**Green Pace Developer: Security Policy Guide Template  
Buxton McCaslin Module 6 Assignment**



# 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 | It’s always a good idea to double-check any data that comes into your system, especially if it’s from an outside source. You never really know what users, files, or third-party APIs might be sending your way, so validating input like checking lengths, formats, and expected values helps prevent a lot of issues. Things like SQL injection or buffer overflows often happen because the data wasn’t checked properly up front. This step might seem small, but it’s one of the easiest ways to catch problems early and keep your app secure. |
| 1. Heed Compiler Warnings | Compiler warnings aren’t just noise, they’re usually trying to tell you something important. If the compiler thinks your code is risky or might not behave as expected, it's worth paying attention. Things like type mismatches or uninitialized variables can slip through if ignored, and those often lead to weird bugs or even serious vulnerabilities. Turning on all warnings and treating them like errors might feel strict, but it helps you write cleaner and more secure code from the start. |
| 1. Architect and Design for Security Policies | Priority issues like SQL injection and unsafe strings are both high risk and common, so they go right to the top.  The Likely group covers things we’ll probably run into, like range issues or exceptions, but they’re not always catastrophic.  Low priority stuff like assertions or credentials still matter, but they’re usually easier to catch and fix early.  And finally, things like pointer initialization or input size checks are important but rare enough to be considered unlikely. |
| 1. Keep It Simple | The simpler the code, the fewer things that can go wrong. Complexity might look clever, but it often hides bugs and makes maintenance harder for everyone. If you can do something in a straightforward way, do it. That applies to logic, structure, naming, and basically everything. Clean and simple code is easier to read, test, and secure, which makes life better for you and your teammates. |
| 1. Default Deny | When it comes to permissions, it’s safest to assume “no” until there’s a clear reason to say “yes.” If you leave everything open by default, it’s easy to miss something and accidentally give access where it’s not needed. Instead, explicitly grant access only when it’s justified. This applies to files, APIs, network ports, and anything else a user or system might interact with. It’s a simple rule that helps create a strong security foundation. |
| 1. Adhere to the Principle of Least Privilege | Less access means less risk. Whether it’s a user, a service, or a piece of code, it should only be able to do what it absolutely needs to do, and nothing more. That way, if something goes wrong or gets compromised, the damage stays limited. It's kind of like giving someone a spare key to your house, but only for the rooms they need. This principle keeps systems more secure by limiting how much access each part of the system has. |
| 1. Sanitize Data Sent to Other Systems | Just like we validate incoming data, we also need to clean up anything we send out. If your app is passing data to a database, another service, or even writing it to a file, make sure it’s safe and properly formatted. Otherwise, you might end up unintentionally injecting code or corrupting someone else’s system. Sanitizing output keeps everything in the communication chain stable and secure. |
| 1. Practice Defense in Depth | No single security measure is perfect, so it's smart to use multiple layers of protection. Think of it like a safety net. If one control fails, another can catch the issue. You might have input validation, access controls, logging, and encryption all working together. This layered approach gives attackers a harder time, and it gives you more chances to stop or detect something before it becomes a bigger problem. |
| 1. Use Effective Quality Assurance Techniques | Good testing and review practices help catch bugs and security issues before they make it to production. Code reviews, static analysis, unit tests, and dynamic testing are all useful tools to spot problems early. The more consistent and thorough your quality checks are, the less likely it is that major vulnerabilities will slip through. Effective QA helps you build trust in the code and gives your team confidence in what you ship. |
| 1. Adopt a Secure Coding Standard | Using a secure coding standard, like the SEI CERT guidelines, helps your whole team stay consistent and avoid common mistakes. These standards capture proven best practices and help reduce the chances of introducing security flaws. Following a shared standard also makes code easier to review, maintain, and audit. It’s a practical way to build security into your workflow from the start. |

