 1. Keep It Simple | Simplicity in code and design reduces the risk of vulnerabilities, makes code easier to audit, and lessens the likelihood of errors that might be overlooked in more complex systems. |
| 1. Default Deny | Systems and code should deny access by default, only allowing it when explicitly granted. This approach limits the attack surface and helps prevent unauthorized actions. |
| 1. Adhere to the Principle of Least Privilege | Both code and users should operate with the minimum permissions necessary to perform their tasks. This limits the potential damage if a vulnerability is exploited. |
| 1. Sanitize Data Sent to Other Systems | Before sending data to external systems, such as databases or web services, it must be sanitized. This prevents injection attacks and helps preserve data integrity. |
| 1. Practice Defense in Depth | Multiple layers of security controls should be implemented so that if one layer fails, others remain in place to protect the system. |
| 1. Use Effective Quality Assurance Techniques | Employing rigorous testing, code reviews, and automated analysis is vital for detecting security flaws before deployment. Quality assurance processes ensure code is both correct and secure. |
| 1. Adopt a Secure Coding Standard | Consistently applying a recognized secure coding standard, such as the SEI CERT C++ Coding Standard, helps keep the codebase robust against 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-nnn-LLL] | Using the correct data types prevents unexpected behavior, data corruption, and vulnerabilities such as buffer overflows or type confusion. Always choose types that accurately represent the data being stored and processed. |

| **Noncompliant Code** |
| --- |
| Assigning a larger value than can be held by the data type, causing overflow. |
| char buffer[10];  int i = 1234567890; // int may be too small on some platforms |

| **Compliant Code** |
| --- |
| Using proper data types and preventing overflow by checking value size. |
| char buffer[10];  long long i = 1234567890LL; // Use appropriate type for large values |

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

| **Principles(s): Least Privilege:** **Define variables and parameters using the most specific and restrictive data type appropriate for their intended use. This minimizes the risk of unintended operations, overflows, or type confusion.**  **Fail-Safe Defaults:** **Choose the safest and most suitable default data type for each context (e.g., use unsigned types for values that should never be negative, or fixed-width types for consistent behavior across platforms). Require explicit casting whenever type conversions are necessary to prevent accidental or unsafe conversions.**  **Mapping Explanation:** **Using precise data types and requiring explicit conversions ensures that only valid operations are performed, reduces the risk of accidental misuse, and helps prevent vulnerabilities such as integer overflows, underflows, or type confusion.** |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | SonarQube 9.9 LTS | SonarQube cpp 6536 | A static code analysis platform that scans source code for bugs, security vulnerabilities, code smells, and duplications. It supports many programming languages, integrates with CI/CD pipelines, and provides actionable feedback to help teams improve code quality and maintainability |
| Clang-tidy | Clang-tidy 15+ | clang-tidy: modernize-use-using, readability-implicit-bool-conversion, relevant SEI CERT/MISRA checks. | A C/C++ “linter” and static analysis tool that checks code for bugs, style issues, and best practice violations. It offers automated suggestions and can be integrated into development workflows for continuous feedback. |
| Coverity | Coverity 2024 | C++:2008, Rule 5-0-15 | A commercial static analysis tool that examines source code for defects, including memory leaks, use-after-free, and security vulnerabilities. Coverity provides detailed reports and remediation guidance, helping teams fix issues before code reaches production. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-nnn-LLL] | Always initialize variables before use and validate their values. This prevents the use of undefined or dangerous values, reducing the risk of logic errors and vulnerabilities. |

| **Noncompliant Code** |
| --- |
| No validation of user input array |
| arr[userIndex] = value; |

| **Compliant Code** |
| --- |
| Validate the index before use |
| if (userIndex >= 0 && userIndex < ARR\_SIZE) {  arr[userIndex] = value;  } else {  // Handle invalid input safely (e.g., log error, return failure).  } |