### 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 – Range Verification** | STD-001-  CPP | Range Verification: When working with arrays or containers in C++, it’s easy to forget how dangerous it can be to access an index without checking if it’s valid. If the index is out of range, you might run into memory corruption, crashes, or worse, open your program to buffer overflow attacks. This standard is here to remind us to always verify the range of values before we use them. It’s one of those habits that might feel tedious at first, but in the long run, it protects the integrity of your code and makes debugging so much easier. |

| **Noncompliant Code** |
| --- |
| This code tries to access an array index that’s outside its valid range, which can lead to undefined behavior, memory corruption, or even crashes. |
| int main() {  int myArray[5] = {1, 2, 3, 4, 5};  // Trying to access an index beyond the array's range  int outOfRangeIndex = 10;  std::cout << myArray[outOfRangeIndex] << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| This version adds a simple range check before using the index. It’s a small step that keeps your program safe and predictable. |
| int main() {  int myArray[5] = {1, 2, 3, 4, 5};  int outOfRangeIndex = 10;  if (outOfRangeIndex >= 0 && outOfRangeIndex < 5) {  std::cout << myArray[outOfRangeIndex] << std::endl;  } else {  std::cout << "Invalid index!" << std::endl;  }  return 0;  } |

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

| **Principles(s):**  **Validate Input Data:** This standard ensures we verify that indexes fall within valid bounds before using them. Invalid or unchecked values can easily lead to out-of-range errors and buffer overflows.  **Practice Defense in Depth:** Range checks add a defensive layer to prevent a class of memory vulnerabilities. Even if one validation is missed earlier in the code flow, this serves as a safety net. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12 | arrayIndexOfBounds | Detects out-of-bound index usage in arrays and containers. |
| SonarQube | 10.3 | Cpp:S3518 | Warns when arrays or containers are accessed without bounds checks. |
| Clang-Tidy | 16.0 | Clang-analyzer-core.NullDereference | Catches possible invalid memory access due to unverified indices. |
| Visual Studio Analyzer | 17.8 | Code Analysis Rule CA1508 | Warns when conditions are always true/false, helping to catch logic errors. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value - Data Value Validation** | STD-002-CPP | Data Value Validation: When you're working with user input or values coming from any external source, it's important to validate those values before using them in your logic. If the input isn't what you expect—for example, if someone enters a number that's way out of range—it can cause crashes, undefined behavior, or even open your code up to exploits. This standard is about making sure we always check that our data values are safe and reasonable before using them in things like array indexing, calculations, or control flow. It’s one of those “better safe than sorry” practices that helps keep code stable and secure. |

| **Noncompliant Code** |
| --- |
| This code accepts a value from the user but doesn’t check if it’s valid before using it to access an array. That can easily lead to memory issues or unexpected crashes. |
| int arr[10];  int index = getUserInput(); // Assume user enters a value  arr[index] = 5; // Dangerous if index is invalid |

| **Compliant Code** |
| --- |
| Here we add a simple validation check to make sure the index is within the bounds of the array before using it. This helps prevent out-of-bounds errors and keeps the program safe. |
| int arr[10];  int index = getUserInput(); // Assume user enters a value  if (index >= 0 && index < 10) {  arr[index] = 5;  } else {  std::cout << "Invalid index." << std::endl;  } |

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

| **Principles(s):**  **Validate Input Data:** This standard directly supports input validation. It ensures that any values coming from users or external systems are checked before they’re used in logic or calculations.  **Use Effective Quality Assurance Techniques:** Proper data validation is testable and enforceable through unit tests and static analysis tools, which help prevent invalid values from causing logic or runtime errors. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Likely | Low | Medium | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12 | unreadVariable, arrayIndexOutOfBounds | Flags unchecked or misused variables in logic that affect value use. |
| SonarQube | 10.3 | cpp:S3518 | Ensures array access is checked and warns on suspicious input handling. |
| Clang-Tidy | 16.0 | |  | | --- | |  |  |  | | --- | | clang-analyzer-core.CallAndMessage | | Warns when values are used unsafely or without being validated. |
| Visual Studio Analyzer | 17.8 | C6385 and C6386 | Detects potential buffer overflows from unchecked user input. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | String Correctness: Working with C-style strings requires a little extra care. Functions like strcpy() might seem convenient, but they don’t check if the destination buffer is big enough, and that can lead to classic issues like buffer overflows or memory corruption. These types of bugs are especially dangerous because they’re often exploited in security attacks. This standard is all about handling strings in a safe, predictable way—using bounds-checked functions, watching buffer sizes, and always making sure strings are properly null-terminated. It’s a small price to pay for safer, more reliable code. |