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

| **Principles(s):**  Least Privilege: Only accept and process data values that are explicitly required and permitted for the task at hand.  Fail-Safe Defaults: Establish secure default values and strict boundaries so that, if unexpected or invalid input is received, the system remains secure and behaves predictably.  Mapping Explanation: By rigorously validating the range, format, and type of all input data, we prevent logic errors, data corruption, and attacks that exploit out-of-range or malicious values. These practices help safeguard against integer overflows, underflows, and other vulnerabilities related to improper data handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | SonarQube 9.9 LTS | SonarQube 2201  Detects potential out of bounds values and missing validarions. | A static code analysis platform that scans source code for bugs, security vulnerabilities, code smells, and duplications. It supports many programming languages, integrates with CI/CD pipelines, and provides actionable feedback to help teams improve code quality and maintainability |
| cppcheck | Cppcheck 2.13 | Outofbounds  Flags unsfe assignments and improper values | Cppcheck is an open-source static analysis tool for C and C++ code. It detects bugs such as memory leaks, out-of-bounds errors, uninitialized variables, and resource leaks—issues that compilers often miss. Cppcheck is designed to minimize false positives, supports custom rules, and works out of the box with minimal setup. It’s lightweight, fast, and can be integrated into IDEs and CI pipelines for continuous code quality checks |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-nnn-LLL] | Strings should be handled with care to prevent buffer overflows, data leaks, or injection attacks. Use safe string functions and always check buffer sizes. |

| **Noncompliant Code** |
| --- |
| Using unsafe copy without buffer size check |
| char dest[16];  strcpy(dest, userInput); |

| **Compliant Code** |
| --- |
| Using safe copy with explicit size and null termination |
| strncpy(dest, userInput, sizeof(dest) - 1);  dest[sizeof(dest) - 1] = '\0'; |

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

| **Principles(s):**  Secure by Design: Always use safe string handling functions, and validate both string lengths and encodings to prevent buffer overflows and data corruption.  Least Privilege: Limit string operations to only what is necessary, and define strict maximum sizes for buffers and fields.  Mapping Explanation: By enforcing strict limits on string processing and consistently using safe string functions, we prevent vulnerabilities such as buffer overflows and memory corruption—common attack vectors for code execution or denial of service. These practices ensure that only valid, expected strings are handled, maintaining both system integrity and security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | Cppcheck 2.13 | bufferAccessOutOfBounds Flags out-of-bounds buffer accesses and unsafe string handling, helping to identify potential vulnerabilities such as buffer overflows. | Cppcheck is an open-source static analysis tool for C and C++ code. It detects bugs such as memory leaks, out-of-bounds errors, uninitialized variables, and resource leaks—issues that compilers often miss. Cppcheck is designed to minimize false positives, supports custom rules, and works out of the box with minimal setup. It’s lightweight, fast, and can be integrated into IDEs and CI pipelines for continuous code quality checks |
| SonarQube | SonarQube 9.9 LTS | Cpp 941 Detects unsafe string operations and unchecked buffer sizes. | A static code analysis platform that scans source code for bugs, security vulnerabilities, code smells, and duplications. It supports many programming languages, integrates with CI/CD pipelines, and provides actionable feedback to help teams improve code quality and maintainability |
| Clang-tidy | Clang-tidy 15+ | Cert-msc30-c, cert msc32-c | A C/C++ “linter” and static analysis tool that checks code for bugs, style issues, and best practice violations. It offers automated suggestions and can be integrated into development workflows for continuous feedback.  Pre-Production:  Comprehensive scans are performed before release to ensure that no unsafe string usage remains in the codebase. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-nnn-LLL] | Sanitize all inputs used in SQL queries or use parameterized statements to prevent SQL injection attacks, which could lead to data breaches or unauthorized access. |

| **Noncompliant Code** |
| --- |
| Unsafe concatenation of user input into SQL |
| std::string query = "SELECT \* FROM users WHERE name = '" + user\_input + "'";  db.execute(query); |

| **Compliant Code** |
| --- |
| Using parameterized queries to avoid injection. |
| std::string query = "SELECT \* FROM users WHERE name = ?";  db.prepare(query);  db.bind(1, user\_input);  db.execute(); |

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

| **Principles(s):**  Input Validation: Validate and sanitize all user input before including it in SQL queries.  Secure by Design: Always use parameterized queries or prepared statements to keep code and data separate, preventing SQL injection attacks.  Mapping Explanation: Combining input validation with parameterized queries ensures that user input cannot be executed as SQL code. This approach effectively blocks attackers from injecting malicious SQL, protecting against data theft, unauthorized access, and data manipulation. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Possible | High | Critical | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | SonarQube 9.9 LTS | Cpp 2007  Detects SQL quiries vulnerable to injection and unsafe strings concatenation in quiries | A static code analysis platform that scans source code for bugs, security vulnerabilities, code smells, and duplications. It supports many programming languages, integrates with CI/CD pipelines, and provides actionable feedback to help teams improve code quality and maintainability |
| CodeQL | CodeQL 2.13 | Flags unparameterized SQL usage and potential injection vectors | Automated tools scan for dynamic SQL construction, unsanitized input in queries, and absence of parameterization. They flag vulnerable lines and provide remediation suggestions.  Where in the process:  Create: IDE plugins highlight risky SQL operations in real time.  Verify: CI/CD static analysis blocks merges with detected injection risks.  Pre-Production: Full scans before release ensure no SQL injection vulnerabilities persist. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-nnn-LLL] | Proper memory management prevents leaks, double frees, buffer overflows, and use-after-free vulnerabilities, all of which can be exploited to compromise system security. |

| **Noncompliant Code** |
| --- |
| Freeing memory twice (double free). |
| int\* ptr = (int\*)malloc(sizeof(int));  free(ptr);  free(ptr); // Danger: double free |

| **Compliant Code** |
| --- |
| Nullifying the pointer after freeing memory. |
| int\* ptr = (int\*)malloc(sizeof(int));  free(ptr);  ptr = NULL; // Prevents double free |

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

| **Principles(s):**  Confidentiality: Protect sensitive data in memory by preventing exposure through improper handling or by ensuring no residual data remains after use.  Least Privilege: Allocate and access only the minimum amount of memory required, and always release it promptly when it is no longer needed.  Mapping Explanation: Strict memory management prevents vulnerabilities such as memory leaks, use-after-free errors, and buffer overflows. This not only improves program stability but also helps prevent data leaks and protects against potential exploitation. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Possible | MEdium | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | Valgrind 3.21 | Memcheck identifies memory leaks uninitialized memory use and invalid memory access | Valgrind is an instrumentation framework for building dynamic analysis tools, most notably used for detecting memory management and threading bugs in programs. Its primary tool, Memcheck, identifies memory leaks, use of uninitialized memory, invalid memory accesses, and improper deallocation |
| AddressSensatizer | AdrressSanitizer clang 15+ | Detects buffer overflows use after free and memory leaks at runtimes | AddressSanitizer (ASan) is a fast memory error detector built into modern C/C++ compilers like Clang and GCC. It detects buffer overflows, use-after-free, and memory leaks at runtime by instrumenting code during compilation. ASan provides detailed diagnostic reports, making it effective for catching memory safety issues during development and testing. |
| Coverity | Coverity 2024 | Resource\_Leak Use\_After\_Free | Coverity is a commercial static analysis tool that scans source code to detect a wide range of defects, including memory leaks, use-after-free, and double frees (RESOURCE\_LEAK, USE\_AFTER\_FREE checkers). It provides line-specific findings and remediation guidance, helping developers address memory management issues before code reaches production. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-nnn-LLL] | Assertions are used to check assumptions made by the program and catch logical errors during development, preventing potential vulnerabilities in production code. |

| **Noncompliant Code** |
| --- |
| Assertion with side effect |
| assert(updateState()); |

| **Compliant Code** |
| --- |
| Use assertions to validate preconditions. |
| assert(state == EXPECTED\_STATE); |