| **Noncompliant Code** |
| --- |
| This example uses strcpy() to copy a string that’s too long for the destination buffer. There’s no size check, which makes it unsafe and error-prone. |
| char dest[10];  strcpy(dest, "This string is too long"); |

| **Compliant Code** |
| --- |
| This version uses strncpy() to limit how many characters are copied and adds a null terminator manually, just to be sure. It’s a much safer way to work with C-style strings. |
| char dest[10];  strncpy(dest, "Safe", sizeof(dest) - 1);  dest[sizeof(dest) - 1] = '\0'; // Ensure null termination |

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

| **Principles(s):**  **Sanitize Data Sent to Other Systems:** Ensures output strings do not overflow or corrupt memory, especially when passed to other components or services.  **Adopt a Secure Coding Standard:** This standard follows SEI CERT guidelines for safe string handling in C++, using bounds-checked methods like strncpy and avoiding unsafe functions like strcpy. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12 | bufferAccessOutOfBounds | Flags use of unsafe string functions and potential overflows. |
| SonarQube | 10.3 | cpp:S1751 | Warns against unchecked calls to functions like strcpy. |
| Clang-Tidy | 16.0 | cert-str34-c | Ensures safe handling of null-terminated byte strings (based on SEI CERT rules) |
| Visual Studio Analyzer | 17.8 | C6386 | Detects out-of-bound writes in string operations. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | SQL Injection Prevention: SQL injection is one of the most well-known and dangerous security vulnerabilities out there, and it usually happens when user input is directly inserted into a query without any validation or sanitization. Attackers can manipulate those queries to read or even modify data they shouldn’t have access to. The fix is straightforward: always use parameterized queries or prepared statements. By separating the data from the query logic, you can stop malicious input from ever changing how the SQL behaves. It’s a simple change that makes a huge difference in keeping your database safe. |

| **Noncompliant Code** |
| --- |
| In this version, user input is directly concatenated into the SQL query, which leaves the system wide open to injection attacks. |
| std::string username = getUserInput();  std::string query = "SELECT \* FROM users WHERE name = '" + username + "'";  executeQuery(query); |

| **Compliant Code** |
| --- |
| This code uses a parameterized query (prepared statement), which keeps user input and SQL logic separate. That way, no matter what the user types, it’s treated as data—not executable SQL. |
| PreparedStatement\* ps = conn->prepareStatement("SELECT \* FROM users WHERE name = ?");  ps->setString(1, username);  ResultSet\* rs = ps->executeQuery(); |

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

| **Principles(s):**  **Validate Input Data:** Prevents malicious input from being used to alter SQL queries by ensuring data is clean before it’s passed into a query.  **Sanitize Data Sent to Other Systems:** Makes sure outbound data—like SQL commands—is safe and handled using secure methods (parameterized queries).  **Practice Defense in Depth:** Even if one layer (like UI input validation) fails, using prepared statements protects the database. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.3 | cpp:S3649 | Detects unparameterized queries and direct string concatenation in SQL logic. |
| Clang-Tidy | 16.0 | security-unchecked-return | Warns when user input is passed directly without validation. |
| Fortify SCA | 23.1 | SQL Injection (Data Flow Analyzer) | Identifies tainted data being used in dynamic SQL queries |
| Visual Studio Analyzer | 17.8 | CA2100 | Warns when SQL commands are built via string concatenation. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Memory Protection: Managing memory manually in C++ gives you a lot of control, but with that control comes a big responsibility. Mistakes like accessing memory that’s already been freed or forgetting to initialize a pointer can lead to nasty issues like crashes, memory leaks, or even serious security vulnerabilities. This standard is all about being careful and intentional with how we allocate, use, and clean up memory. That means making sure pointers are properly initialized, deleted when no longer needed, and then immediately set to nullptr so we don’t accidentally use them again. It’s a solid habit that helps prevent use-after-free bugs and keeps your code stable. |

| **Noncompliant Code** |
| --- |
| This code creates a pointer, deletes it, and then tries to use it again. That’s undefined behavior and a common source of security bugs. |
| int\* ptr = new int(42);  delete ptr;  \*ptr = 99; // Use-after-free vulnerability |

| **Compliant Code** |
| --- |
| In this version, we still delete the pointer, but we immediately set it to nullptr to make sure we don’t accidentally use it again. It’s a quick, safe habit that avoids a lot of trouble. |
| int\* ptr = new int(42);  delete ptr;  ptr = nullptr; // Safe: prevents further use |