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

| **Principles(s):**  Integrity: Assertions explicitly check code assumptions and invariants, ensuring program correctness and preventing silent failures.  Fail-Safe Defaults: By enforcing checks at critical points, assertions cause the system to fail safely and visibly when unexpected conditions arise, rather than allowing it to continue in an unsafe or undefined state.  Mapping Explanation: Assertions serve as guardrails during development and testing. They make assumptions explicit and help catch errors early—before they can lead to vulnerabilities or reach production. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Insert text.]Clang-tidy | Clang-tidy 15+ | Cert-err34-c, misc-assert-side-affect  Flags improper use of assertions such as those with side affects or inccorect conditions | clang-tidy is a clang-based C++ “linter” and static analysis tool. It provides an extensible framework for diagnosing and fixing typical programming errors, such as style violations, interface misuse, and bugs detectable via static analysis. clang-tidy supports a wide range of checks, including those for security, performance, portability, and modern C++ best practices. |
| Cpp check | Cppcheck 2.13 | AssertWithSideEffects  Detects assertions that may alter program state or be improperly used | cppcheck is a static analysis tool for C and C++ code that focuses on detecting bugs and potential issues that compilers often miss. It analyzes the source code for problems such as out-of-bounds access, memory leaks, uninitialized variables, null pointer dereferences, and other logic errors. cppcheck is known for its ease of use, configurability, and ability to provide clear, actionable diagnostics to improve code quality |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-nnn-LLL] | Exception handling ensures the program can recover safely from errors, preventing undefined behavior and resource leaks. |

| **Noncompliant Code** |
| --- |
| Empty catch block and catch-all handler |
| try {  riskyOperation();  } catch (...) {  // Swallows all exceptions, no logging or handling.  } |

| **Compliant Code** |
| --- |
| Specific exception handling with logging or remediation |
| try {  riskyOperation();  } catch (const std::runtime\_error& e) {  logError(e.what());  // Take appropriate action or safely recover.  } |

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

| **Principles(s):**  Error Handling: All exceptional conditions must be explicitly handled to prevent silent failures and avoid exposing sensitive information.  Least Privilege: Exception handling code should not grant extra permissions or reveal internal details beyond what is necessary to recover from or report the error.  Mapping Explanation: Comprehensive and explicit exception handling prevents unexpected crashes, denial of service, and information leaks. Proper error management ensures that issues are detected, logged, and handled securely, supporting both application stability and security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | SonarQube 9.9 LTS | Cpp 1186  Detects empty catch blocks catch all handlers and uncaught exceptions | SonarQube is a tool that scans code to find bugs, security issues, and code quality problems. It gives developers feedback in real time and helps teams keep their code clean and secure. |
| CodeQL | CodeQL 2.13 | Flags exceptions thagt are thrown but not caught or caught but not handled properly | CodeQL lets you search your code for bugs and security vulnerabilities using custom queries. It’s great for finding complex issues that other tools might miss and can be automated in your development process. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-nnn-LLL] | Pointer misuse can lead to severe vulnerabilities such as dangling pointers, use-after-free, and buffer overflows. Always initialize pointers, check for nullptr, and prefer smart pointers in C++. |

| **Noncompliant Code** |
| --- |
| Dereferencing a pointer without checking for nullptr. |
| int\* ptr = nullptr;  int value = \*ptr; // Undefined behavior |

| **Compliant Code** |
| --- |
| Checking for nullptr before dereferencing. |
| int\* ptr = nullptr;  if (ptr) {  int value = \*ptr;  } |

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

| **Principles(s):**  Secure by Design: Always perform bounds checks on memory and buffer operations to prevent out-of-bounds reads and writes.  Least Privilege: Access only the memory required for each operation, and always validate external or user-controlled sizes before use.  Mapping Explanation: Enforcing strict bounds checking and memory access control protects against buffer overflows, which are a common source of code execution, privilege escalation, and data leaks. These practices keep code resilient against both accidental mistakes and malicious attacks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| AddressSanatizer | AddressSanatizer clang 15+ | Detects memory errors such as buffer overflows, use-after-free, and memory leaks at runtime when code is compiled and run with the -fsanitize=address flag | Finds memory errors like buffer overflows and use-after-free bugs while your program runs. |
| Clang-tidy | Clang-tidy 15+ | Cert-msc50-c, cert-msc51-c | Checks your C++ code for bugs, style issues, and best practice violations. |
| cppcheck | Cppcheck 2.13 | Buffer access out of bounds, out of bounds.  Detects out of bounds reads and writes also, unsafe pointer arithmetic | Scans C and C++ code for bugs and potential problems that compilers might miss. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-nnn-LLL] | All programs must validate user input to ensure it meets the application's requirements, preventing injection, overflow, and other attacks. |

| **Noncompliant Code** |
| --- |
| Accepting unchecked user input. |
| std::cin >> age;  // No validation |

| **Compliant Code** |
| --- |
| Validating input range. |
| std::cin >> age;  if (age < 0 || age > 120) {  std::cerr << "Invalid age." << std::endl;  } |

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

| **Principles(s):**  Least Privilege: Only accept input that matches explicitly defined allowed formats, values, or characters (whitelists).  Input Validation: Strictly validate all inputs against these whitelists before processing, and reject or sanitize anything not permitted.  Mapping Explanation: Whitelisting input is more secure than blacklisting because it ensures that only approved data is accepted. This approach greatly reduces the risk of injection attacks, logic errors, and unexpected behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | SonarQube 9.9 LTS | Cpp 5867  Flags missing improper input validations especially for whitelisting and sanitization | Scans code for bugs, security issues, and code quality problems. Supports many languages and integrates with CI/CD to give developers feedback and improve code reliability |
| CodeQL | CodeQL 2.13 | Detects places where user input is not properly properly checked against a whitelist | Lets you search code for bugs and security flaws using custom queries. Helps find complex vulnerabilities by analyzing code as data. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | [STD-nnn-LLL] | Resources such as files, sockets, and memory must be properly acquired and released to prevent leaks, denial of service, or corruption. |

| **Noncompliant Code** |
| --- |
| Hardcoded password and insecure memory handling |
| std::string password = "SuperSecret123";  log("User password: " + password); |

| **Compliant Code** |
| --- |
| Prompt user, securely zero memory after use, and never log sensitive data. |
| std::string password = promptForPassword();  // ... use password securely ...  secureZeroMemory(&password[0], password.size()); |

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

| **Principles(s):**  Confidentiality: Always protect sensitive data (like passwords, cryptographic keys, and personal information) in memory, during transmission, and in storage.  Least Privilege: Only allow access to sensitive data for code and personnel who absolutely need it for essential tasks.  Mapping Explanation: Strictly controlling and limiting access to sensitive data helps prevent leaks, breaches, and unauthorized access—ensuring privacy, security, and regulatory compliance. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Possible | High | Critical | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | SonarQube 9.9 LTS | Cpp 2068  Detects hardcoded credentials and improper handling of sensitive data | Scans code for bugs, security vulnerabilities, code smells, and secret exposures using static analysis. |
| CodeQL | CodeQL 2.13 | Flags hardcoded secrets and insecure data handling | Lets you query codebases to find bugs and security flaws using custom, database-like queries. |
| Secret Scanner | Advanced Security Secret Scanning | Detect credentials API keys and secrets checked int osource control | Searches code for hardcoded secrets like passwords, API keys, and tokens to prevent accidental leaks. |

### 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.

## Infinity Loop Diagram: DevSecOps Automation

The infinity loop (figure-eight) visually represents the continuous, iterative nature of secure software development in DevSecOps. Each phase—Assess & Plan, Design & Build, Test, Deploy, Monitor—is interconnected, with security practices embedded throughout. DevSecOps sits at the center, making security an ongoing, integrated process.

## How Automation Fits In

* Continuous Security:  
  Security checks are automated at every phase—planning, building, testing, deploying, and monitoring—ensuring all changes are evaluated against security standards.
* Enforcement Points:
  + Assess & Plan: Automated scans for compliance risks in requirements and architecture.
  + Design & Build: Static analysis, dependency checks, and linting block insecure code as it’s written.
  + Test: Automated security tests (dynamic analysis, vulnerability scans, fuzzing) catch issues before deployment.
  + Deploy: Policy-as-code tools validate infrastructure and deployment scripts automatically.
  + Monitor: Ongoing automated monitoring and alerting for post-deployment security deviations.
* Central Role of DevSecOps:  
  DevSecOps synchronizes development, operations, and security. Automation ensures every feedback loop—code changes, deployments, incidents—triggers the right security checks and responses.
* Feedback & Remediation:  
  Automated feedback ensures any security violation prompts immediate remediation, keeping the system secure and improving over time.