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

| **Principles(s):**  **Practice Defense in Depth:** Improper memory access can cause cascading vulnerabilities. Using techniques like smart pointers and memory sanitization adds layered protection.  **Adopt a Secure Coding Standard:** SEI CERT strongly recommends safe memory handling, including automatic memory management, proper deallocation, and initialization of pointers.  **Fail Securely:** Prevents access to freed, uninitialized, or invalid memory that could lead to undefined behavior or crashes. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 16.0 | clang-analyzer-cplusplus .NewDeleteLeaks | Detects memory leaks due to unmatched new/delete operations. |
| Cppcheck | 2.12 | memoryLeak | Scans for potential leaks and bad deallocations in manual memory management. |
| Valgrind | 3.21 | memcheck | Runtime tool that detects invalid memory access and use of uninitialized memory |
| SonarQube | 10.3 | cpp:S5247 | Flags dangerous usage of raw pointers, recommends using smart pointers instead |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Proper Use of Assertions: Assertions are a great tool during development because they help you catch logic errors and assumptions that might not always hold true. But they’re not a substitute for real error handling, especially in production environments. In release builds, assertions can be disabled entirely, which means any checks you rely on might silently disappear. That’s risky. This standard encourages using assertions for internal sanity checks while actively developing, but when it comes to validating input or handling unexpected situations in production, always use proper runtime error handling. That way, your application can fail gracefully instead of crashing or behaving unpredictably. |

| **Noncompliant Code** |
| --- |
| This example uses an assertion to check user input, which is risky because assertions can be disabled in production builds—allowing bad data to slip through. |
| void process(int age) {  assert(age > 0); // Dangerous in production  std::cout << "Age: " << age << std::endl;  } |

| **Compliant Code** |
| --- |
| This version replaces the assertion with proper input validation and throws an exception if the value is invalid. It’s much safer and reliable for real-world use. |
| void process(int age) {  if (age <= 0) {  throw std::invalid\_argument("Age must be positive");  }  std::cout << "Age: " << age << std::endl;  } |

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

| **Principles(s):**  **Fail Securely:** Assertions help ensure a system fails in a predictable and controlled way if assumptions are violated.  **Use Effective Quality Assurance Techniques:** Assertions are a form of self-checking code that validates internal assumptions, improving code reliability during testing and debugging.  **Promote Accountability:** When assertions are used consistently, they make it easier to trace the root cause of logic failures and track responsibility. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 16.0 | cert-err33-c | Warns against relying on assert() for runtime validation or user input checks. |
| SonarQube | 10.3 | cpp:S5869 | Detects unsafe use of assertions that could crash the application |
| Visual Studio Analyzer | 17.8 | C26466 | Warns when assert() may cause control flow issues or logic inconsistency |
| Cppcheck | 2.12 | assertUsage | Can be extended to flag assert misuse during non-debug builds. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Exception Handling: Handling exceptions properly is all about knowing what could go wrong and responding to it in a meaningful way. When we catch everything without checking what actually happened—or worse, when we ignore exceptions entirely—we're not just missing bugs, we're potentially hiding serious problems. That can leave the system in a bad state or make it impossible to troubleshoot issues down the line. This standard is about being specific when catching exceptions, logging errors clearly, and making sure the program stays stable when something unexpected happens. It's all about giving yourself and your team the visibility you need to handle problems the right way. |

| **Noncompliant Code** |
| --- |
| This version catches every exception but does absolutely nothing with it. That’s a big risk—it swallows real errors and leaves you in the dark. |
| try {  riskyOperation();  } catch (...) {  // Swallowing the error — insecure and unsafe  } |

| **Compliant Code** |
| --- |
| This example catches a specific type of exception and logs an error message. That way, if something does go wrong, you'll know exactly what happened. |
| try {  riskyOperation();  } catch (const std::exception& e) {  std::cerr << "Operation failed: " << e.what() << std::endl;  } |

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

| **Principles(s):**  **Fail Securely:** Proper exception handling ensures the system can gracefully recover from unexpected situations without exposing sensitive operations or crashing.  **Adopt a Secure Coding Standard:** SEI CERT emphasizes consistent and predictable exception handling, especially avoiding “error hiding” or empty catch blocks.  **Promote Accountability:** Structured exceptions help developers pinpoint where and why a failure happened, improving traceability and accountability. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 16.0 | cert-err09-cpp | Warns about using exception handling inconsistently or catching generic exceptions |
| SonarQube | 10.3 | cpp:S1181 | Detects catch blocks that are too broad or suppress all exceptions |
| Visual Studio Analyzer | 17.8 | C6285 | Flags empty or logically flawed catch blocks |
| Cppcheck | 2.12 | unusedCatch | Identifies catch blocks that don’t take any action or are always bypassed |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Pointer Initialization | STD-008-CPP | Pointer Initialization: Working with pointers in C++ gives us a lot of flexibility, but it also comes with some serious risks if we’re not careful. One of the biggest mistakes is using a pointer before it’s been initialized. If that pointer happens to reference some random spot in memory, the program could crash, corrupt data, or behave in totally unexpected ways. This standard reminds us to always initialize pointers when we declare them—ideally to nullptr—so we can safely check before using them. It’s a simple habit that helps prevent some of the most frustrating and dangerous bugs. |

| **Noncompliant Code** |
| --- |
| In this example, the pointer is declared but never initialized. Using it leads to undefined behavior, which can easily crash your program or cause silent data corruption. |
| int\* ptr;  \*ptr = 10; // Undefined behavior |

| **Compliant Code** |
| --- |
| Here, the pointer is initialized to nullptr, and we check before using it. That way, we avoid undefined behavior and make the code more predictable and safe. |
| int\* ptr = nullptr;  if (ptr != nullptr) {  \*ptr = 10;  } else {  std::cout << "Pointer is null." << std::endl;  } |

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

| **Principles(s):**  **Protect Data:** Encryption at rest ensures sensitive data stored on disk (like databases, configuration files, or logs) is unreadable without the proper keys.  **Practice Defense in Depth:** Even if physical or unauthorized access to storage occurs, encryption provides a critical safeguard.  **Minimize Attack Surface:** By encrypting data before it hits disk, you reduce the exposure window for attackers who may gain low-level system access |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| VeraCrypt | 1.26 | N/A | Full-disk encryption tool for secure file and disk storage |
| AWS KMS / Azure Key Vault | Varies | Policy configuration tools | Automatically enforce key usage policies and audit encryption key access |
| Fortify SCA | 23.1 | Insufficient Encryption | Flags data written to persistent storage without encryption |
| SonarQube | 10.3 | cpp:S5542 | Detects hardcoded keys and missing encryption in file-writing code |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Buffer Size Checking | STD-009-CPP | Input Buffer Size Checking: When you're working with fixed-size buffers, one of the biggest risks is reading in more data than the buffer can actually hold. If that happens, you can end up with a buffer overflow—which can crash your program or leave it vulnerable to attacks. It’s a classic security issue in C++, and one of the most avoidable. This standard emphasizes always checking how much data is coming in and making sure it fits within the allocated buffer. A simple bounds check or using safe input functions goes a long way toward keeping your application secure and stable. |

| **Noncompliant Code** |
| --- |
| This version uses std::cin to read user input into a fixed-size buffer, but there’s no size restriction—making it a prime target for buffer overflows. |
| char buffer[10];  std::cin >> buffer; // No size check |

| **Compliant Code** |
| --- |
| Here, we use std::cin.get() with a defined size limit. It’s a safer way to handle user input and protects against buffer overflows. |
| char buffer[10];  std::cin.get(buffer, sizeof(buffer)); // Safe input handling |