## Guidance for Automating DevOps Security at Green Pace

1. Integrate Security Tools:  
   Add static analysis, secret scanning, dependency checks, and policy-as-code tools at each pipeline stage.
2. Automate Gates:  
   Use automated checks to block non-compliant code from merging or deploying.
3. Centralize Reporting:  
   Feed all tool results into a central dashboard for visibility and compliance tracking.
4. Continuous Feedback:  
   Automatically notify teams of issues for quick remediation and secure coding culture.
5. Policy as Code:  
   Define security standards as code for automatic enforcement, versioning, and auditing.

Summary:  
Automation is the backbone of DevSecOps, ensuring security is continuous, enforceable, and always improving—just as the infinity loop diagram illustrates.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Possible | Medium | High | 2 |
| STD-003-CPP | Critical | Possible | High | Critical | 2 |
| STD-004-CPP | High | Possible | Medium | High | 2 |
| STD-005-CPP | Medium | Unlikely | Low | Medium | 3 |
| STD-006-CPP | High | Unlikely | Medium | High | 2 |
| STD-007-CPP | critical | Possible | High | Critical | 1 |
| STD-008-CPP | High | Possible | Medium | High | 2 |
| STD-009-CPP | Critical | Possible | High | Critical | 1 |
| STD-010-CPP | Critical | Possible | High | Critical | 1 |

### 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 | Encrypts stored data (disk, database, backups) to protect if storage is lost or stolen. Use strong algorithms (e.g., AES-256) for all stored data. Manage and rotate keys securely. Any time data is stored—on disk, in databases, backups, or cloud. |
| Encryption in flight | Encrypts data during transmission to prevent interception or eavesdropping. Require secure protocols (TLS 1.2+, SSH, HTTPS). Disable insecure ones and enforce certificate checks. All data sent over networks, including logins, APIs, and internal traffic. |
| Encryption in use | Protects data while it’s being processed (in memory), reducing risk from memory attacks. Use Trusted Execution Environments, memory encryption, and minimize time data is unencrypted. When handling highly sensitive data, especially in shared/cloud environments. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Verifies identity before granting access. Prevents unauthorized logins. Enforce multi-factor authentication, strong passwords, and central identity providers. For all logins—users, admins, service accounts, API clients. |
| Authorization | Grants or denies access based on roles/permissions. Enforces least privilege. Use role-based access control (RBAC), assign minimum privileges, log changes. When accessing files, apps, databases, or performing privileged actions. |
| Accounting | Tracks/logs user activities for audits and incident response. Log all significant events, review logs regularly, retain logs for compliance. For all sensitive/admin actions and where compliance is needed. |