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

| **Principles(s):**  **Protect Data:** Encryption in flight ensures data transmitted over a network is unreadable to unauthorized interceptors.  **Practice Defense in Depth:** Even if the network layer is compromised, encryption adds a protective boundary between endpoints.  **Secure the Weakest Link:** Network transmission is one of the most vulnerable points in a system, and encrypting it reduces the attack surface significantly. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OpenSSL | 3.2.0 | TLS handshake logging & auditing | Ensures proper certificate handling and TLS configuration. |
| Wireshark | 4.2.0 | Traffic inspection filters | Detects unencrypted traffic or improperly configured HTTPS endpoints |
| SonarQube | 10.3 | cpp:S4423 | Flags usage of insecure network functions like send() without encryption |
| Fortify SCA | 23.1 | Insecure Transport | Warns when data is transmitted without encryption over sockets or HTTP |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Avoid Hardcoded Credentials | STD-010-CPP | Avoid Hardcoded Credentials: Hardcoding credentials like passwords or API keys directly into your source code might feel convenient at first, but it’s one of the riskiest things you can do in terms of security. If someone gets access to your code—whether through version control, a build artifact, or reverse engineering—they also get access to those secrets. This standard is all about keeping sensitive information outside of your code. A much safer approach is to use environment variables or encrypted config files so that credentials are never stored in plain sight. It’s a small change that seriously improves the security of your application. |

| **Noncompliant Code** |
| --- |
| This example stores a password directly in the source code. If this file ends up in a shared repo or compiled binary, that password becomes easy to find and exploit. |
| const char\* password = "admin123"; // Insecure |

| **Compliant Code** |
| --- |
| In this version, the password is pulled from an environment variable. This keeps it out of the codebase and makes it easier to rotate or manage securely. |
| #include <cstdlib>  #include <string>  std::string password = std::getenv("APP\_PASSWORD"); // Secure method |

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

| **Principles(s):**  **Protect Data:** Encryption in use keeps data secure even while it’s actively being processed in memory, particularly in multi-tenant or cloud environments.  **Minimize Attack Surface:** Helps reduce risk from memory scraping, side-channel attacks, or unauthorized runtime access.  **Practice Defense in Depth:** Adds another layer to protect sensitive information in the rare but real case where an attacker bypasses other security controls. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Intel SGX SDK | 2.22 | N/A | Enables encrypted memory enclaves for processing sensitive data securely |
| Microsoft Azure Confidential Compute | Current | Audit Logs / Policy Tools | Ensures virtual machines encrypt memory during use in cloud computing environments |
| Fortify SCA | 23.1 | Insecure Memory Handling | Flags operations on sensitive data without secure memory management techniques |
| SonarQube | 10.3 | cpp:S5547 | Warns about missing zeroization and secure disposal of sensitive in-memory data |

### 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.

**Explanation:**

At Green Pace, we’re aiming to keep security consistent and efficient by building it directly into our development workflow. Instead of relying on manual checks at the end, we use automation to catch problems early and enforce standards as code moves through the DevSecOps pipeline. During the Plan and Create stages, developers are supported by IDE-based tools like Clang-Tidy and SonarLint. These tools provide real-time feedback as developers code, allowing them to catch and fix issues early—before anything even gets committed. It’s a simple way to build good habits and avoid introducing risky patterns. In the Verify stage, we bring in tools like Cppcheck, SonarQube, and Fortify SCA through our CI/CD process. These tools handle static analysis and scan for things like memory leaks, injection risks, and unsafe API calls. They help ensure that our code meets the secure coding standards we’ve defined—every time we push a build. Before deployment, in the Pre-Production stage, we use Valgrind and runtime sanitizers to stress test memory usage, alongside fuzz testing for unpredictable inputs. This helps us catch deeper issues that might not show up during regular testing. From Release through Respond, we use automation to validate signatures, run integrity checks, and monitor traffic and user behavior. Everything is logged and auditable so we can spot anomalies and respond fast if something doesn’t look right. Overall, automation gives us confidence that we’re shipping secure software—without slowing down our devs. It keeps us compliant, consistent, and ready for audits without creating extra friction.

### 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 | Medium | Likely | Low | Medium | 4 |
| STD-003-CPP | High | Likely | Low | High | 5 |
| STD-004-CPP | High | Likely | Low | High | 5 |
| STD-005-CPP | High | Likely | Medium | High | 5 |
| STD-006-CPP | Medium | Unlikely | Low | Medium | 3 |
| STD-007-CPP | High | Likely | Medium | High | 5 |
| STD-008-CPP | High | Likely | Medium | High | 5 |
| STD-009-CPP | High | Likely | Low | High | 5 |
| STD-010-CPP | Medium | Unlikely | High | Medium | 3 |

### Create Policies for Encryption and Triple A

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

* 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 | This policy requires all sensitive data stored on disk (e.g., databases, backups, logs) to be encrypted using AES-256 or stronger. This ensures that if a device or disk is stolen or accessed without authorization, the data remains protected. It applies to storage servers, local drives, and cloud-based data repositories. |
| Encryption in flight | All data transmitted over public or internal networks must be encrypted using secure protocols like TLS 1.2+ or SSH. This protects against eavesdropping, man-in-the-middle attacks, and session hijacking. It applies to web apps, APIs, and internal services communicating across networks. |
| Encryption in use | This policy applies to the protection of sensitive data while it is being processed in memory. Where possible, data should be processed inside secure enclaves (e.g., Intel SGX) or with memory encryption techniques. It's especially important in multi-tenant cloud environments or when handling credentials, encryption keys, or personally identifiable information (PII). |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication ensures that only verified users or systems can access Green Pace resources. All user logins must use multi-factor authentication (MFA) and secure password policies. This helps prevent unauthorized access through stolen or weak credentials. |
| Authorization | Authorization controls what an authenticated user can do. Role-based access control (RBAC) must be used to limit access to data and functions based on the user’s job role. This minimizes risk and enforces the principle of least privilege. |
| Accounting | Accounting refers to logging and monitoring user actions within the system. It includes tracking logins, data access, system changes, and failed login attempts. These logs help detect suspicious behavior and support forensic investigations in the event of a breach. |

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

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

### Map the Principles

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

**NOTE:** Green Pace has already successfully implemented the following:

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

|  |  |  |
| --- | --- | --- |
| Standard | Principle(s) | Justification |
| **STD-001-CPP** Range Verification | 1. Validate Input Data 4. Practice Defense in Depth | Ensures indexes are checked before use to avoid crashes or memory corruption. Adds a layer of protection even if earlier validations fail. |
| **STD-002-CPP** Data Value Validation | 1. Validate Input Data 6. Use Effective QA Techniques | Prevents invalid data from triggering logic errors or vulnerabilities. It’s also easy to test for these edge cases. |
| **STD-003-CPP** String Correctness | 1. Validate Input Data 2. Sanitize Data Sent to Other Systems | Unsafe strings can corrupt memory or allow code injection. Sanitizing and validating strings ensures they're safe before use or transmission. |
| **STD-004-CPP** SQL Injection | 1. Validate Input Data 2. Sanitize Data Sent to Other Systems | Input that reaches SQL queries must be cleaned and handled safely to prevent manipulation. These principles directly address that. |
| **STD-005-CPP** Memory Protection | 4. Practice Defense in Depth 3. Adopt a Secure Coding Standard | Using smart pointers and proper memory cleanup limits memory-related vulnerabilities. Following standards helps avoid these mistakes. |
| **STD-006-CPP** Assertions | 5. Fail Securely 6. Use Effective QA Techniques | Assertions provide self-checks in dev/test environments and help detect failures early. But they should never replace real error handling. |
| **STD-007-CPP** Exception Handling | 5. Fail Securely 3. Adopt a Secure Coding Standard | Properly handled exceptions help systems fail gracefully and prevent information leaks. |
| **STD-008-CPP** Encryption at Rest | 7. Protect Data 4. Practice Defense in Depth | Encryption keeps stored data safe even if storage is compromised. It supports layered defense strategies. |
| **STD-009-CPP** Encryption in Flight | 7. Protect Data 8. Secure the Weakest Link | Data in motion is highly vulnerable. Encrypting it secures a common weak spot in system design. |
| **STD-010-CPP** Encryption in Use | 7. Protect Data 9. Minimize Attack Surface | Protecting data during processing limits risk from memory scraping and advanced attacks like side-channel exploits. |

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 04/07/2025 | Finalized security policy for Project One | Buxton McCaslin |  |
| [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 |