**\***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

STD-001-CPP: Enforcing Correct Data Values

* Principles Applied: 1 (Least Privilege), 3 (Secure by Design), 7 (Reduce Attack Surface)
* 1 (Least Privilege): By enforcing strict data value limits, the code ensures that only necessary access and operations are allowed, reducing the risk of unauthorized actions.
* 3 (Secure by Design): Integrating data validation from the start helps prevent vulnerabilities by design, rather than as an afterthought.
* 7 (Reduce Attack Surface): Limiting what data can be processed or accessed reduces the number of ways an attacker can exploit the system.
* STD-002-CPP: String Correctness
* Principles Applied: 1 (Least Privilege), 3 (Secure by Design), 4 (Fail Securely)
* 1 (Least Privilege): Proper string handling prevents attackers from exploiting overflows to gain unauthorized access.
* 3 (Secure by Design): Secure string operations are incorporated as a fundamental part of the code, not as a patch.
* 4 (Fail Securely): If string operations fail, the system is designed to do so safely, without exposing sensitive data or system integrity.
* STD-003-CPP: SQL Injection Prevention
* Principles Applied: 1 (Least Privilege), 2 (Defense in Depth), 3 (Secure by Design), 8 (Mediation of Inputs)
* 1 (Least Privilege): Restricting database permissions limits the impact of a successful attack.
* 2 (Defense in Depth): Multiple layers of validation and sanitization protect against SQL injection at different points in the data flow.
* 3 (Secure by Design): Input validation and parameterized queries are built into the code from the start.
* 8 (Mediation of Inputs): All user inputs are checked and controlled before interacting with the database, preventing malicious data from being processed.
* STD-004-CPP: Memory Protection
* Principles Applied: 1 (Least Privilege), 3 (Secure by Design), 7 (Reduce Attack Surface)
* 1 (Least Privilege): Only necessary memory is allocated and accessed, reducing the risk of unauthorized memory manipulation.
* 3 (Secure by Design): Safe memory management practices are integrated into the system’s design.
* 7 (Reduce Attack Surface): Protecting memory boundaries and access points minimizes opportunities for exploitation.
* STD-005-CPP: Assertions
* Principles Applied: 3 (Secure by Design), 4 (Fail Securely), 6 (Keep It Simple)
* 3 (Secure by Design): Assertions help enforce correct program logic, catching errors early by design.
* 4 (Fail Securely): When an assertion fails, the system is designed to fail in a controlled, safe manner.
* 6 (Keep It Simple): Assertions simplify logic and make code easier to understand and maintain, reducing the chance of hidden bugs.
* STD-006-CPP: Exception Handling
* Principles Applied: 3 (Secure by Design), 4 (Fail Securely), 10 (Accountability)
* 3 (Secure by Design): Exception handling is implemented to ensure errors are managed securely.
* 4 (Fail Securely): The system handles exceptions without exposing sensitive data or leaving the system in an unsafe state.
* 10 (Accountability): Consistent logging of exceptions supports tracking, auditing, and accountability.
* STD-007-CPP: Buffer Overflow Prevention
* Principles Applied: 1 (Least Privilege), 2 (Defense in Depth), 3 (Secure by Design), 7 (Reduce Attack Surface)
* 1 (Least Privilege): Limiting buffer sizes and access rights reduces the risk of exploitation.
* 2 (Defense in Depth): Multiple safeguards (such as bounds checking and safe libraries) provide layered protection.
* 3 (Secure by Design): Buffer overflow prevention is built into the code from the outset.
* 7 (Reduce Attack Surface): Preventing overflows reduces the number of vulnerabilities available to attackers.
* STD-008-CPP: Input Whitelisting
* Principles Applied: 1 (Least Privilege), 3 (Secure by Design), 8 (Mediation of Inputs)
* 1 (Least Privilege): Only approved, necessary inputs are permitted, minimizing risk.
* 3 (Secure by Design): Input validation is part of the initial design, not an afterthought.
* 8 (Mediation of Inputs): All user input is validated and mediated before use, preventing harmful data from entering the system.
* STD-009-CPP: Sensitive Data Protection
* Principles Applied: 1 (Least Privilege), 3 (Secure by Design), 8 (Mediation of Inputs)
* 1 (Least Privilege): Access to sensitive data is tightly controlled and limited to only those who need it.
* 3 (Secure by Design): Data protection is a core part of the system’s architecture.
* 8 (Mediation of Inputs): All access to sensitive data is checked and controlled, preventing unauthorized exposure.
* STD-010-CPP: Sensitive Data Protection (Extended)
* Principles Applied: 1 (Least Privilege), 3 (Secure by Design), 4 (Fail Securely), 10 (Accountability)
* 1 (Least Privilege): Sensitive data is only accessible to those with explicit permission.
* 3 (Secure by Design): The system is architected to protect sensitive data at every layer.
* 4 (Fail Securely): If a failure occurs, sensitive data is not exposed or put at risk.
* 10 (Accountability): All access and actions involving sensitive data are logged for auditing and accountability.

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 | 06/15/2025 | Second Draft | Teisha Yoder |  |
| [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 |